# Formal Specification and Verification of Secure Information Flow for Hardware Platforms

*Kevin Cheang*

# Formal Specification and Verification of Secure Information Flow for Hardware Platforms

by

Kevin Cheang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Krste Asanović
Associate Professor Alvin Cheung
Professor Emeritus David L. Dill

Summer 2023

Formal Specification and Verification of
Secure Information Flow for Hardware Platforms

## Abstract

Formal Specification and Verification of
Secure Information Flow for Hardware Platforms

by

Kevin Cheang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Hardware platforms, such as microprocessors and Trusted Execution Environments (TEEs), aim to provide strong memory isolation properties. However, in recent years, this has been shown not to be the case through hardware attacks such as the class of transient execution attacks. These attacks affect programs executing on widely-used microprocessor designs in our present-day devices. Although mitigations have been proposed, many have not been adopted and lack formal guarantees. As a result, security-critical applications have been conservative in using hardware platforms without some form of cryptographic approach for secure computation, despite the additional computational overhead. One approach to ensure safety for this class of attacks is to use formal methods to prove information flow properties. Yet, there is limited work in verifying attacks on hardware platforms that are heterogeneous in nature, namely those that contain hardware and software in the trusted computing base.

This thesis defines a notion of secure information flow for hardware platforms and proposes methods to formally verify non-interference-based properties efficiently using abstractions and composition. To accomplish the former, we formalize the *trace property-dependent observational determinism* property for capturing a new class of non-interference properties. This property is motivated by verifying transient execution attacks and the need for *secure speculation*. To enable efficient verification on hardware platforms, we introduce an efficient proof system, *SymboTaint*, and the formalism of *information flow state machines* to reason about secure information flow compositionally. Finally, we explore a complementary method to enforce secure information flow for general programs by relaxing the programming model of a family of TEE designs and by formally verifying them. This direction builds on top of existing abstractions of TEEs to provide memory isolation guarantees with an efficient memory-sharing scheme on TEEs through combined design and verification. Together, this provides a methodology for enforcing memory isolation for heterogeneous systems, where joint modeling and analysis of hardware and software have become imperative for security.

To my parents.

# Contents

# List of Figures

# List of Tables

## Acknowledgments

Throughout the years, I have had the privilege of crossing paths with numerous researchers, mentors, and individuals, each of whom added unforgettable moments to my life. Whether it was chatting with colleagues both inside and outside the office, pulling countless all-nighters alongside my co-authors and classmates, having discussions with researchers at conferences and workshops, or simply spending time with friends who were always there for me, each one of them contributed to my growth and filled my life with an overwhelming amount of gratitude and meaning. It is only proper that I express my gratitude to everyone who played such a significant role, that ultimately led to this monumental moment in my life.

First and foremost, I thank my advisor Sanjit A. Seshia, for his life-changing guidance and support, for which I am eternally grateful. Among the myriad of things that he has done for me, there are a few that stand out that I would like to acknowledge here. When I first entered the lab, I was a rather reserved individual. Sanjit immediately took note of this with empathy, and over the years, he consistently encouraged me to engage with fellow researchers, taught me how to promote my research and emphasized its importance, and gave me opportunities that propelled my growth as a researcher and on a personal level. I feel very fortunate to have worked with Sanjit and give him my deepest gratitude. I also thank the other members of my qualifying exam and thesis committee for providing invaluable guidance along the way: Krste Asanović, Alvin Cheung, and David Dill.

I thank my co-authors, all of whom I had a great deal of pleasure working with. First, I thank Pramod Subramanyan. From his contributions to the formalism of *TPOD* and development of UCLID5, to all the graduate student advice he gave me when I first started, his influence is what made much of my research possible. I thank Cameron Rasmussen, Pranav Gaddamadugu, Dayeol Lee, and Adwait Godbole for all the intellectually stimulating conversations and fruitful discussions that led to a lot of writing, paper submissions, conferences, and never-ending laughter. I also thank the folks in the UCLID5 team for all UCLID5 related collaborations: Federico Rocha, Elizabeth Polgreen, Yatin A. Manerkar, and Shaokai Lin. I also need to thank many of my lab mates (who were not already mentioned above), Ben Caulfield, Tomasso Dreossi, Daniel Fremont, Shubham Goel, Markus Rabe, Sumukh Shivakumar, Edward Kim, Shiv Kushwah, Hazem Torfah, Marcell Vazquez-Chanlatte, Sahil Bhatia, Pei-Wei Chen, Niklas Lauffer, Shaokai Lin, Ameesh Shah, Victoria Tuck, Justin Wong, and Beyazit Yalcinkaya, for making my time in the office *never* a dull moment.

During the course of my time here, I was also fortunate to have met many great industry mentors and colleagues that have given me indispensable graduate student advice while also teaching me a vast amount in my research area. From my internships at Facebook/Novi, I thank David Dill (again), Shaz Qadeer, Clark Barrett, Wolfgang Grieskamp, Evan Cheng, Sam Blackshear, and Jingyi Emma Zhong. From my internship at Intel, I thank Robert Jones and Amit Goel. I also extend my gratitude to the folks at Intel who provided insightful feedback on my work in our monthly meetings.

Naturally, I could not have done this without the folks back at home. I thank my parents for the countless sacrifices they made over the decades and for their unwavering support which helped pave the way for this moment. I also thank my brother for taking care of our parents while I was away for all these years. Additionally, I owe a debt of thanks to Alan J. Hu and Mark R. Greenstreet, as it was their initial support and encouragement that led me to start my journey here at Berkeley.

Finally, to all the individuals (including those unnamed for the sake of privacy) who have blessed me with the strength to persevere through this degree, *thank you*.

# Chapter 1

# Introduction

## 1.1 The Rise of Insecure Hardware Platforms

*Hardware platforms* such as the microprocessors in our phones, laptops, and web servers, aim to provide strong *memory isolation guarantees* to protect a user's confidential data from being leaked to untrusted entities (Figure 1.1). For example, a user interacting with a web application from their phone would involve using an input/output (I/O) interface to control how their application executes and responds. In this interaction, the user may provide sensitive data to their application such as credit card numbers or information about their identity. Naturally, a user would hope that untrusted entities executing on the same hardware platform, such as other applications, are unable to infer *any* information about their confidential data. However, the discovery of a number of hardware-based security flaws — in particular, *transient execution vulnerabilities* such as Spectre [116] and Meltdown [133] — in widely-used present-day microprocessors, has raised major security concerns about the memory isolation guarantees between programs running on these platforms. If we are

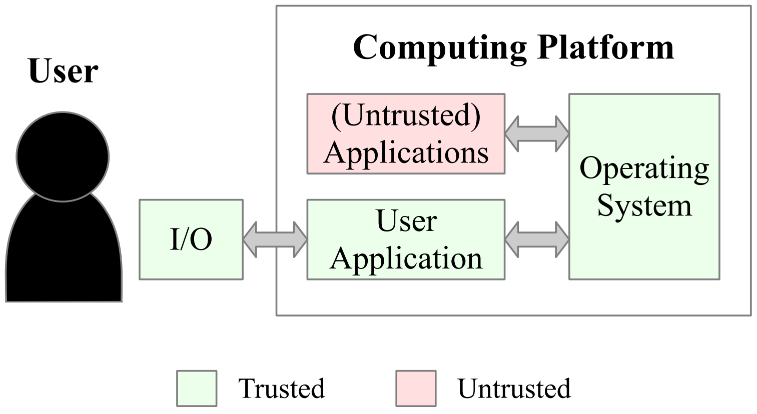


Figure 1.1: User executing their application on a platform with untrusted applications and the operating system.

not careful, untrusted entities that share the platforms where we store our data have the potential to exploit these newly found attack vectors in order to gain access to our confidential information.

One approach to counter these attacks is by using *formal methods* to establish formal proofs of memory isolation on the platform. Formal methods is an area of computer science that relies on rigorous mathematical proofs to provide system correctness through specification, modeling, and verification. Over the last several decades, formal methods have made great advances in becoming practical and have become extensively used in the industry for proving system correctness. One prominent technique in formal methods is *model checking*, which is used to mechanically prove properties about finite state machines. We refer to these proofs as *formal guarantees*, which is often a reliable measure of system correctness.

Unfortunately, it is becoming increasingly difficult to ensure that hardware platforms are secure due to their growing complexity. To exacerbate the situation, new vulnerabilities are discovered every year, requiring non-trivial changes to the hardware platform [63] in order to protect against new attacks. Even without the added complexity, formal guarantees of security such as memory isolation for hardware platform designs often lag years behind their commercial releases. The reason for this lies predominantly in the reliance on manual effort when applying formal methods to hardware platform designs, as exemplified by work such as seL4 [114] and Komodo [70]. It is immensely challenging and costly to manually apply formal verification to a hardware platform because of the sheer size of the design.

On the other hand, *secure computation* on *secure* hardware platforms [53, 101, 112, 127, 141, 161, 200] has been of growing interest for security-critical applications in recent years. For instance, machine learning inference models [91], blockchain technology [67, 232, 234], and serverless computing [71, 160] have become ubiquitous, and many of these applications require a high degree of confidentiality for the underlying data used for computation. These applications execute on hardware platforms and require that they provide strong memory isolation guarantees. Unfortunately, the only viable hardware platforms that we can rely on to provide memory isolation have become a prime target for transient execution attacks and more generally, what we refer to as *hardware attacks* [29, 116, 133, 174, 186, 189, 215, 216, 224, 225]. As a result, the gap between the demand for secure hardware platforms and the actual level of security provided has been widening. This has led to a noticeable disparity that leaves security-critical applications with no other option but to resort to less performant alternatives for enforcing memory isolation, such as cryptographic-based methods [74].

Mitigations proposed [89] for hardware attacks have mostly been hardware- or software-based, yet there is a barrier to adoption. Hardware mitigations are often regarded as too intrusive and vulnerability-specific. On the other hand, the security of software mitigations depends on the hardware it runs on. This frequently leads to platform-specific software mitigations (e.g., using x86 serializing instructions [102]) and warrants more precise analysis that includes details about the hardware, which are often lacking in many analysis methodologies. In addition, new side-channel attacks [40] have repeatedly found a way to circumvent proposed mitigations. Time has proved that this class of attacks is difficult to overcome with hardware- and software-based mitigations alone.

Alternatives to secure computation include cryptographic-based alternatives and hardware-based isolation techniques. The former includes techniques such as secure multi-part computation and homomorphic encryption, however, these techniques have been shown to be computationally expensive, even with hardware accelerators [182]. On the other hand, hardware-based isolation techniques implemented by platforms such as *trusted execution environments* (TEEs) [52, 53, 127] have been of growing research interest [67] to the community as they are more performant. Unfortunately, they lack maturity in feature richness and assume a very restrictive programming and memory-sharing model which severely limits usability.

The culmination of these barriers to secure computation on hardware platforms leads to one possible solution: to develop and improve existing formal techniques that can provide hardware platforms with formal guarantees in a practical manner. However, this direction comes with major challenges that we discuss in the next section. Broadly speaking, with the current advances in secure computation technology and the discovery of new vulnerabilities, the *trusted computing base* of hardware platforms, the part of a system that is critical to security, has grown to include both software and hardware components. We refer to these mixed component systems as *heterogeneous* systems, which because of their complexity, are inherently difficult to both model and analyze using traditional methods.

For the remainder of this chapter, an overview of the problems addressed in this thesis is presented. Specifically, section §1.2 discusses the formal methods-based approaches used and section §1.3 outlines the challenges of applying formal methods to hardware platforms. Section §1.4 presents the thesis statement and summarizes the contributions of this thesis. Lastly, we conclude with acknowledgments in section §1.5 to thank those who have funded my work and played a crucial role in making this research possible.

## 1.2 Secure Information Flow

The security of hardware platforms relies on a core component that defines the behavior of the software programs that they execute: the microprocessor. To safeguard against the attacks mentioned earlier using formal methods, we need to both model and verify properties defined over microprocessor designs. Specifically, we target speculative microprocessor and TEE models where strong memory isolation is essential. To get a better sense of how one can model these platforms and use them to prove memory isolation, it is important to understand the types of components in these systems that are exploited and how they are exploited in hardware attacks to even bypass the more resilient security measures.

Microprocessors and TEEs are types of heterogeneous systems that rely heavily on both software and hardware for security. Perhaps unsurprisingly, hardware attacks exploit both of these components of the platform to leak secrets, explaining the elusiveness of these vulnerabilities. For instance, in transient execution vulnerabilities, a microprocessor implements speculative features to increase throughput, but these features inadvertently cause unintended executions of instructions that lead to secrets being leaked to covert channels such as the data cache. Clearly, proof of memory isolation needs to include the cache along with the exploited program. Similarly for TEEs, even though they rely on specialized hardware to enforce memory isolation, speculation can be exploited to break security [215, 224]. Thus, in addition to the exploited components in a microprocessor model, one also needs to consider the hardware mechanisms specific to TEEs that are used to strengthen memory isolation.

Integrating both hardware and software components into a single model requires them to share the same programming model. As a result, a significant challenge in formalizing a hardware platform model is precisely defining this unified programming model. First, one needs to determine the level of abstraction at which the models are described at. Ideally, the abstraction level should be capable of describing both the hardware and software components being exploited. For instance, one such level of abstraction is the instruction level. Second, one needs to determine the operations of the model and the granularity at which they execute to change the system state. For example, updates to hardware components are typically driven by a clock signal that concurrently and atomically updates the components connected to that clock signal. On the other hand, software does not have a notion of a clock signal and thus the atomicity of program execution that should be modeled is less obvious. In light of these differences, it is natural to consider a *transition system*-styled model in which each atomic operation of the system can be chosen to suit the assumptions of the application and the associated attacker model. The work presented in this thesis assumes this modeling style. We now turn to the type of attacker model needed to model these attacks and the security properties we need to prove.

**A general attacker model**. The rate of discovery of new vulnerabilities is concerning as it has implications for the attacker model. Historically, attacker models have been fixed for a given vulnerability of a hardware platform. However, this approach has proven to be increasingly limited in its productivity and effectiveness. Often, checking for the security of a single adversary model is not sufficient for sound guarantees of security on a hardware platform. To scale, we must consider models that are parameterized and allow us to capture classes of attacks. For that reason, the attacker models considered throughout the thesis is one that is general and has the following two major characteristics. One, we assume that it can execute an unbounded number of steps whenever the platform allows it to execute. Two, we assume that it has access to an abstract *observation function* and *tamper function* which defines the components that it can observe and modify when it executes.

**Enforcing secure information flow**. Finally, in order to verify memory isolation for hardware platforms, it is necessary to prevent any information leakage from secret states to observable states, irrespective of the adversary's actions. One way to capture this notion of security is by using the standard *non-interference* [76] security property based on information flow. Non-interference is used to restrict the information flow of a system; it states that a group of users executing a set of commands on a system does not affect what another group of users (on the same system) can see. This thesis presents extensions of the non-interference property, which we collectively refer to as *secure information flow* (SIF). We will describe these properties briefly in Chapter §2.3 and define them formally in Chapters §4, §5 and §6.

## 1.2.1 A Workflow that Guarantees Secure Information Flow

While we have already mentioned the applications that motivate our work, the central focus of this thesis is really motivated by the vision of a workflow that provides secure information flow of programs executing on hardware platforms. Naturally, explaining this workflow will provide insight into the research directions of our work.

The workflow begins with the consideration of using a prominent hardware platform that aims to enforce memory isolation without incurring a large performance penalty: trusted execution environments. TEEs use hardware primitives to ensure that *enclave* program memory is protected from untrusted entities executing on the platform. Examples of TEEs include Intel SGX [141] and Trust Domain Extensions [101], ARM TrustZone [10], Keystone [127], MIT Sanctum [53], and AMD SEV [112], many of which exist in the processors of major computer hardware manufacturers. However, existing TEEs have been discovered to be vulnerable to various side-channel attacks such as Foreshadow [215, 224]. Moreover, TEEs have a restrictive programming model that limits how they can share memory. Without memory sharing, enclave programs are very limiting for typical user applications. Thus, it is crucial that these issues are first addressed for the widespread adoption of TEEs. This leads us to the following observation. If we have a methodology to prove that a class of enclave programs is protected against hardware attacks (i.e., by proving SIF) and if we can extend the functionality of enclaves to capture a wider family of applications, then TEEs can become a practical approach for enforcing memory isolation of more general applications. This is depicted in Figure 1.2.

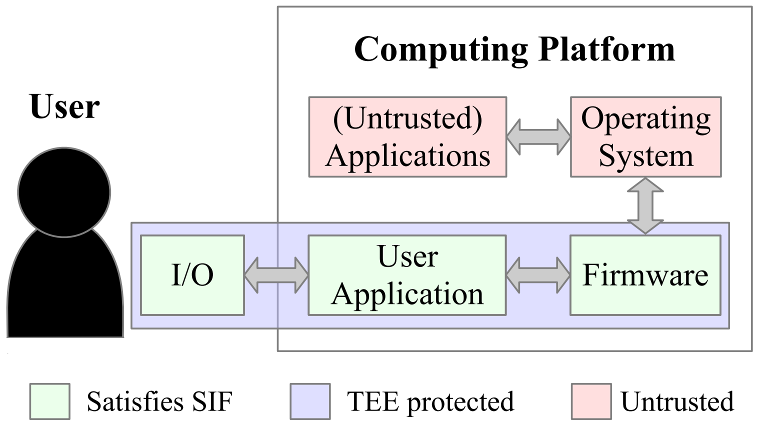


Figure 1.2: Application protected by the TEE and formal guarantees of secure information flow.

To better understand the complexity of developing such a methodology and extending TEEs securely, we describe the models needed to prove SIF on such a workflow in more detail.

At the core of these hardware platforms, there is an architectural model that describes how the platform state evolves at a functional level as it executes. However, the extent of security that a purely architectural model can provide is limited to functional correctness. Naturally, it is not unless we also consider the other component of the heterogeneous system — the implementation-based microarchitectural model — that we can reason about hardware-based attacks such as transient execution vulnerabilities. Below, we discuss some key aspects that should be taken into account when verifying hardware platforms.

**Microprocessors**. A microprocessor design consists of the computer architecture, on which an instruction set architecture (ISA) [9, 103, 221, 222] is used to define the semantics of how a program controls the CPU and the microarchitecture which implements these semantics. Historically, methodologies have often focused on verifying the functional correctness of microprocessor designs [36, 70, 199] and thus only modeled the architectural state of a microprocessor. However, with transient execution attacks, new formal models of processors have been increasingly microarchitecture-aware [45, 125, 153, 204]. Microarchitectural buffers and caches such as the pattern history table, page tables, reorder buffer, translation lookaside buffer, caches, store, and load buffers have become intrinsic to the formal analysis of hardware platform security, and there is an increasing trend toward unifying program semantics and hardware behavior as a monolithic model. As one can imagine, direct translations or implementation-accurate models of hardware implementations in hardware description languages (HDL) such as RTL or Verilog are often too complex and requires complete remodeling whenever the hardware design is changed. Thus, our work considers a middle-ground model that abstractly models microarchitectural components. This allows the model to capture a class of microarchitectures while also being amenable to changes due to its simplicity. As a transition system, this translates to modeling each instruction from the ISA as an operation that atomically updates the architectural and microarchitectural components according to the ISA semantics and hardware design.

**Trusted execution environments**. Instead of relying on the operating system to isolate memory, a trusted execution environment provides a set of operations for users to manage and execute their *enclave* program within a protected area of the platform [10, 101, 127, 141]. From a modeling standpoint, one may extend the microprocessor model described above to execute the set of enclave operations. Similar to microprocessors, enclave platforms modeled without hardware components have left it open to speculative attacks [215].

## 1.3 Challenges

As alluded to in the previous section, verifying non-interference properties on hardware platforms presents several challenges. To better understand these challenges, it is useful to examine the desired characteristics of a formal approach that checks non-interference on hardware platforms against a class of attacks. Three key requirements emerge in verifying these platforms: *accuracy*, *reusability*, and *scalability*. Accuracy is needed to precisely explain the underlying cause of a vulnerability, while reusability and scalability allow an approach to adapt to changes and scale to realistic problems. While we desire our approach to incorporate these aspects, it is challenging to have all three. Thus, the crux of this thesis is also about finding the right balance of abstractions and developing techniques to satisfy these requirements for hardware platforms.

### 1.3.1 Accuracy

Accuracy is imperative to any formal approach. We refer to two types of accuracy in an approach. The first type of accuracy is about precisely capturing the desired property, such as the class of attacks or vulnerabilities. The second type refers to the soundness and completeness of the procedures used to determine security.

**Formalizing secure information flow**. Proving security requires one to first define a security criterion. The first part of this criterion is the attacker model. The second and more subtle part is how one specifies a property to capture the vulnerabilities or attacks of interest. For example, checking that a program satisfies non-interference is not as informative as checking for a violation of non-interference due to optimizations in a platform. A more informative property is desirable because it can more precisely capture and explain the violation of a given property. In addition, the standard non-interference property is often too strong to be practically enforced and is violated by most programs whether or not they are vulnerable to hardware attacks.

**Soundness and Completeness**. As in any standard decision procedure for determining the security of a system, we require soundness at a minimum, which states that if the procedure says the system is secure, then the system is truly secure. While soundness is necessary and fundamental in our approaches, an approach is only practical when it has a minimal number of false positives. That is, it cannot be too overapproximate and return an excessive number of incorrect results. Overly abstract models in formal verification, for instance, are often prone to resulting in false positives. Thus, one challenge of writing abstractions is determining the right level of detail. Conversely, completeness states that the procedure returns an answer whenever one exists. We acknowledge that while completeness is desirable, this thesis does not address this aspect.

### 1.3.2 Reusability

With the rapidly growing number of hardware attacks discovered every year and the growing complexity of the systems on which they exploit, an approach needs to be reusable and scale to the growing pace of the system. These challenges can be alleviated by considering parameterizability and automation.

**Parameterizability**. In writing models, we must also consider how various hardware attacks can be expressed because the security of a hardware platform design (e.g., a microprocessor [108]) is dependent on the individual components and optimizations implemented. Ideally, formalisms are defined in a way that allows models to check a range of attacks beyond a single variant that it was designed for so they can be reused whenever there are slight changes to the attack vector or design. Thus, modeling and specification formalisms should be compositional and easily parameterizable. Additionally, this would allow the models to adapt to new changes, which is crucial for formal models that would often take an expert, months to write.

**Automation**. While parameterizability enables us to scale to a range of attacks and platforms, automation is typically necessary to maintain the usage of the approach practically. Hardware platforms contain thousands of lines of code at a minimum, either as firmware or as part of the application that we are verifying. As a result, our approaches should ideally automatically generate models and proofs from the implementation.

### 1.3.3 Proof Scalability

Lastly, parameterizability brings issues with proof scalability because of the complexity of the platform. This is commonly addressed through model abstraction and compositional reasoning.

**Reducing model complexity with abstraction**. Formal models can still be fragile when used with state-of-the-art formal method-based engines, which may not always terminate. A model of a complex system must have the right level of abstraction. For starters, this can help alleviate the modeling effort required to compose different components of a system together. Determining which parts of the model to abstract away is also important for reducing the runtime of these engines [32, 33]. Second, the encoding of these models needs to be efficient. Formal models are often written as logical formulas in Satisfiability Modulo Theories [17, 18] (SMT). The SMT problem is the problem of determining whether a given logical formula is satisfiable. This problem is solved in practice by using SMT solvers, which are often used in formal method engines [57, 129, 169]. The formulas compiled from the security questions we ask, often contain a range of theories in SMT and can have alternative encodings that are equivalent or approximate, each of which can vary drastically in runtime when given to the solvers. As such, an efficient encoding [30, 177, 211] with a suitable logic or set of theories for complex models is imperative in preventing state explosion and avoiding undecidable logical encodings [27]. We acknowledge that there is an obligation to check that the abstract model is a sound abstraction of the implementation model [49], however, this is not within the scope of this thesis.

**Scaling proofs by composition**. Lastly, the method of proof needs to be efficient. For example, executing bounded model checking on a large model will likely result in state explosion for symbolic model-checking techniques. Thus, designing a methodology that can take advantage of assumptions and security criteria to construct compositional proofs is often necessary to scale to realistic programs.

To that end, we present the thesis statement and a list of contributions that overcome these challenges to address the problems presented in §1.1. Through this thesis, we aim to take a step towards realizing what we can call *secure hardware platforms*.

## 1.4 Thesis Statement

*We develop methodologies to prove secure information flow by formalizing abstract models of hardware platforms, formalizing trace property-dependent observational determinism to precisely capture classes of hardware attacks, and introducing an interpolant-based proof methodology to efficiently verify heterogeneous systems against hardware attacks.*

### 1.4.1 Contributions

A summary of the research contributions to address this thesis statement includes the following:

1. Chapter §2 provides background on secure hardware platforms and the applications of this thesis.

2. Chapter §3 provides background on the general approaches used, relevant concepts from formal methods, and a primer for the verification toolkit used throughout this thesis, Uclid5 [169, 193].

3. Chapter §4 presents the *secure speculation* property, used to capture information flow leaks caused by transient execution attack vulnerabilities. Furthermore, we generalize this property as *trace property-dependent observation determinism.*

4. Chapter §5 generalizes the approach from the previous chapter to efficiently verify information flow-based properties using the *SymboTaint* proof system and *information flow state machines.*

5. Chapter §6 returns to the problem of hardware-based secure computation and provides a methodology for combined design and verification of secure and efficient trusted execution environments.

6. Chapter §7 concludes with discussions about the existing work and future directions.

## 1.5 Acknowledgements

Over the years I have had the honor of publishing work with collaborators who have been immensely helpful, insightful, and a pleasure to work with. This thesis is a compilation of these works that we wrote together jointly. In this section, I describe their contributions and acknowledge the sources of funding which made this thesis possible.

### 1.5.1 Collaborative Work

Chapter §3 presents a primer to UCLID5 and is partially based on the paper "*UCLID5: Multi-Modal Formal Modeling, Verification, and Synthesis*" [169], which is a joint contribution with Elizabeth Polgreen, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia who designed and developed the UCLID5 verification tool, whose initial version were created by Pramod Subramanyan and Sanjit A. Seshia [193].

Chapter §4 is based on " *A Formal Approach to Secure Speculation*" [46]. Cameron Rasmussen contributed by having countless discussions with me and by helping out with the experiments. Pramod Subramanyan contributed by proposing TPOD, in addition to providing lots of guidance and feedback with Sanjit A. Seshia.

Chapter §5 is based on "*Compositional Proofs of Information Flow Properties for Secure Platforms*" [44]. Adwait Godbole contributed by having countless discussions with me and by helping formalize the SymboTaint proof system and information flow state machines. Yatin A. Manerkar and Sanjit A. Seshia contributed by providing invaluable guidance and feedback throughout this project as well.

Chapter §6 is based on "*Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing*" [125]. Dayeol Lee contributed by having countless discussions with me, designing Cerberus, and implementing it in Keystone. Pranav Gaddamadugu contributed by translating the TAP model from Boogie to UCLID5 and attempting to formally verify the initial version of Cerberus [72]. Alexander Thomas and Catherine Lu for making significant contributions to the experimental sections and the implementation of Cerberus. Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanović contributed by providing invaluable guidance and feedback.

### 1.5.2 Funding

The research described in this thesis was supported in part by SRC tasks 2867.001 and 2854.001, the iCyPhy Center, NSF grants CNS-1739816 and CNS-1646208, the Science and Engineering Research Board, a gift from Microsoft Research, the Qualcomm Innovation Fellowship, Amazon, Intel under the SCAP and Scalable Assurance program, and by RISE, ADEPT, and SLICE Lab industrial sponsors and affiliates.

# Chapter 2

# Hardware Platforms

Hardware platforms consist of many layers, from the I/O layer and networking layer to data buses and individual hardware intellectual properties, but the core component that is generally responsible for security is the microprocessor that executes the application software on our devices. Consequently, our primary focus is *soley* directed towards ensuring security for microprocessor-based designs. Although microprocessor security has been extensively studied, it wasn't until recently that reasoning about hardware and software together has become more conventional for security [135, 204, 209] on hardware platforms. In this section, we provide background for microprocessor models and TEE models to illustrate how modeling has changed, the vulnerabilities of the respective platforms, and briefly discuss the attacker models.

## 2.1 Speculative Microprocessors

Ever since processors began to adopt out-of-order execution, speculation, and superscalar optimizations [164, 195, 200], microprocessors became increasingly complex. We describe this complex architecture below and explain how it has become increasingly vulnerable over the years.

### 2.1.1 Breaking Down The Microprocessor Design

A microprocessor design can be thought of as a dyadic of abstraction layers: the architectural and the microarchitectural designs. The former can be used to describe how a program is computed and the latter how the underlying computation engine is implemented. We describe each of these layers below.

**Architectural design**. A microprocessor is designed to adhere to what is called the architectural model, which includes a set of registers such as the program counter, general purpose integer registers, control status registers, and floating point (FP) registers, in addition to an *instruction set architecture* (ISA) [9, 11, 103] that defines the semantics of how these registers change values as an instruction executes through the processor pipeline.

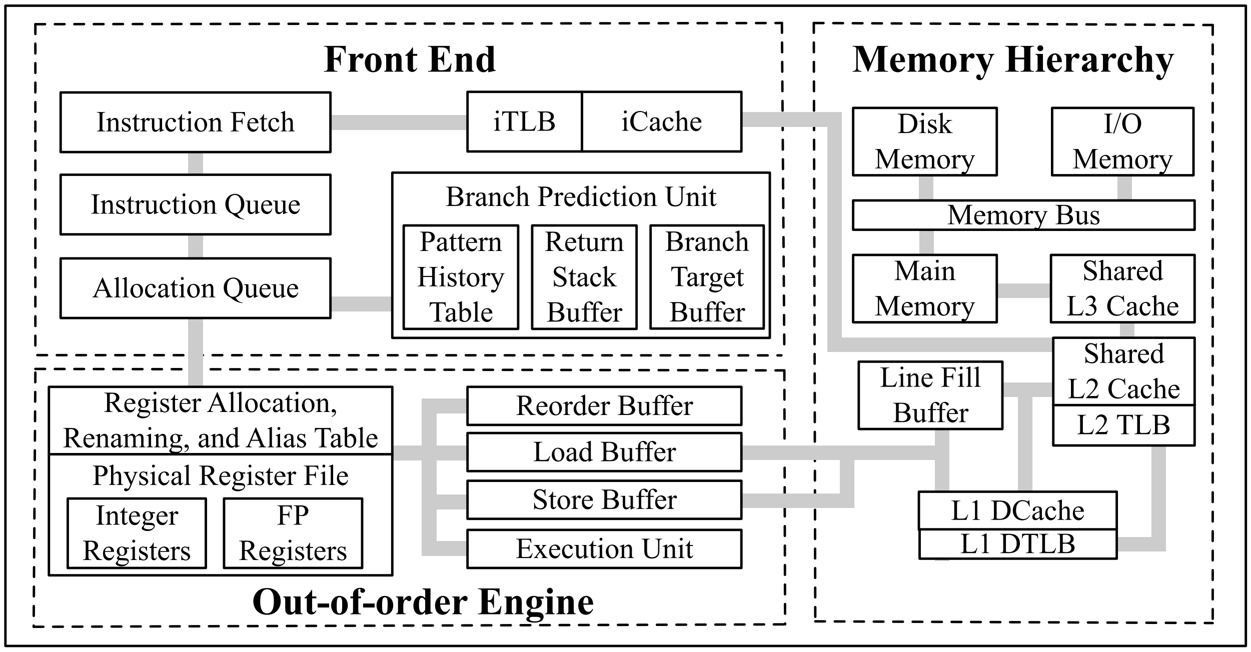


Figure 2.1: Example speculative microprocessor design [104] with a front-end component, the memory hierarchy, and an out-of-order engine. Arrows represent data flow from one state to another.

**Microarchitectural design**. Buffers, caches, arithmetic logical units, and other digital logic blocks constitutes the actual implementation of the processor. As each instruction is executed by the processor pipeline, each component is used to compute the next architectural state. More precisely, the microarchitecture includes all the components that implement the logic of how an executed instruction affects the registers, memory, arithmetic logic and other states defined in the architecture. As an example, Figure 2.1 depicts a design of a microprocessor (described at some arbitrary level of abstraction) with three main parts consisting of the front end, the memory hierarchy, and the out-of-order engine. The front end deals with anything related to instruction fetch, instruction decode, and register allocation, which includes logic used to determine the next instruction to execute such as the branch predictors. The out-of-order engine is part of the microprocessor that deals with asynchronous events that may occur out-of-order such as the reorder buffer and load/store buffers. Lastly, the memory hierarchy optimizes the speed at which memory accesses and writes are made using caches.

### 2.1.2 A Decades-Old Flaw: Transient Execution Vulnerabilities

Unfortunately, the introduction of optimizations and buffers mentioned in the previous section has led to numerous *side channels* that can be used by an attacker as a data extraction channel of a victim process. On top of that, extraction is made simpler because of speculation.

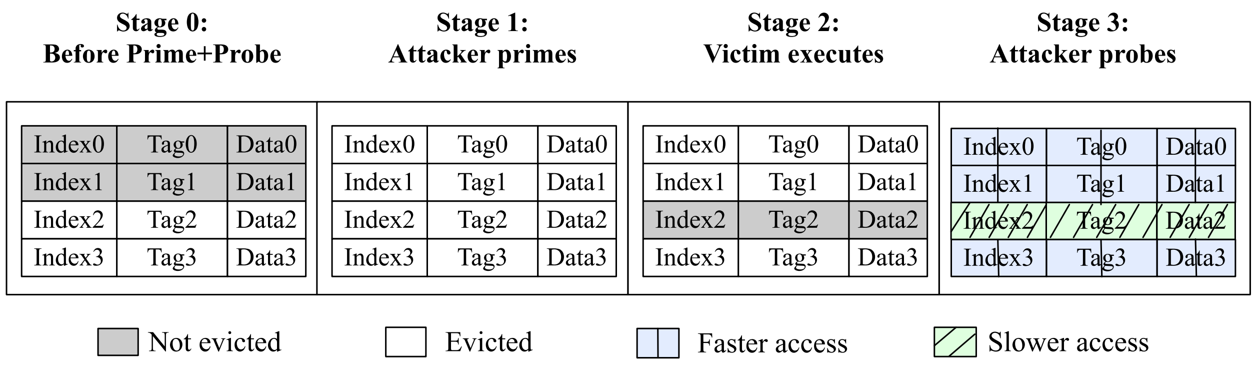


Figure 2.2: Stages of the Prime+Probe attack assuming a 4-way set associative cache. In the first stage, the attacker primes the cache by making memory accesses from their address space whose addresses map to the cache lines of a set. This invalidates all entries in a cache set. In the second stage, the victim executes, resulting in an evicted cache line. Finally, the attacker times the same memory accesses to determine information about which address the victim accessed.

**Creating a side channel for an attacker**. The discovery of new side-channel retrieval techniques has enabled attackers to retrieve information about sensitive data more efficiently over the years. Attackers can use these techniques to break address space layout randomization techniques and retrieve information about the execution paths of programs to break encryption schemes. For example, *Flush+Reload* [228] and *Prime+Probe* [134] are prominent techniques that reveal the address of a victim's memory accesses to an attacker. The attacker accomplishes this in three stages as shown in Figure 2.2, which depicts the Prime+Probe attack for a given *set* of the cache. We describe these three stages below.

*Stage 1.* In the first stage of the attack, the attacker first constructs an *eviction set* using a buffer that is accessible in their own address space. The attacker uses this set to evict every cache line from the cache set with their data. In a later step, the attacker uses the eviction set to determine which addresses the victim has accessed.

*Stage 2.* In the second stage of the attack, context switches to the victim which executes their code. As they execute, they make memory accesses whose contents are eventually cached at some level of the memory hierarchy.

*Stage 3.* In the final stage of the attack, the attacker makes memory accesses using the same eviction set constructed in the first stage and times each access. If access to a particular memory address in the eviction set is *fast*, then this corresponds to accessing a cache line that was previously accessed during the prime stage and nothing is learned. However, if the access to a memory address in the eviction set is *slow*, this means that the corresponding cache line was evicted during victim execution. Consequently, this reveals to the attacker information about the address that was accessed by the victim. The attacker can then use this information to infer the secret-dependent execution path taken in algorithms such as square-and-multiply and AES, revealing information about secrets.

We note that this attack has been shown to be effective even for the last-level cache, thus it is even easier for attackers to retrieve information from the L1D cache if the attacker and victims execute on the same physical core.

**Speculation**. The remaining element that is fundamental to reproducing transient execution attacks is speculation in a processor. Speculation allows a microprocessor to *guess* the value of a particular result currently being computed in the pipeline and proceeding without knowing what the actual value is. In the event that the speculated value is incorrect, the processor reverts its state to a former state before it made the speculation. This technique allows the microprocessor to increase instruction throughput and ultimately improves the processor's performance. Examples of speculation include branch prediction, value prediction, and store-to-load forwarding [195].

**Combining speculation with side channels**. Together, side channels and speculation in microprocessors form a new class of vulnerabilities called transient execution vulnerabilities that existed for over a decade before it was discovered. Unfortunately, the focus of formal verification for hardware has mainly been on verifying functional correctness. When out-of-order and superscalar processors became widely adopted, the trend of using the ISA as a correctness criterion persisted, giving birth to techniques such as the *Burch-Dill flushing technique* [37, 88, 185, 199, 217] which has been used as a general correctness criterion for leading semiconductor chip companies. Software-based formal verification, on the other hand, only reasons at the instruction level, without details about the hardware which is fundamental to these attacks. Consequently, many existing techniques that reason about hardware only or software only, have become inadequate for security.

A transient execution vulnerability typically follows four stages as shown in Figure 2.3. In the first stage of the attack (S1), the attacker prepares the side channel similar to the Prime+Probe attack and executes code gadgets to trick the microprocessor into speculating in a later stage. In the second stage (S2), the attacker calls the victim function, which speculatively executes code that wouldn't have otherwise been executed under non-speculative execution. In the

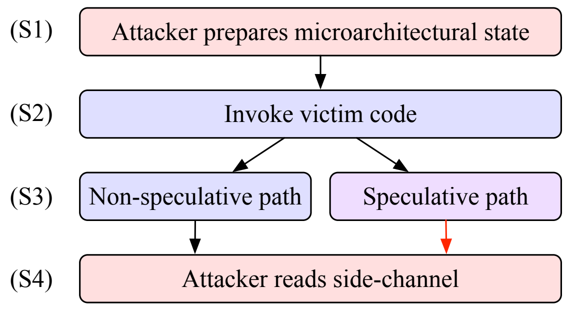


Figure 2.3: Stages of a speculative attack.

third stage (S3), the victim's program accesses secret information and propagates it to a side channel as indicated by the red arrow. At that point, all the attacker needs to do is initiate the probe stage (S4) to infer information about the secret. While this may seem similar to an attack using the Prime+Probe attack, we will later see in Chapter §4, the difference in the extent to which secrets can be revealed.

### 2.1.3 Attacker Model

For a majority of hardware-based attacks [116, 133], the attacker model is assumed to be executing on the same physical core so that it shares the same resources such as the L1D cache, branch prediction buffers, etc. However, this is not always the case, and the attacker has been shown to effectively observe leakage even across cores [134]. The attacker is also typically assumed to have access to user space and can execute for an unlimited duration, use system calls, and execute instructions to accurately time programs (e.g., using an instruction such as `rdtsc` in x86 architectures).

## 2.2 Trusted Execution Environments

Trusted execution environments first appeared in the mobile device industry over a decade ago to prevent malicious users from exploiting their networks and to isolate applications from confidential user data. Over time, they have become popular for the secure computation of unencrypted data in both commercial devices and cloud computing. In this section, we mention the existing formal security efforts for TEEs, why we should embrace their use, and motivate how the use of formalisms and formal guarantees can alleviate the limitations that exist. Lastly, we briefly explain how even with such a powerful hardware mechanism for enforcing memory isolation, hardware attacks began to cast doubt on TEEs in recent years.

### 2.2.1 Formal Guarantees for TEEs

As TEE adoption increased, users became more security-aware about the security of their platforms, and efforts to verify properties critical to correctness and safety such as linearizability [130] and confidentiality [198, 204] began to materialize. However, to date, there is still limited effort in proving the formal properties of these platforms. Despite this shortcoming, TEEs are still a promising tool as we explain below, and we believe that the community should embrace both the usage and automated formal verification of TEEs to further enforce security.

### 2.2.2 Why Use a TEE?

Besides the low overhead in performance cost compared to other secure computation alternatives, TEEs have a number of additional benefits which we detail below.

**Hardware-Based Memory Isolation**. TEEs provide strong memory isolation by using *hardware mechanisms* to isolate memory accesses from a given process to their own *domain*. Typically, these hardware mechanisms rely on non-traditional hardware components. For example, Intel's SGX and TDX [101] contain reserved memory regions inside the DRAM referred to as the Processor Reserved Memory (RPM) and the Secure Arbitration Mode (SEAM) memory range respectively. These memory ranges are isolated based on the logic of the memory management unit or memory controller, which is a hardware component within the processor. Similarly, in Keystone, the set of protected memory ranges is specified by the physical memory protection (PMP) registers and memory access is enforced through the memory controller. This strengthens traditional process isolation enforced by the operating system because in order for an attacker to break isolation, they would now need to also bypass the access checks of the memory controller in hardware.

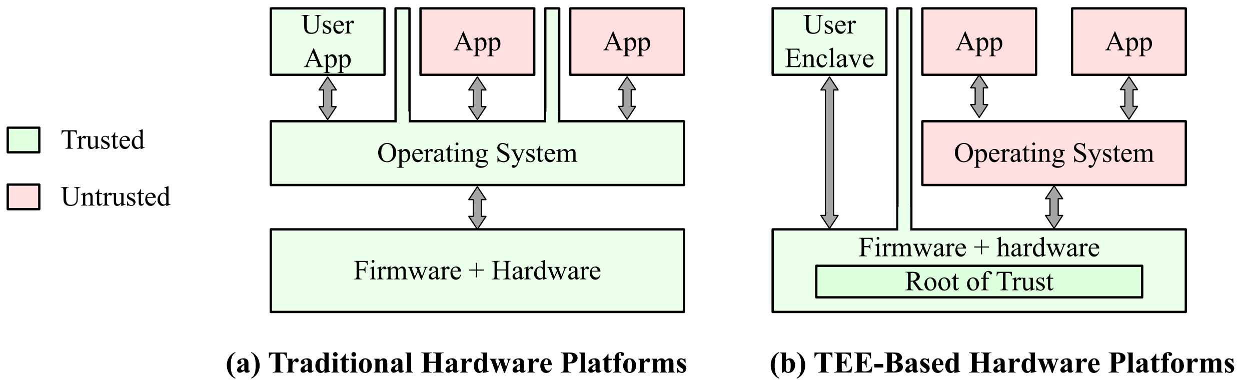


Figure 2.4: Tradition hardware platforms vs. TEE-based hardware platform isolation boundaries represented by the vertical bars.

**Small trusted computing base**. The other upside of a TEE is that the *trusted computing base* (TCB) of the TEE (Figure 2.4 (b)) – the part of the system that is critical for security – is typically smaller than the TCB of traditional computing platforms (Figure 2.4 (a)), due to several factors. First, a TEE-based hardware platform provides a handful of hardware-assisted functions to manage enclave programs (e.g., launch, destroy, pause, etc) as shown in Figure 2.5(b). In contrast to traditional platforms, this is much smaller than the hundred(s) of system calls provided by an operating system for privileged operations as shown in Figure 2.5(a). Second, TEEs generally expose these functions through a trusted *security monitor* as part of the firmware, as the only way to manage the protected enclave programs. For instance, in Keystone [127], the security monitor executes in a privileged *machine mode* and only exposes these functions to manage enclaves at the user level. Thus, the operating system, which often contains tens of millions of lines of code, does not directly interact with trusted enclave programs. Consequently, this separates the operating system from the TCB and greatly reduces the part of the system that needs to be formally verified as indicated by the green components in Figure 2.4(b). This reduction of complexity makes verification of security properties such as non-interference possible on hardware platforms.

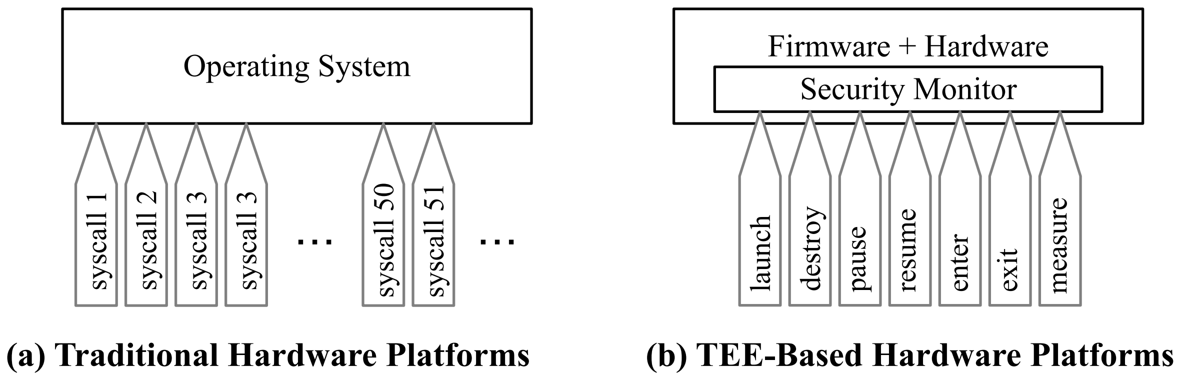


Figure 2.5: The number of operations in the traditional hardware platform is comparatively more than that of the TEE-based hardware platforms.

**Secure remote execution with attestation**. Beyond proving memory isolation for enclave programs on the platform, TEEs are often implemented with *remote attestation* to ensure the integrity of a remote enclave program. The function provides an attestation report to the user of the remote platform and tells the user that the program executing in the cloud is indeed the program they uploaded. The attestation report is a *chain of trust* containing a tamper-proof key from the *silicon root of trust*, a component integrated into the hardware by the platform vendor.

### 2.2.3 Limitations Due to a Lack of Formalisms

Nevertheless, existing designs of modern TEEs come with a number of limitations that warrant consideration for future designs and verification efforts.

**Restrictive programming model**. While having a simple programming model is beneficial because a reduced number of operations is needed to be reasoned about, it imposes constraints on the behavior of enclave programs and their optimizations. An example of this is many TEE designs also do not allow enclave programs to access many of the functions provided by the OS (e.g., system calls) while executing. As a result, the only alternative the enclave program [159, 210] can take is to perform costly context switches.

**Strict memory sharing model**. Another downside is that TEE designs often impose strict memory isolation at the physical memory level [10, 53, 112, 141]. This results in unnecessary overhead from cold-start latency, which is undesirable for programs that frequently read from the same libraries or data models. Work on extending enclaves to support memory sharing exists [63, 131, 231], but they often lack formal analysis. Without a formal model, it is difficult to reason about the claimed security guarantees, and introducing memory sharing on top of existing vulnerabilities without formal analysis only adds to the unreliability of such claims.

**Missing a common platform and threat model**. Lastly, TEEs such as Intel SGX and TDX, AMD SEV, Arm TrustZone, Keystone, and Sanctum lack a consistent threat or platform model. This was historically influenced by the industry, due to a difference in goals that address each vendor's business needs and the customers they target for commercial products. However, this makes it almost impossible to design formal verification methodologies that aren't one-off approaches for each platform. Having a formal model that TEE designers can use as a specification would provide security ”for free”.

These issues are addressed in Chapter §6 by extending an existing formal abstract TEE model called the *Trusted Abstract Platform* [204] with memory sharing. This allows us to provide support for all TEEs that implement this abstraction and more importantly, provide a way to escape the traditional approach of building TEEs without combined design and verification.

### 2.2.4 Hardware Attacks on Trusted Execution Environments

Unfortunately, transient execution attacks even affect memory isolation in TEEs. In 2019, the Foreshadow [215] attack was discovered in Intel's SGX enclaves, which exploits a flaw in the exception handling logic when accessing enclave memory, resulting in secret information being leaked through the caches similar to the Spectre attack. This was later extended [224] to break the virtual memory abstraction, resulting in secrets being leaked across user program, virtual machine, and operating system domains.

In addition, many TEEs do not offer formal assurances of secure computation or protection against transient execution attacks. Ideally, platforms such as TEEs, that claim to be secure, should have a workflow that allows checking security properties an efficient task.

## 2.3 Security for Hardware Platforms

While there exist many techniques to prove functional correctness on these platforms, literature on proving secure information flow is sparse. Two key security properties from the CIA triad that a platform should satisfy are *confidentiality* and *integrity*[*]. These properties form the basis of secure information flow.

[*]We forgo *availability* because the applications we consider (i.e., the hardware platforms) typically do not aim to provide this guarantee, which is consistent with existing work.

**Confidentiality**. When a user executes their program on a hardware platform with sensitive information, they hope that no other entity is able to access that information. Enforcing such a policy is what is referred to as confidentiality.

**Integrity**. While the protection of data is important, we also need to know that the result is trustworthy or unmodified by an untrusted entity. Integrity says that a platform will always execute in an expected way for a given user input sequence. For example, if a user asks their program to return the balance of their bank account, they would hope that the result is the actual balance of their bank account and not that of another user.

More precisely, these two properties are *hyperproperties* [51] that require reasoning about multiple copies of a platform. We will see in later chapters why providing guarantees about hyperproperties on hardware platforms is challenging. In the next chapter, we present our approach to providing formal guarantees of secure information flow for speculative microprocessors and TEEs presented in this chapter.

# Chapter 3

# Secure Information Flow with Formal Methods

In the previous chapter, we explained the necessity for proving hyperproperties on hardware platforms. While there are numerous model checking approaches for hyperproperties, existing approaches and hyperproperties are not tailored for the class of hardware attacks. Our work focuses on extending these approaches to precisely and efficiently prove security against these hardware attacks. In this chapter, we motivate why this class of attacks requires formal approaches and provide the necessary formal definitions that the subsequent chapters build upon.

In the context of our work, formal methods enable one to definitively check the satisfaction of a property $\phi$ about a formal model $M$ of a platform against a formal adversary model $\mathcal{A}$ (Figure 3.1). Due to the scale of our models, our approaches focus on the use of automated techniques such as bounded and inductive model checking [24] and interpolation [143] with an underlying Satisfiability Modulo Theory (SMT) solver-based [15, 17, 56] backend.

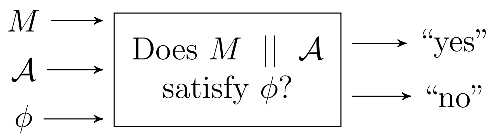


Figure 3.1: Formal verification allows one to prove that a system $M$ composed with adversary $\mathcal{A}$ either satisfies $\phi$ or not.

This approach exhaustively explores all reachable states of a model and can be brittle when the models are complex, often resulting in *state explosion* [50]. Alternative approaches that check information flow without sacrificing scalability such as static analysis and dynamic analysis on the other hand, only provide approximations to the satisfaction of $\phi$ and are prone to returning false positives for complex systems.

However, these approximations are inadequate for the class of transient execution attacks where a lack of precise guarantees led to a multi-decade-old flaw in modern-day processor designs. Following, we highlight some of the inadequacies of these approaches and discuss why there is a strong need for formal methods for this class of attacks.

**Static analysis**. Static analysis [155] and abstract interpretation [54] are often used to approximate the security of a system without execution or simulation. However, the downside is that these approaches lack precision. More often than not, the execution of a system at a given state is conditional on its state and inputs, thus without some form of simulating execution, it's not possible to evaluate the condition and thus determine if a state is reachable. To preserve soundness, these methods often make coarse approximations at each step of the analysis, often resulting in false positives (i.e., a violation of $\phi$ when there is no violation on the actual implementation) and overapproximating the set of reachable states. As a result, this class of analyses is not well suited for transient execution attacks where the vulnerability relies on the precise semantics of software and hardware.

**Dynamic analysis**. On the other hand, approaches that execute or simulate a system to determine the bad reachable states avoid false positives [12, 28, 110]. However, they come at the cost of losing soundness because it's often infeasible to exhaustively execute a system under all inputs to explore all the possible execution paths. For proving the security of a system, having soundness for non-interference properties is key to preventing these bugs, thus dynamic analysis is ill-suited for these classes of attacks also.

**Symbolic execution**. Traditionally, using symbolic execution or formal methods to provide security guarantees to a system consists of three major aspects: modeling, specification, and verification. In our context, modeling requires defining a formal model $M$ of the system implementation or design to precisely define the semantics of the system. Moreover, in our context, a recurring theme will be the importance of specifying an adversary model $\mathcal{A}$ that executes interleaved with $M$. Specification involves formalizing and specifying a property $\phi$ that we would like to prove on the composition of our models, denoted $M \mid\mid A$. Finally, verifying that $M \mid\mid A$ satisfies $\phi$ is often a challenge on its own. For example, overly complicated properties with many quantifiers can result in poor verification performance even with state-of-the-art SMT solvers. Thus, careful and delicate tuning of these models (e.g., by using separation logic [167, 236], compositional proof systems, removing quantifiers [125, 176], proving a small model property [168], using a smaller set of theories and etc) is often necessary to successfully verify a system. Thus, this brings us to the question of how we can model our system $M$. In the following section, we describe the standard formalisms of transition systems and their properties and build upon this definition in subsequent chapters to prove the secure information flow of hardware platforms.

## 3.1 Secure Information Flow

Throughout this thesis, the central class of security properties that we are concerned with are information flow-theoretic properties such as non-interference in the form of confidentiality and integrity. In this section, we introduce the standard non-interference property and variations of non-interference that Chapters §4, §5 and §6 extend. We begin by defining the modeling formalism of transition systems over which our security properties can be defined.

**Labelled Transition System.** Let $AP$ be a set of atomic propositions, then a labeled transition system – or more specifically, a Kripke structure – is of the form $M = \langle Q, I, L, \delta \rangle$ where $Q$ is the set of states of the system, $I \subset Q$ is the set of initial states of the system, $L : Q \rightarrow 2^{AP}$ is the labeling function and $\delta : Q \times L \times Q$ is a transition relation that describes the valid transitions of the system. Alternatively, $Q$ can be viewed as the set of valuations of state variables $V = \{v_1, v_2, ..., v_{|V|}\}$ and thus we sometimes write $M = \langle V, I, \delta \rangle$. The relation between $Q$ and $V$ can be described by defining the domain set $\mathbb{D}(v)$ of values that the variable $v \in V$ can be assigned to. Then the set of states can be characterized by $V$ in the following way: $Q \doteq \mathbb{D}(v_1) \times \mathbb{D}(v_2) \times ... \times \mathbb{D}(v_{|V|})$. Under this formulation, we will write $q.v$ to be the valuation of variable $v \in V$ in state $q \in Q$. Lastly, the execution of $M$ emits a trace $\pi$ which is a sequence of states $\pi = \pi^{(0)}\pi^{(1)}\pi^{(2)}...\pi^{(n)}$ where $\forall i \in [n].\ \ \pi^{(i)} \in Q$ and $[n] = \{1, ..., n\}$. We write $Tr(M)$ to be the set of all traces of $M$. We note that these traces can also be infinite.

**Security labels.** In order to discuss whether information flows from a secret value to a publicly visible state variable, we define a security lattice $\mathcal{L} = \{l_1, ..., l_n\}$ associated with an ordering relation $\sqsubseteq$ that determines valid information flows.

In this thesis, we consider concurrent systems consisting of two components: an untrusted (or public/low-security) component and a trusted (secret/high-security) component. Thus a standard security lattice [119] that can describe these components is the one containing only high and low labels $\mathcal{L}_2 = \{\mathtt{H}, \mathtt{L}\}$ with the ordering relation respecting $\mathtt{L} \sqsubseteq \mathtt{H}$. This relation states that information can only flow from low to high at each step of the system. Each of these labels can be used to define the sensitivity of the information contained in a given state variable of the system. To describe the sensitivity of a given variable, we write $v \vdash l$ to mean that variable $v$ has sensitivity $l$. For example, in a microprocessor, the program counter is typically assumed to be publicly visible under the constant-time programming discipline [43] and would be labeled low under this assumption.

**Self-Composition for Security.** A typical requirement of the confidentiality property is that the low component must not be able to distinguish between the secret states of the high component. This can be formalized by defining a notion of *indistinguishability*, which is captured using the operator $\approx_{\mathtt{L}}$. $\approx_{\mathtt{L}}$ denotes the equivalence of all variables labeled low between two states $q$ and $q'$: $q \approx_{\mathtt{L}} q' \doteq \forall v \in V.\ \ v \vdash \mathtt{L} \Rightarrow q.v = q'.v$. This then allows us to express whether an adversary can tell a difference between the two states. This notion of equivalence is used to compare the states between two copies of a system, each containing different secrets. Similar to system states, low-equivalence can be extended to traces in the obvious way. Two traces $\pi_1$ and $\pi_2$ are low-equivalent, written $\pi_1 \approx_{\mathrm{L}} \pi_2$, if each $i$-th state is low-equivalent: $\pi_1 \approx_{\mathrm{L}} \pi_2 \doteq \forall i \in [n].\ \ \pi_1^{(i)} \approx_{\mathrm{L}} \pi_2^{(i)}$. The composition of multiple copies (or *instances*) of the platform is also known as a *self-composition* over the platform model and allows one to describe *hyperproperties* [51], which is a class of properties that can only be fully evaluated on a single trace. Using self-composition, we can now formally define non-interference and observational determinism.

$$\forall \pi_1, \pi_2 \in Tr(M), \pi_1^{(0)} \approx_{\mathrm{L}} \pi_2^{(0)} \Rightarrow \pi_1^{(n)} \approx_{\mathrm{L}} \pi_2^{(n)} \tag{3.1}$$

**Non-interference**. Intuitively, non-interference [77] says that if a high user is working on their secret data, it should not interfere with the low user who can observe the low state variables. In other words, the low user should operate independently of the secrets. This is formalized as Eq 3.1. One standard approach to verify this property is illustrated in Figure 3.2, where two copies of the platform emit two traces $\pi_1$ and $\pi_2$, with the assumption that they start off in low-equivalent states $\pi_1^{(0)}$ and $\pi_2^{(0)}$, and the proof obligation is to show that their final states $\pi_1^{(n)}$ and $\pi_2^{(n)}$ are also low-equivalent. We note that while non-interference is also used as an umbrella term for a class of information flow properties, we specifically refer to this formulation when we mention non-interference in this thesis.

$$\begin{tikzcd}
\pi_1^{(0)} \arrow[r] \arrow[d, phantom, "\wr\wr_{\mathrm{L}}"] & \pi_1^{(n)} \arrow[d, phantom, "\wr\wr_{\mathrm{L}}"] \\
\pi_2^{(0)} \arrow[r] & \pi_2^{(n)}
\end{tikzcd}$$

Figure 3.2: Illustration of the assumptions and proof obligations for the non-interference property depicted by the execution of two traces of the system. The initial assumption of low equivalence is shown in blue ($\approx_{\mathrm{L}}$) and the proof obligation is shown in green ($\approx_{\mathrm{L}}$).

However, this classical definition of non-interference introduced by Goguen and Mesaguer was originally applied to deterministic programs and only requires that the final states of the traces be low-equivalent. Thus, adversaries that can execute interleaved with the program and violate low-equivalence within intermediate states of the trace cannot be captured.

$$\forall \pi_1, \pi_2 \in Tr(M), \pi_1^{(0)} \approx_{\mathrm{L}} \pi_2^{(0)} \Rightarrow \pi_1 \approx_{\mathrm{L}} \pi_2 \tag{3.2}$$

**Observational determinism**. To deal with this inadequacy, the definition of non-interference above was generalized by McLean [142] to traces and then later to concurrent programs by

Zdancewic and Myers [233] as *observational determinism*. Observational determinism formalized as Eq 3.2. As depicted in Figure 3.3, in contrast to the original non-interference property (Eq 3.1) there is an obligation to prove low-equivalence at each $i$-th state of the traces.

$$\begin{tikzcd}
\pi_1^{(0)} \arrow[r] \arrow[d, phantom, "\approx_{\mathrm{L}}"] & \cdots & \pi_1^{(i)} \arrow[r] \arrow[d, phantom, "\approx_{\mathrm{L}}"] & \pi_1^{(i+1)} \arrow[r] \arrow[d, phantom, "\approx_{\mathrm{L}}"] & \pi_1^{(i+2)} \arrow[r] \arrow[d, phantom, "\approx_{\mathrm{L}}"] & \cdots & \pi_1^{(n)} \arrow[d, phantom, "\approx_{\mathrm{L}}"] \\
\pi_2^{(0)} \arrow[r] & \cdots & \pi_2^{(i)} \arrow[r] & \pi_2^{(i+1)} \arrow[r] & \pi_2^{(i+2)} \arrow[r] & \cdots & \pi_2^{(n)}
\end{tikzcd}$$

Figure 3.3: Illustration of the assumptions and proof obligations for observational determinism. Low equivalence is checked at each intermediate state ($\approx_{\mathrm{L}}$) and assumed in the initial state ($\approx_{\mathrm{L}}$).

Having defined the security properties of interest, we now turn to describe how our approaches use formal methods to provide strong security guarantees to secure hardware platforms.

## 3.2 Formal Methods for Heterogeneous Systems

Automated formal verification has found great success in proving the functional correctness of hardware circuit designs [37, 59] and software [114]. However, applications have focused mostly on verifying only hardware or only software. This led verification tools to be designed for sequential (and concurrent) software verification [16, 86, 129] or circuit verification [107, 120, 138, 150]. As a result, modeling heterogeneous systems is cumbersome [204], often requiring the user to individually verify the hardware and software (possibly on two different verification tools). What is more challenging, is finding the right level of abstraction to interface the software with the hardware logic for assume-guarantee reasoning.

**Desired features of verification tools for heterogeneous systems**. Over the years, the design of verification tools and languages have naturally been driven by application domains. Thus, many tools have tended towards a particular paradigm, rarely integrating different modes of computation. On the other hand, heterogeneous systems require a combination of modes that is lacking in many existing verification tools. However, tools are not only lacking in the functional aspect, but also in fidelity and robustness to changes due to a lack of automation. Below, are some of the major traits that one would desire to have in their verification tool for modeling and verifying heterogeneous systems.

- **Diversity of Specification**. Many security properties [119] are complex hyperproperties [51], which require reasoning about multiple traces of a platform model. It is important these are easy to write and maintain.

- **Compositionality.** Heterogeneous systems consist of multiple components each with varying designs, hence it is important that the designs can be composed. This enables the automation of modeling combinations of designs.
- **Diversity in Modeling.** Each component of a system may also be more easily expressed in a specific style. For example, hardware may be more naturally expressed using an axiomatic model [75, 135, 153], whereas software is often imperative. Thus, a tool that is capable of various modes of modeling would allow the seamless integration of different components using a single tool.
- **Range of Verification Techniques**. Verifying properties over traces (security properties) requires bounded or inductive model checking, while functional verification is more procedural. Thus, a tool that allows for a range of verification techniques would allow the entire verification process to be done with one tool.

To that end, we describe UCLID5 [169, 193], a verification tool designed for modeling and verifying heterogeneous systems. UCLID5 addresses the needs above and is the main formal methods tool used throughout the thesis for modeling, specification, and verification.

## 3.3 Formal Modeling, Specification, and Verification with Uclid5

In this section, we demonstrate how we use UCLID5 to verify observational determinism. We provide an overview of the relevant features in UCLID5 for our work, showcase these features by verifying an example C program, and then conclude with an explanation of how the internals of UCLID5 work to verify the property.

### 3.3.1 Overview

UCLID5 implements a number of key features to enable a high degree of automation and efficient verification for heterogeneous systems. These features include the ability to naturally model transition systems, model sequential code, combine these two modeling approaches, use a mix of axiomatic and operational modeling, and verify specifications using a range of techniques. UCLID5 also incorporates algorithmic program synthesis based on inductive learning as a core feature [192].

**Concurrent modeling with modules and instances**. A module in UCLID5 models a transition system, in which one can define concurrent updates. An instance is an extended functionality of modules that allows one to reason about multiple copies of modules, which automates the modeling required to prove hyperproperties which are multi-trace properties.

**Sequential modeling with procedures**. UCLID5 also allows the definition of procedures for modeling sequential code. Procedures are defined using common programming constructs and model control flow, assignments, etc. Procedures in UCLID5 are similar to procedures in software verification tools such as Boogie [16]; they include the standard *requires*, *ensures*, and *modifies* statements for specifying preconditions, postconditions, and a list of variables that are modified in the procedure.

**Combining concurrent and sequential modeling**. Procedures can also be called in concurrent updates of the transition systems. This enables UCLID5 to easily combine both sequential and concurrent modeling, necessary for modeling heterogeneous systems that consist of concurrent hardware and sequential software models.

**Axiomatic modeling and operational modeling**. The last notable feature of UCLID5 used in this thesis includes the ability to mix both *operational* and *axiomatic* styles of modeling. An operational style of modeling is similar to traditional procedural programming languages, whereas axiomatic encodings allow one to specify first-order logic properties that hold at either a local level (e.g., within a procedure at a given program point) or at a global level (as an axiom). This allows us to easily define abstract (uninterpreted) functions and add assumptions of the platform.

**Multi-modes of verification**. Finally, to check properties or invariants over the platform model, UCLID5 allows a number of verification modes such as bounded model checking, inductive model checking, and procedural verification. On top of verification, UCLID5 also provides native support for program synthesis through SyGuS [7].

Following, we provide an example to illustrate how these features in UCLID5 can be used to verify memory isolation for a C program. We use a common formal modeling paradigm [204] that models hardware platforms as a transition system. Thus, this will also serve as a precursor to the programming models introduced in Chapters §4, §5 and §6, and how we model hardware platforms.

### 3.3.2 Example Victim C Program

```
// N = 10
// &tmp = 0x8000, &a = 0x8008
void victim_prog(int x) {
  if (x < N) {
    tmp = a[x];
  }
}
```

Example 3.4: Victim program in C.

Consider the task of verifying that a platform isolates a victim's and adversary's memory regions. We assume that the adversary has access to a victim-defined program as shown in Example 3.4 named `victim_prog`. In the victim's memory region, we have a variable `tmp` which is of byte length, and `a` which is a byte array of length `N = 10`. For simplicity, let's also assume that the address of `tmp` is `0x8000` and that the address of `a` is `0x8008`. Consequently, if these variables are the only variables accessible to our victim, then the victim's memory region can be defined by `0x8000 - 0x800A`, assuming `tmp` and `a[i]` are byte-sized. Thus, if the victim's memory region is isolated from the adversary, then one would expect that no matter how the attacker calls `victim_prog` with their choice of `x`, the adversary should not observe differences in their memory space. This property can be proven using the observational determinism property defined as Eq 3.2. Now, we show how this can be encoded and proven in UCLID5.

### 3.3.3 Modeling The Victim C Program

In order to verify a property such as observational determinism on the victim C program, one needs to first formalize a model of the platform on which the program executes, a model of the victim program, and the attacker model. We demonstrate this in UCLID5.

**Modeling the platform**. Example 3.5 illustrates one way that a platform can be modeled in UCLID5 as a transition system $M = \langle V, I, \delta \rangle$, where $V := \{\texttt{pc}, \texttt{mem}, \texttt{x}, \texttt{addr}, \texttt{obs}\}$, $I := (\texttt{tmp\_ptr} == \texttt{0x8000bv64}) \wedge (\texttt{a\_ptr} == \texttt{0x8000bv64}) \wedge (\texttt{N} == \texttt{0x10bv64} \wedge (\texttt{obs} == \texttt{0bv64})$. $\delta$ is defined by either an execution of `victim_prog` if `pc` == `0x8000bv64` or `adversary_obs`. UCLID5 allows the definition of state variables using the `var` keyword with an associated identifier for the variable and its type. Similarly, `const` can be used to define the usual constants. For instance, the program counter variable is defined as a variable with the identifier `pc` with the 64 bit-vector (`bv64`) type. To model memory, we use a byte-accessible array indexed by 64-bit addresses, declared as `mem` with the array type `[bv64]bv8`. Finally, we use constants to declare the address of array `a` as `a_ptr`, the address of the temporary variable `tmp` as `tmp_ptr`, and the value `N` is declared using the same name.

```
// platform.ucl
module M {
  // adversary inputs and
   observations
  input x, a:         bv64;
  var obs:            bv8;

  // platform state variables
  var pc:             bv64;
  var mem:            [bv64]bv8;
  const a_ptr:        bv64;
  const tmp_ptr, N:   bv64;

  init {
    assume (tmp_ptr == 0x8000bv64);
    assume (a_ptr == 0x8008bv64);
    assume (N == 10bv64);
    obs = 0bv8;
  }

  next {
    havoc pc;
    if (pc == 0x8000bv64) {
      call victim_prog();
    } else {
      call adversary_obs();
    }
  }
}
```

Example 3.5: Platform model in UCLID5.

```
// platform.ucl
module M {
  ...
  procedure victim_prog() {
    if (x < N) {
      mem[tmp_ptr] = mem[a_ptr + x];
    }
  }
  ...
}
```

Example 3.6: Continued platform model in UCLID5 defining the victim program.

**Modeling the victim program**. The program victim_prog (e.g., from Example 3.4) is modeled as a procedure of the same name in UCLID5 at line 4 of Figure 3.6. We omit the explanation of the UCLID5 semantics of the program and refer to reader to the paper [169]. The takeaway is that the procedure defines sequential semantics for the program.

```
// platform.ucl
module M {
  ...
  procedure adversary_obs() {
    obs = mem[a];
  }
}
```

Example 3.7: Continued platform model in UCLID5 defining the adversary observation.

**Modeling the attacker**. We model a passive attacker with an observation function as described in Chapter 1.2. To model this, we define a procedure `adversary_obs` which updates the adversary's observation `obs` variable with the value it reads from memory. This is defined at line 4 of Figure 3.7. Inputs to the platform include `adv_x` and `adv_addr`, which correspond to the attacker-controlled argument `x` of `victim_function` and the memory address that the attacker reads from to update its observations. Inputs to a module are read-only variables that can only be changed by an outer module that instantiates it (i.e., the main module).

**Modeling the execution model**. The platform's transition $\delta$ is then described on lines 29-36. At line 25, the `pc` is *havoced* to unconstrain the value of `pc` (i.e., it becomes a non-deterministic value). This allows the platform to execute either the victim program or the adversary at each step.

## 3.3.4 Specifying Observational Determinism

**Instantiating platform traces**. Using the platform model defined in Example 3.5, one can specify non-interference to check whether or not the adversary's memory regions are disjoint from the user's. This is encoded in Example 3.8. In this example, two instances `t1` and `t2` of the platform are declared at lines 7 and 8, which corresponds to the two traces $\pi_1$ and $\pi_2$ of the non-interference property in Eq 3.1. At each step of the main module, both of the instances `p1` and `p2` synchronously take a transition defined by the `next` statement in Example 3.5.

**Specifying the property**. The property is then defined as `od` at line 16, which asserts that the observable states of the two traces `t1` and `t2` are equal.

```
 1 // main.ucl
 2 module main {
 3   // (low) attacker input
 4   var adv_x: bv64;
 5   var adv_a: bv64;
 6   // traces
 7   instance t1: M(x: (adv_x), a: (adv_a));
 8   instance t2: M(x: (adv_x), a: (adv_a));
 9   // programs execute in lockstep
10   axiom t1.pc == t2.pc;
11   // t1 and t2 public memories are equal
12   axiom forall (a: bv64) ::
13           (a < 0x8000bv64 ==>
14             t1.mem[a] == t2.mem[a]);
15   // observational determinism
16   invariant od: t1.obs == t2.obs;
17
18   next {
19     next(t1); next(t2);
20   }
21
22   control {
23     v = bmc(5);
24     check;
25     print_results;
26     v.print_cex();
27   }
28 }
```

Example 3.8: Main proof module in UCLID5.

**Control Block**. Finally, we define the verification decision procedure used to check our property inside a `control` block. In the example, we use *bounded model checking* with a horizon of 5 steps (instructed using the command `bmc(5)`) to check the invariant `od`. This means that the transition system will unroll for 5 steps and for each of the 6 states, UCLID5 asserts that the invariant holds. This is done by generating verification conditions for each assertion which is then translated into SMT-LIB queries and run on an SMT solver. We explain the UCLID5 workflow further in section §3.3.7. In this particular example, it checks for the equality of the observations in the two traces $t1, t2$ at each state: `t1.obs == t2.obs`. The control block can also be used to specify other modes of verification such as procedural verification and inductive model checking which we use heavily for the proofs in Chapter §6. The `check` command instructs UCLID5 to execute the verification commands and `print_results` instructs UCLID5 to print whether each assertion (including invariants and procedure post-conditions) within the purview of the main module successfully verified or failed to verify. Lastly, `v.print_cex()` instructs UCLID5 to print counter-examples to the verification queries for any assertion that fails. For bounded or inductive model checking, the counter-example returned is in the form of a trace. For procedural verification, a counter-example returned is a pair of pre- and post-states that violate a post-condition.

**Hyperinvariants and hyperaxioms**. UCLID5 also supports a more compact way to specify hyperproperties by using `hyperinvariant` and `hyperaxiom` that avoids having to explicitly declare all of the traces.

This is illustrated in Example 3.9 which extends the module definition of `M` from Example 3.5. Instead of instantiating two instances of `M`, one can simply specify observational determinism as a (hyper)property within the module `M` as shown on line 5. The `[2]` indicates that the property contains two traces and the suffix operator `.i` (where `i` is an integer) used with variable `v`, written `v.i`, indicates variable `v` corresponding to the $i$-th trace. In addition to the axioms in Example 3.8, we must also add axioms that the attacker input must be the same between the two traces, written as `same_a` and `same_x`. In Example 3.8, these assumptions were implicit in the model because we passed the same variables `adv_a` and `adv_x` into the module definitions.

```
 1 // continued platform.ucl
 2 module M {
 3   ...
 4   // same adversary inputs
 5   hyperaxiom[2] same_adv_a: a.1 == a.2;
 6   hyperaxiom[2] same_adv_x: x.1 == x.2;
 7   // assume the programs execute in
       lockstep
 8   hyperaxiom[2] same_pc:    pc.1 == pc.2;
 9   // t1 and t2 public memories are equal
10   hyperaxiom[2] same_public_mem:
11     forall (a: bv64) ::
12       (a < 0x8000bv64 ==>
13         mem.1[a] == mem.2[a]);
14   // observational determinism
15   hyperinvariant[2] od: obs.1 == obs.2;
16   ...
17 }
```

Example 3.9: Observational determinism using `hyperinvariant` and `hyperaxiom` in UCLID5.

## 3.3.5 Verifying Observational Determinism

Executing UCLID5 then converts each assertion in the modules into an SMT query through a series of compilation passes as described in §3.3.7, where each assertion corresponds to a property or invariant such as `od`. If it holds, UCLID5 states that the property passes, otherwise, a counter-example in the form of a trace is returned containing a valuation of the variables `adv_x`, `adv_a`, `t1.obs`, `t2.obs`, `t1.pc`, `t2.pc`, etc, at each state. We note that, as one would expect, running UCLID5 on Example 3.8 would return a counter-example to `od` because there are no constraints on the address `adv_a` that the attacker can use to read from memory. However, if we have the assumption that the attacker can only access public memory, e.g., we add the module-level axiom `axiom (adv_a < 0x8000bv64)`, then UCLID5 successfully verifies the property `od`.

## 3.3.6 Program Synthesis

While this thesis does not employ program synthesis for its endeavors, UCLID5 supports the synthesis of proof artifacts [152] through the use of SyGuS [7] solvers.

An example of the synthesis feature is illustrated in Example 3.10 through the use of *synthesis functions*. Instead of manually trying to determine system invariants, a user can specify the function that they want to synthesize. This is demonstrated by the declaration of `synth_axiom`. Optionally, the user may specify a (context-free) grammar that defines the search space of candidate functions to consider, e.g., `ax_cfg`. Running UCLID5 with the synthesis feature turned on along with these additional declarations yields the result:

```
// continued platform.ucl
module M {
  // ...
  // CFG of function to synthesize
  grammar ax_cfg(tmp: bv64): boolean = {
    (start : boolean) ::= (expr <= expr);
    (expr: bv64) ::= tmp | 0x7FFFbv64;
  }
  // Synthesis function declaration
  synthesis function synth_axiom(tmp :
   bv64): boolean grammar ax_cfg(tmp);
  // Axiom to synthesize
  axiom (synth_axiom(adv_a));
}
```

Example 3.10: Observational determinism using `hyperinvariant` and `hyperaxiom` in UCLID5.

```
define synth_axiom (tmp:  bv64):  boolean = (tmp <= 32767bv64);
```

Replacing `synth_axiom` in Figure 3.10 by `tmp <= 32767bv64` (i.e. `adv_a` < `0x8000`) allows the proof of `od` to successfully verify.

### 3.3.7 UCLID5 Workflow

The check the specified observational determinism property in §3.3.5, UCLID5 goes through a number of transformative passes from the UCLID5 language down to a SMT-LIB query which is solved using an SMT solver such as Z3 [56] or CVC5 [15].

**Front-end**. First, UCLID5 parses the files shown on the left of Figure 3.11 (e.g., `platform.ucl`, `main.ucl`) along with user configurations which specify the underlying backend SMT solver used. During the parsing stage, UCLID5 also checks for errors that can be inferred statically (e.g., syntax errors). From these files, the front-end builds an abstract syntax tree (AST) of the modules defined in the files in the UCLID5 language.

**AST transformative passes**. The AST then goes through a number of type-checking passes and transformative passes. The latter rewrites many of the programmatic constructs such as switch-cases, loops, define statements, procedure calls and etc in a module such that the resulting module is semantically equivalent to the original. Eventually, all of the modules are flattened into one monolithic module.

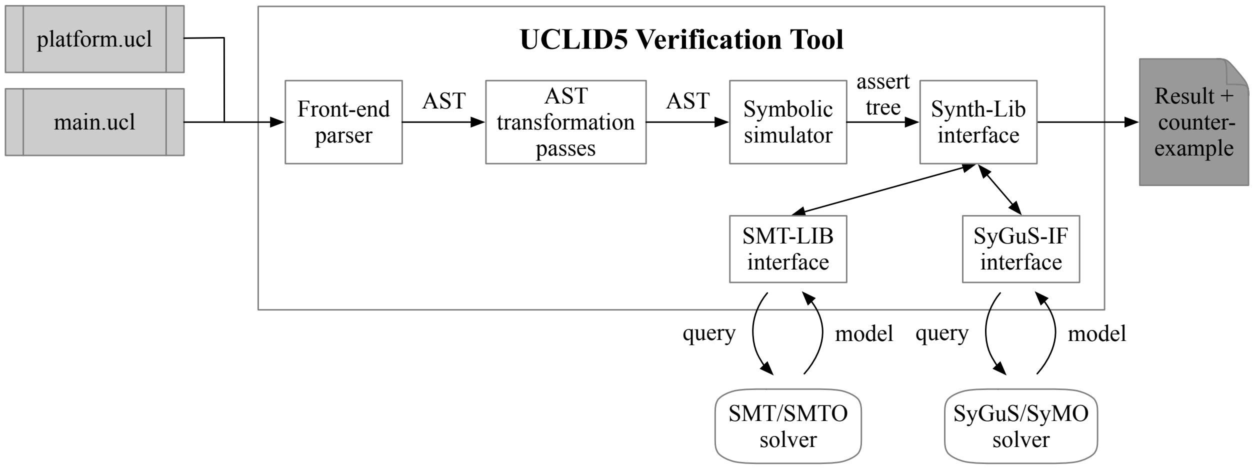


Figure 3.11: UCLID5 workflow for verifying the heterogeneous model from §3.3.1.

**Symbolic simulator**. The symbolic simulator simulates the execution of the given transition system model based on the verification commands in the control block and produces a set of assertions solved by an SMT solver. For instance, Example 3.8 simulates bounded model checking (i.e., `bmc`) for 5 steps of the transition system, where an assertion checking the violation of a property at each step is represented by a separate assertion tree. Within each step, are *verification conditions* translated according to the semantics defined in the next block and the procedures called within the next block.

**Synth-Lib Interface**. UCLID5 supports both verification and synthesis using the Synth-Lib interface. The interface constructs either a verification or synthesis query based on the compiler options and assertions generated by the symbolic simulator. These are then passed to either the SMT-LIB or SyGuS-IF interfaces to create a verification query in the SMT-LIB or SyGuS-IF language. UCLID5 then calls state-of-the-art SMT and SyGuS solvers to solve these queries. The results of the queries and any generated models are then returned to the user.

Worthy of note, UCLID5 also has support for SMT with oracles and synthesis with oracles. UCLID5 leverages SMTO and SyMO solvers [170] to solve SMT and synthesis problems that would otherwise be difficult to solve without black-box oracle functions. This is a promising new direction that enables one to replace a complex or difficult-to-encode function to be used in an SMT-LIB- or SyGuS-IF-based query, with an oracle interface. This oracle interface is then used during the SMT-solving phase to call executable binaries, instead of relying on the complex encoding, to drastically reduce the burden of the SMT solver.

# Chapter 4

# Trace Property-Dependent Observational Determinism

One of the core contributions of this thesis that motivates the need for secure hardware platforms is the idea of *secure speculation*. Secure speculation is a safety property that captures the essence of transient execution attacks such as Spectre [116] and Meltdown [133]. More generally, this property is a specialization of *trace property-dependent observational determinism*, a novel 4-safety property formulation of observational determinism where violations are conditional based on trace properties. These properties are the first to precisely capture the broad class of speculation-based hardware attacks in microprocessor designs that have materialized in recent years. In addition to capturing what it means to be secure against this broad class of hardware attacks, we also present the first methodology to formally verify a program's susceptibility to these Spectre-styled attacks. This approach is further extended in the chapter §5 to effectively capture a broad class of hardware attacks and scale to realistic examples.

## 4.1 Introduction

In 2017, transient execution attacks such as Spectre and Meltdown [116, 133] were discovered and the longstanding presumed security of many optimizations in microprocessors were suddenly called into question. Over the years, these attacks [39, 87, 93, 94, 98, 99, 100, 117, 137, 215] continued to grow at an alarming rate, and have shown that side-channel attacks are more exploitable than previously thought. While side-channel attacks [2, 105, 128, 134, 162, 165, 191] such as Prime+Probe are nothing new, transient execution attacks are interesting because they combine side-channels with microprocessor optimizations in an intricate and obscure way that creates a vulnerability we have not considered. A number of software and hardware mitigations [93, 94, 95, 96, 99, 100, 149, 158, 213] have been proposed for these vulnerabilities, many of which have also been implemented in widely-used software such as the Linux kernel [97, 203], Microsoft Windows [38, 148] and the Microsoft Visual

Studio compilers and associated libraries [149]. However, these mitigations are not provably secure and some (e.g., Spectre mitigations in Microsoft Visual Studio) have even been found to be incomplete [115].

Transient execution vulnerabilities exploit microarchitectural side-channels in modern high-performance superscalar processors. These processors contain microarchitectural optimizations — e.g., branch prediction, data and instruction caching, out-of-order execution, speculative memory address disambiguation, and data forwarding, to name a few — in order to execute programs more efficiently [195, 200]. Many of these optimizations include speculation techniques [73, 132, 139, 230] that operate on data before the microprocessor knows it needs to. This is accomplished by using prediction structures (e.g., a pattern history table for conditional branch prediction) to guess whether a particular execution is likely to occur before its results are available and speculatively executes based on the prediction. If the prediction turns out to be wrong, the microprocessor uses a rollback mechanism to revert the architectural state – which consists of register and memory values – back to their original values before speculative execution started on that instruction, and the execution begins again on the correct execution path. In many cases, it is possible to build predictors that mostly guess correctly and speculation leads to huge performance and power benefits.

The subtle issue underlying this rollback mechanism, which was previously thought to be sufficient in preventing unintended leakage of secret information, is that only the architectural state is restored while the *microarchitectural* state, such as the cache and branch predictor state, is *not*. Transient execution vulnerabilities exploit this by mistraining prediction structures to speculatively execute vulnerable wrong-path instructions and exfiltrate confidential information by examining the residual microarchitectural side-effects of misspeculation.

### 4.1.1 Preventing Transient Execution Attacks

The above leads to two obvious templates for preventing these vulnerabilities: (1) restrict speculation, or (2) prevent leakage of information through microarchitectural side-channels.

**Restricting speculation**. A number of mitigations take the first approach by turning off speculation in a targeted manner [38, 93, 94, 97, 148, 158, 203, 213]. While most of these mitigations were developed through careful manual analysis of known exploitable vulnerabilities, automated tools for Spectre mitigation also take this approach [149]. While most of these mitigations were developed through careful manual analysis of known exploitable vulnerabilities, automated tools for Spectre mitigation also take this approach [149]. Unfortunately, the latter has been found to be incomplete [115] while the former techniques do not come with provable security guarantees. The larger point here is that there is no formal methodology for reasoning about the security of mitigations to transient execution vulnerabilities.

**Preventing leaky side-channels**. The second approach attempts to close the exfiltration side-channel by ensuring that it does not leak information to the attacker at all. For instance, Dynamically Allocated Way Guard (DAWG) [113] closes the cache side-channel by fully isolating memory between protection domains. Despite this, other side channels (e.g., prefetchers, DRAM row buffers, load-store queues, etc) potentially remain exploitable with these solutions and partitioning comes with a significant performance penalty. Here too, it remains unclear whether partitioning a few exfiltration channels is sufficient to prevent all transient execution vulnerabilities.

Thus, current approaches to these vulnerabilities face two major challenges: the lack of provable security and large performance penalties. In this context, it is noteworthy that versions of the Linux Kernel have turned off certain Spectre mitigations by default because performance slowdowns of up to 50% [123, 212] were observed for certain workloads. We believe these high overheads are a result of the inability to reason about the security of the mitigations precisely. If we could systematically and accurately capture security, it will be possible to develop more aggressive mitigations that disable speculation in a very targeted manner, resulting in much lower performance overheads.

The points above highlight the need for formal verification techniques for *secure speculation*. This problem is most closely related to the secure information flow, which has been studied by a rich body of literature [20, 51, 76, 142, 178, 206, 233]. Unfortunately, existing work on secure information flow is not sufficient to precisely capture the class of transient execution vulnerabilities. Specifically, it is important to note that traditional notions of information flow security like non-interference [76] and observational determinism [142, 178, 233] are only satisfied when there is *no information flow* from confidential state to adversary observable state. In the context of Spectre, this would imply no information flow from confidential memory locations to microarchitectural side channels. For most programs of interest, e.g., the Linux kernel and Microsoft Windows operating system, all modern commercial processors *do* leak information about confidential operating system state through microarchitectural side channels like caches, prefetchers, DRAM row buffers, etc. Therefore, traditional formulations of secure information flow are always violated for such programs regardless of whether they are vulnerable to transient execution attacks.

### 4.1.2 Challenges

Evidently, the inadequacy of these traditional properties points to a key challenge in the verification of secure speculation: *formulating the right property*. It is necessary to find a property that precisely captures the *new* leaks introduced by the exploitation of microarchitectural side-channels through speculation. These new leaks stand in contrast to the previously known side-channel leaks which are already captured via traditional notions of secure information flow such as noninterference/observational determinism.

A second important challenge is coming up with a *general system and adversary model* that can be used to reason about the category of transient execution attacks, as opposed to pattern-matching known vulnerabilities. Thirdly, we need a verification methodology that can be used to prove that specific programs satisfy secure speculation.

### 4.1.3 Contributions

In this chapter, we address each of the above challenges. We introduce a formal methodology for reasoning about security against transient execution attacks. Our approach is based on the formulation of a new class of secure information flow security properties called *trace property-dependent observational determinism* (TPOD) [46]. These properties, an extension of observational determinism, are defined with respect to a trace property and intuitively TPOD captures the following notion of security: *does violation of the trace property introduce new counterexamples to observational determinism?*

We use TPOD to reason about the security of software-based Spectre mitigations. For this, we present an assembly intermediate representation (AIR) into which machine code can be lifted and introduce speculative operational semantics for this AIR. We introduce a general adversary that captures transient execution attacks and we define a secure speculation property for this adversary as an instance of TPOD. We verify secure speculation in an automated fashion using bounded model checking and induction in the UCLID5 verification tool [193, 214] on a suite of small but illustrative benchmarks, several of which are from the literature on Spectre mitigations [115].

To summarize, the contributions of this work are the following.

- We introduce a novel methodology for reasoning about the security of microarchitectural speculation mechanisms. Our methodology can prove that a program is secure against transient execution vulnerabilities.
- We introduce a new class of information-flow security properties called trace property-dependent observational determinism. This class of properties allows us to reason about information leaks that occur due to interactions between microarchitectural mechanisms.
- We introduce speculative operational semantics for an assembly intermediate representation, an adversary model for transient execution attacks over this representation, and a secure speculation property. Violations of the property correspond to transient execution vulnerabilities.
- We demonstrate our methodology by automatically proving secure speculation for a suite of illustrative programs.

The rest of this work is organized as follows. Section 4.2 presents an overview of transient execution attacks. Section 4.3 reviews observational determinism and introduces trace property-dependent observational determinism. Section 4.4 describes the assembly intermediate representation and speculative operational semantics for it. The adversary model and the secure speculation property are described in Section 4.5. Sections 4.6 and 4.7 present our verification approach and case studies. Section 4.8 reviews related work and Section 4.9 provides some concluding remarks.

## 4.2 Overview

In this section, we present an overview of transient execution vulnerabilities as exemplified by Spectre and review the verification challenges posed by these vulnerabilities.

### 4.2.1 Introduction to Transient Execution Attacks

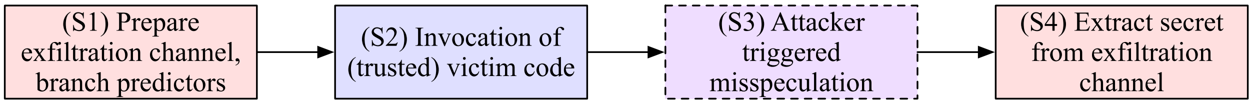

(S1) Prepare exfiltration channel, branch predictors
(S2) Invocation of (trusted) victim code
(S3) Attacker triggered misspeculation
(S4) Extract secret from exfiltration channel

Figure 4.1: Four stages of a speculative execution attack. The execution of untrusted code is shown in red, while the execution of trusted code is in blue. We show the attacker-triggered misspeculation in the trusted code in the violet dotted box.

Transient execution attacks involve two security domains represented as the following components of the platform: an untrusted/low component (the attacker) that interacts with a trusted/high component (the victim) over some communication interface. The attacker exfiltrates confidential information from the victim by exploiting microarchitectural artifacts of misspeculation in high-performance processors. As shown in Figure 4.1, a transient execution attack has four stages. We explain these four stages using the code snippet shown in Figure 4.4(a), which is vulnerable to Spectre variant 1.

**(S1) Prepare**. In the first stage of the attack, the attacker makes the necessary preparations to (a) trigger code that leaks information to a side-channel in the next stage of the attack and (b) *flush* or *prime* the side-channel. One commonly used exfiltration side channel is the data cache. To accomplish the former, the attacker must construct or find a snippet of code that can transmit secret(s) from the victim domain to the cache. One such type of code snippet consists of conditional branch-guarded memory accesses. For this particular type of gadget, the attacker needs to train the branch predictor by repeatedly executing the victim code snippet with carefully chosen input arguments so that the branch predictor learns to always take (or not take) the branch. To complete the preparation, the attacker can use existing methods to flush [228] the cache (e.g., using `clflush` in x86 architectures) so that all entries are invalidated or prime [134] the cache by accessing a specific range of addresses to evict cache entries.

**(S2) Invocation of victim code**. In the second stage of the attack, the attacker invokes victim code with carefully chosen input arguments to trigger misspeculation. This invocation occurs over some communication interface between the untrusted and trusted code. One such interface is through system calls and returns; here the attacker is an untrusted user-mode process while the victim is the operating system kernel. Another example in the context of browser-based sandboxing, e.g. Native Client [229], would be function calls and returns. The attacker is untrusted code running within the sandbox while the victim is NaCl's trusted API. Many other vulnerable interfaces exist: hypercalls, enclave entry, software interrupts, etc. The victim may not even be explicitly invoked: implicit invocation is possible by mistraining the branch predictor or by causing a hardware interrupt to occur! For simplicity, we focus on a function call/return interface but our techniques are easily generalized to other interfaces.

**(S3) Exploiting misspeculation**. In the third stage of the attack, the victim code executes, typically through a call by the attacker. At some point, it will misspeculate in an attacker-controlled manner resulting in the execution of "wrong path" instructions. These wrong path instructions update speculative architectural state – register and memory values – and microarchitectural state including caches, branch predictors, and prefetchers. Eventually, the wrong path is resolved and its instructions are flushed. Speculative updates to the architectural state (registers and memory) are flushed, but the microarchitectural state (e.g., cache updates) is not restored.

While many past attacks have exploited microarchitectural side channels to extract confidential data [2, 105, 128, 134, 162, 165, 191], the difference with transient execution vulnerabilities is that the latter only manifest due to misspeculation in the processor. *Even programs whose architectural (non-speculative) execution is carefully designed to not have any side-channel leaks could be vulnerable to transient execution attacks.*

**(S4) Exfiltration**. Finally, control returns to the attacker who examines the microarchitectural side-channel state to exfiltrate confidential data from the victim. In cache-based side channel attacks, this involves *reloading* or *probing* the cache in order to infer secrets.

## 4.2.2 Spectre Variants and Associated Verification Challenges

We now describe Spectre variant 1, *the bounds check bypass attack*, as the running example of transient execution vulnerabilities. We will then use this example to explore various modifications of the code to explain a standard approach to mitigate the vulnerability and how the vulnerability can be conditional. These examples will provide a sense of how powerful these vulnerabilities are and how precise the analysis needs to be for secure speculation.

```
uint8_t a1[M];
uint8_t a2[P];
uint8_t foo(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    return a2[v*S];
  }
  return 0;
}
```

Figure 4.2: Spectre v1 vulnerability.

Each of the examples contains a victim function named `foo`, similar to Figure 4.2. This function is trusted but it is invoked by an untrusted attacker with an arbitrary attack-chosen `i`. `foo` has access to two arrays: `a1` and `a2`. Note that any architectural execution of these functions should never see accesses to `a1[i]` for $\texttt{i} \geq \texttt{N}$. Therefore, one might expect that no information could possibly leak about these values in the array through any side-channel. As we will see, the Spectre attack shows how these values can be inferred by a clever attacker. In the following code snippets, we assume $\texttt{M} > \texttt{N}$ and the argument `i` is an untrusted (low-security) input to the trusted (high-security) function `foo`. For simplicity, we also assume that the cache is direct-mapped.

## Spectre Variant 1 Walkthrough

Figure 4.2 shows a snippet of code that demonstrates vulnerability to the Spectre variant 1 attack [116]. To help explain the vulnerability, we show how the cache state evolves during each stage of the attack in Figure 4.3.

**(S1) Prepare**. First, the attacker sets up (i.e., "primes") the cache by bringing two addresses `A` and `B` into the cache. These addresses are carefully chosen so that they reside in the same cache set as the subsequently-accessed addresses `a2[0]` and `a2[S]` respectively. Next, the attacker mistrains the branch predictor so that in a subsequent execution of the branch `if (i < N)` at line 4, the microprocessor will take the branch regardless of the actual value of `i < N`. This completes the preparations for the attack.

**(S2) Invocation of victim code**. The attacker then invokes `foo` with the argument $\texttt{i} = \texttt{N} + 2$. We note that the `+2` is used to bring the first memory access outside of what is considered the public part of memory and thus, the value accessed `a1[N+2]` is potentially a secret.

**(S3) Exploiting misspeculation**. The argument $\texttt{i} = \texttt{N} + 2$ along with branch predictor mistraining in S1 triggers a misspeculation on line 4. This results in `a1[N+2]` and `a2[a1[N+2]*S]` being speculatively brought into the cache. Eventually, the processor realizes that the branch prediction was incorrect and "undoes" modifications to the architectural state, but the cache state is *not* restored.

**(S4) Exfiltration**. In the final stage, the attacker exploits the fact that the address brought into the cache on line 6 depends on the *value* (not address) of `a1[N+2]`. The attacker determines this address by loading `A` and `B`. One of these will miss in the cache and this timing channel allows the attacker to infer the value of `a1[N+2]`.

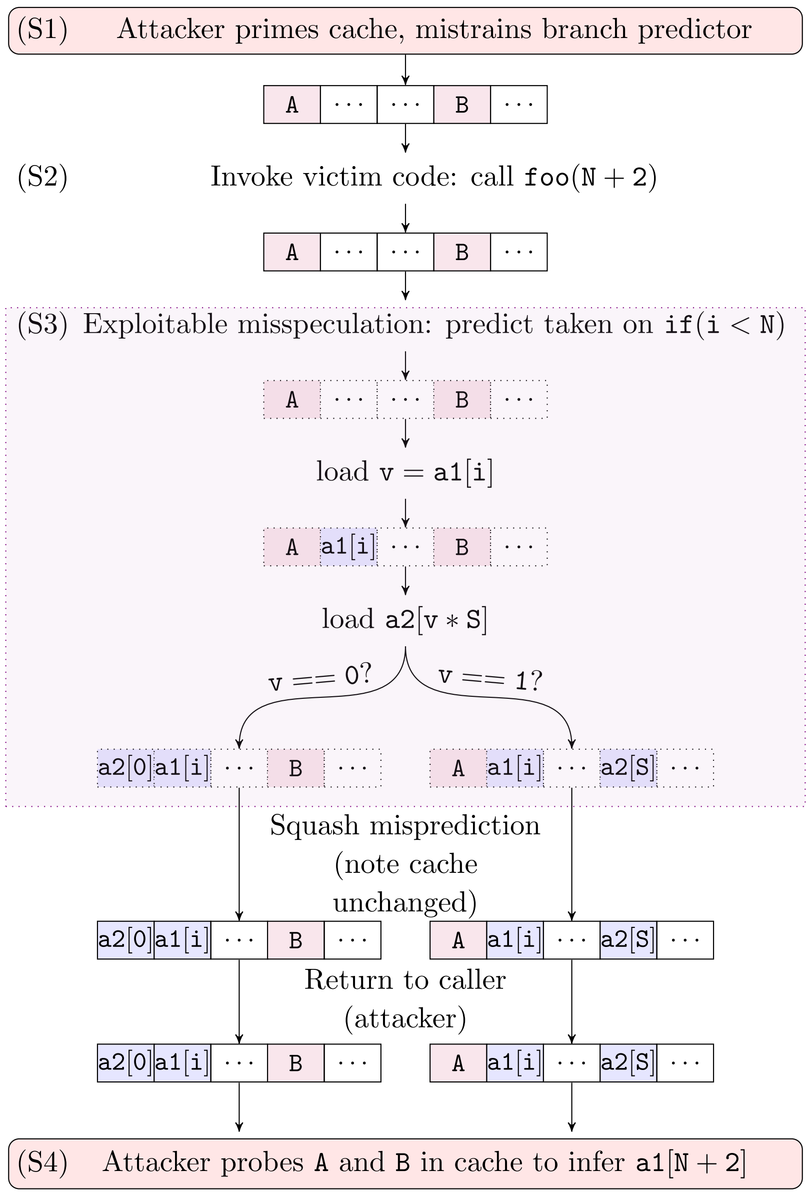

(S1) Attacker primes cache, mistrains branch predictor
A
· · ·
· · ·
B
· · ·
(S2) Invoke victim code: call foo(N + 2)
(S3) Exploitable misspeculation: predict taken on if(i < N)
load v = a1[i]
a1[i]
load a2[v ∗ S]
v == 0?
v == 1?
a2[0]
a2[S]
Squash misprediction (note cache unchanged)
Return to caller (attacker)
(S4) Attacker probes A and B in cache to infer a1[N + 2]

Figure 4.3: Cache state evolution in Spectre variant 1. The rectangular boxes show the addresses that are cached. Untrusted accesses are red while accesses by trusted code are blue. For simplicity, we show the attack on a direct-mapped cache.

```
1 uint8_t a1[M];
2 uint8_t a2[P];
3 uint8_t foo(unsigned i) {
4   if (i < N) {
5     _mm_lfence();
6     return a2[a1[i]*S];
7   }
8  return 0;
9 }
```

(a) Fix for Spectre v1.

```
1 uint8_t a1[M], a2[P];
2 uint8_t foo(unsigned i) {
3   if (i < N) {
4     uint8_t v = a1[i];
5     _mm_lfence();
6     return a2[v*S];
7   }
8  return 0;
9 }
```

(b) Another fix for Spectre v1.

Figure 4.4: Illustrative examples of fixes to vulnerable snippet in Figure 4.2.

### Fixes to Spectre Variant 1

As the leaks in Spectre are due to interactions between the branch predictor and the cache, a straightforward fix is to prevent speculation. Figure 4.2 is made secure by inserting a *load fence* [94, 95] as shown in Figures 4.4a and 4.4b. In Figure 4.4a, the load fence on line 5 ensures that no memory accesses are made until the processor is sure that the branch will be taken.

Figure 4.4b on the other hand, is slightly more involved. The load fence executes after the first load and before the second load. At first glance, it may appear to be insecure, because `a1[i]` can still be brought into the cache speculatively. However `i` is attacker-chosen while the base address of `a1` can also be inferred by the attacker. Therefore, bringing `a1[i]` into the cache leaks no additional information. Figure 4.4b is secure.

### A Conditionally Vulnerable Variant of Spectre

Figure 4.5 presents an interesting variation of Figure 4.2. In this case, the first memory load always accesses `a1[0]`. Since this value is leaked through the cache (when $\texttt{i} < \texttt{N}$) even without misspeculation, it would seem that this code is not vulnerable to transient execution attacks. However, if $\texttt{N} = 0$, then `a1[0]` should not be accessed. But the attacker can mistrain the branch predictor to predict that the branch on line 4 is taken[*] and then infer the value of `a1[0]`. This code exhibits transient execution vulnerabilities when $\texttt{N} = 0$ but not when $\texttt{N} > 0$.

```
1 uint8_t a1[M], a2[P];
2 uint8_t foo(unsigned i) {
3   if (i < N) {
4     return a2[a1[0]*S]+i;
5   }
6   return 0;
7 }
```

Figure 4.5: Conditional variant.

[*]One way to do this might be to exploit aliasing in branch predictor indexing by training a different branch which maps to the same predictor index.

#### Verification Challenges

In Figure 4.2, information about `a1[i]` leaked when `i` $=$ `N` $+ 2$. For this value of `i`, `foo` should not have performed any memory/cache accesses. This points to one challenge in verifying secure speculation: the verification model needs to capture *interactions* between *microarchitectural side-channels* to detect leaks.

Another challenge is demonstrated by Figures 4.4a, 4.5 and 4.4b. Identifying the vulnerability requires precise semantic analysis of program behavior. Static analysis and dynamic analysis of code are often not sufficient because they result in a number of false positives and negatives.

It is also important to note that the secure versions `foo` in Figures 4.4a and 4.4b do not satisfy traditional notions of information flow security [51]: noninterference [76] or observational determinism [142, 178, 233] because there *is* information flow from `a1` to the cache side-channel even if the function is executed on a processor without a branch predictor.

Finally, work on formally verifying practical programs at the instruction level often does not come without sacrificing accuracy or running into the path explosion problem. As a result, one needs to be delicate in using the appropriate theorem provers and have a clever encoding of the model with the right logical theories to avoid these issues.

### 4.2.3 Scope and Assumptions

Considering these challenges, we reduce our approach to a particular scope and make several simplifying assumptions. We list these below:

- Basic blocks of programs make up the atomic transitions of the platform model as our work targets verifying speculative in-order processors. While this can accurately capture the semantics of in-order programs, we acknowledge that practical platforms execute instructions out of order aggressively.

- While the adversary can exploit the cache side channel, its capabilities are not omnipotent, and reading from the LLC cache [134] can be difficult. However, we elide the detail that observations are probabilistic. This allows us to over-approximate the observations of the adversary to preserve soundness and simplify the logic required to model the attack.

## 4.3 Specification using Trace Property-Dependent Observational Determinism

To address the challenges raised in § 4.2.2, we formulate a secure speculation property that precisely captures transient execution vulnerabilities. Toward this end, we first review observational determinism in the context of this model. We will discuss the shortcomings of observational determinism for capturing transient execution attacks. We then motivate and describe trace property-dependent observational determinism: a novel class of information flow properties that includes secure speculation.

### 4.3.1 Programming Model

We begin by introducing the programming model under which observation determinism will be defined. This is necessary to precisely capture the execution patterns of the adversary and victim, as well as what parts of the system are trusted and untrusted. Naturally, we can introduce the components based off the definition of $\mathcal{L}_2$ consist of two parts: the operations that the platform executes and the system state that can be read and written.

**Low and high components**. To specify these components, we formalize them as two distinct operations, $op_{\texttt{L}}$ and $op_{\texttt{H}}$. The low component consists of the untrusted adversary and public components of the platform. The high component consists of the trusted victim program and confidential memory owned by the victim. We formalize this difference in security domains using the following operations. We write $op_{\texttt{L}}(q)$ to denote the operation executed by the low component in a particular state $q$; $op_{\texttt{L}}(q)$ is $\perp$ if the low component is not being executed in state $q$. Similarly, the operation executed by the high component in the state $s$ is denoted by $op_{\texttt{H}}(q)$. We overload notation and refer to $op_{\texttt{L}}(\pi)$ and $op_{\texttt{H}}(\pi)$ to denote the trace of operations executed by the low and high components respectively in $\pi$. Formally, this is defined as $op_{\texttt{L}}(\pi) = op_{\texttt{L}}(\pi^{(0)})op_{\texttt{L}}(\pi^{(1)})op_{\texttt{L}}(\pi^{(2)})...op_{\texttt{L}}(\pi^{(n)})$ and similarly defined for $op_{\texttt{H}}$.

**Execution with the adversary**. To generalize to a wide range of programming models, our programming model allows an adversary to execute interleaved with the victim. At any point during the adversary execution, it is allowed to execute for a potentially unbounded number of steps to observe or tamper the platform variables. We provide details on the specifics of the adversary model we use for the Spectre attack in section §4.5.1.

### 4.3.2 Low Operation-Equivalent Extension of Observational Determinism

We extend the observational determinism for the programming model presented above, which introduced low and high operations corresponding to the operations of the adversary and the trusted victim program respectively. This allows us to filter traces where different untrusted operations are the reason for a violation of the property. This extension is described as Eq. 4.1, which specializes Eq. 3.2 by adding an assumption to constrain the low operations of the two traces. This assumption is based on the constant-time programming discipline [42].

$$\forall \pi_1, \pi_2.\ \left(\pi_1^{(0)} \approx_{\texttt{L}} \pi_2^{(0)} \wedge op_{\texttt{L}}(\pi_1) = op_{\texttt{L}}(\pi_2)\right) \implies \left(\pi_1 \approx_{\texttt{L}} \pi_2\right) \tag{4.1}$$

$$\begin{tikzcd}
\pi_1^{(0)} \arrow[r, "L"] \arrow[d, phantom, sloped, "\approx_L"] & \cdots & \pi_1^{(i)} \arrow[r, "H_1"] \arrow[d, phantom, sloped, "\approx_L"] & \pi_1^{(i+1)} \arrow[r, "L"] \arrow[d, phantom, sloped, "\approx_L"] & \pi_1^{(i+2)} \arrow[d, phantom, sloped, "\approx_L"] & \cdots & \pi_1^{(j)} \arrow[r, "H_1"] \arrow[d, phantom, sloped, "\approx_L"] & \pi_1^{(j+1)} \arrow[r, "L"] \arrow[d, phantom, sloped, "\approx_L"] & \cdots \\
\pi_2^{(0)} \arrow[r, "L"] & \cdots & \pi_2^{(i)} \arrow[r, "H_2"] & \pi_2^{(i+1)} \arrow[r, "L"] & \pi_2^{(i+2)} & \cdots & \pi_2^{(j)} \arrow[r, "H_2"] & \pi_2^{(j+1)} \arrow[r, "L"] & \cdots
\end{tikzcd}$$

Figure 4.6: Illustrating observational determinism in the context of the low attacker and high (trusted) program: low instructions are labeled $L$, while high instructions are labeled $H_1$ and $H_2$, proof obligations are shown in green and assumptions are shown in blue.

Figure 4.6 depicts observational determinism, along with the initial assumptions and proof obligations required for it to hold on a platform. Initially, the traces start in low-equivalent states $\pi_1^{(0)}$ and $\pi_2^{(0)}$. In each subsequent step of the two instances, the low operations are assumed to be identical denoted by $L$ above the state transition arrows. However, when the trusted program is executing, each instance may execute different high operations $H_1$ and $H_2$. Similar to Eq. 3.2, this variant of observational determinism holds if every corresponding pair of states in the two traces are low-equivalent. Thus, a violation of this property would correspond to some execution of the instances that end in states being distinguishable from each other.

**Limitations of Observation Determinism for Secure Speculation**

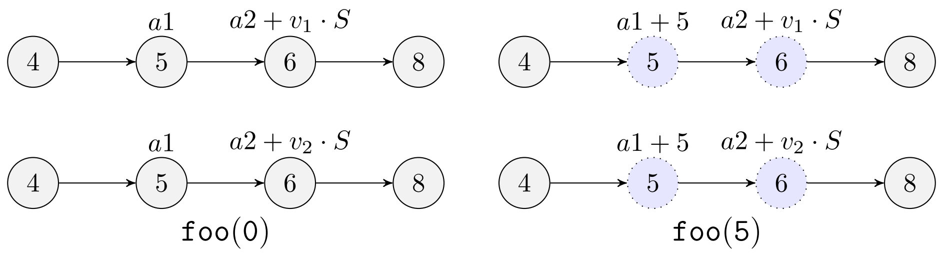


Figure 4.7: Illustrating the strawman observational determinism property for Figure 4.2. Numbers within each state refer to program counter values (shown as line numbers). Labels above each state indicates the data memory address accessed (if any). States shown in dotted circles are specuative states.

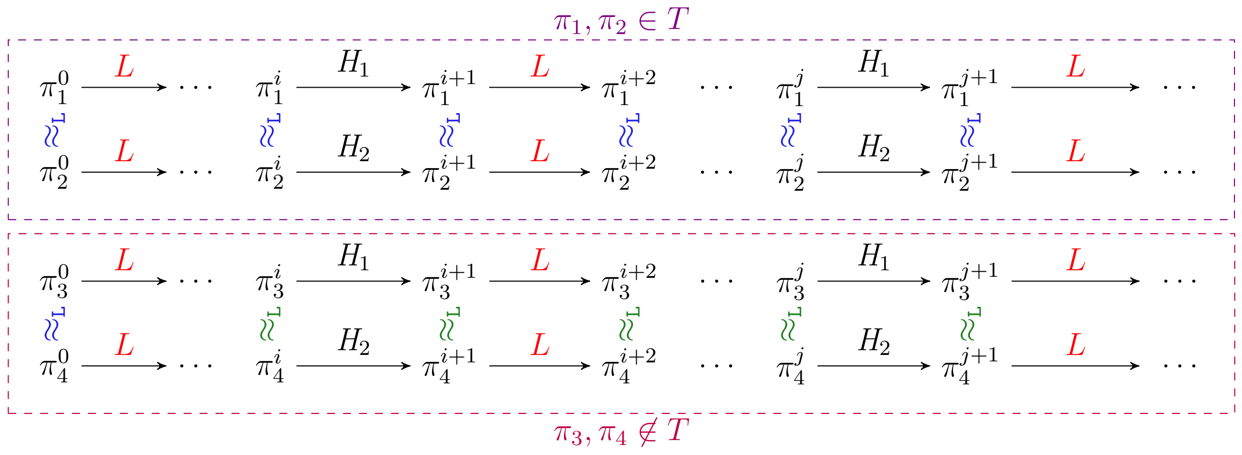
As a strawman proposal, consider an observational determinism property that attempts to capture secure speculation by requiring that the trace of memory accesses by the function `foo` in Figure 4.2 be identical for all pairs of invocations where the untrusted argument $i$ is equal.

Two such pairs of traces are shown in Figure 4.7. `N=4` in both pairs; in (a), `i=0` and the program does not misspeculate while in (b), `i=5` and the program misspeculates. The values

$v_1$ and $v_2$ correspond to the confidential data stored at the location `a1[i]`. We see that the property is violated in (b) as the traces of memory addresses differ if $v_1 \neq v_2$. This violation is due to the transient execution vulnerability. It is also violated in (a) because the program leaks `a1[i]` even though there is no misspeculation. The larger point is that observational determinism can capture transient execution vulnerabilities only if the program satisfies the observational determinism property – has zero violations of the property – in the absence of misspeculation. Most programs of interest (e.g., the Linux kernel) do not satisfy such a property. Applying the strawman methodology to these programs results in a flood of counterexamples to observational determinism that are completely unrelated to speculation, rendering the methodology useless.

We wish to isolate violations of observational determinism solely caused by the satisfaction/violation of a particular property of the trace (e.g., misspeculation). As noted above, observational determinism does not allow us to do this generally for different programs.[†] In the following, we capture this security requirement in the form of a 4-safety property to isolate these trace property-dependent violations.

## 4.3.3 Trace Property-Dependent Observational Determinism



$\pi_1, \pi_2 \in T$
$\pi_1^0 \xrightarrow{L} \cdots \quad \pi_1^i \xrightarrow{H_1} \pi_1^{i+1} \xrightarrow{L} \pi_1^{i+2} \quad \cdots \quad \pi_1^j \xrightarrow{H_1} \pi_1^{j+1} \xrightarrow{L} \cdots$
$\approx_L$
$\pi_2^0 \xrightarrow{L} \cdots \quad \pi_2^i \xrightarrow{H_2} \pi_2^{i+1} \xrightarrow{L} \pi_2^{i+2} \quad \cdots \quad \pi_2^j \xrightarrow{H_2} \pi_2^{j+1} \xrightarrow{L} \cdots$
$\pi_3^0 \xrightarrow{L} \cdots \quad \pi_3^i \xrightarrow{H_1} \pi_3^{i+1} \xrightarrow{L} \pi_3^{i+2} \quad \cdots \quad \pi_3^j \xrightarrow{H_1} \pi_3^{j+1} \xrightarrow{L} \cdots$
$\approx_L$
$\pi_4^0 \xrightarrow{L} \cdots \quad \pi_4^i \xrightarrow{H_2} \pi_4^{i+1} \xrightarrow{L} \pi_4^{i+2} \quad \cdots \quad \pi_4^j \xrightarrow{H_2} \pi_4^{j+1} \xrightarrow{L} \cdots$
$\pi_3, \pi_4 \notin T$

Figure 4.8: Illustrating trace property-dependent observational determinism. As in Figure 4.6 low instructions are labelled $L$, while high instructions are labelled $H_1$ and $H_2$, proof obligations are shown in green and assumptions are shown in blue.

In a processor that never misspeculates – either because it does not speculate or because speculation is perfect – there is no information leakage due to transient execution. Therefore, finding transient execution vulnerabilities is equivalent to finding information leaks that would not have occurred in the absence of misspeculation.

[†]We explain this further in Section 4.6.

To formulate the notion of information leakage above, we introduce a class of information flow properties called trace property-dependent observational determinism (TPOD), a hyperproperty over four traces $\pi_1, \pi_2, \pi_3, \pi_4$ that is defined with respect to a trace property $T$.

**Definition 4.1** (Trace Property-Dependent Observational Determinism). *Suppose the following assumptions hold:*

1. *traces $\pi_1$ and $\pi_2$ satisfy the trace property $T$,*
2. *traces $\pi_3$ and $\pi_4$ do not satisfy the trace property $T$,*
3. *all four traces execute the same low operations,*
4. *traces $\pi_3$ and $\pi_4$ execute the same high operations as $\pi_1$ and $\pi_2$ respectively,*
5. *traces $\pi_1$ and $\pi_2$ are low-equivalent and the initial states of $\pi_3$ and $\pi_4$ are low-equivalent.*

*Then, TPOD is satisfied if $\pi_3$ and $\pi_4$ are low-equivalent. High operations in $\pi_1$, $\pi_3$ and $\pi_2$, $\pi_4$ respectively must be identical; they are not necessarily identical in $\pi_1$, $\pi_2$ or $\pi_3$, $\pi_4$. This is formalized as Eq 4.2 below.*

$$\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4 \in Tr(M).\\
&\quad \pi_1 \in T \wedge \pi_2 \in T \wedge \pi_3 \notin T \wedge \pi_4 \notin T \implies\\
&\quad op_{\mathtt{L}}(\pi_1) = op_{\mathtt{L}}(\pi_2) = op_{\mathtt{L}}(\pi_3) = op_{\mathtt{L}}(\pi_4) \implies\\
&\quad op_{\mathtt{H}}(\pi_1) = op_{\mathtt{H}}(\pi_3) \wedge op_{\mathtt{H}}(\pi_2) = op_{\mathtt{H}}(\pi_4) \implies\\
&\quad \pi_1 \approx_{\mathtt{L}} \pi_2 \wedge \pi_3^{(0)} \approx_{\mathtt{L}} \pi_4^{(0)} \implies\\
&\quad \pi_3 \approx_{\mathtt{L}} \pi_4
\end{aligned} \tag{4.2}$$

TPOD is shown in Equation 4.2[‡] and depicted in Figure 4.8. A violation of TPOD corresponds to a sequence of low operations that were unable to distinguish between high states when the trace property $T$ was satisfied, but *are* able to distinguish between high states when $T$ is not satisfied. In other words, violation of the trace property $T$ introduced a new counterexample to observational determinism. Naturally, whether a platform speculates can be expressed as a trace property. We use this insight to define secure speculation by specializing TPOD in §4.4.

[‡]We follow the convention that the implication operator is right-associative.

### Refinement and TPOD

In general, hyperproperties may not be preserved by refinement [178]. However, as we show below TPOD is subset-closed: if any set of traces satisfies TPOD, then every subset of this set also satisfies TPOD.

**Lemma 4.1.** *TPOD is a subset-closed hyperproperty.*

*Proof.* Assume some instance of TPOD, say $\varphi$, is not subset-closed. There must exist some sets of traces $T_1$ and $T_2$ such that $T_1 \subset T_2$, $T_1 \not\models \varphi$ and $T_2 \models \varphi$. Then there exists a tuple of traces $\pi \in T_1$ that violates $\varphi$. Since $T_1 \subset T_2$ then $\pi \in T_2$ and thus $T_2 \not\models \varphi$. Contradiction. $\square$

Subset-closed hyperproperties are important because they are preserved by refinement [51]. This means that one can prove TPOD on an abstract system, and through iterative refinement show that TPOD holds on a concrete system that is a refinement of the abstract system. Therefore, TPOD can potentially be scalably verified on complex systems.

**Corollary 4.1.** *TPOD is preserved by refinement.*

A minor extension to the template shown in Equation 4.2 is to consider an antecedent trace property $U$ that must be satisfied by all traces. The trace property $U$ may be used to model constraints on valid executions.

$$\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4 \in Tr(M).\\
&\quad \pi_1 \in U \wedge \pi_2 \in U \wedge \pi_3 \in U \wedge \pi_4 \in U \implies\\
&\quad \pi_1 \in T \wedge \pi_2 \in T \wedge \pi_3 \notin T \wedge \pi_4 \notin T \implies\\
&\quad op_{\mathtt{L}}(\pi_1) = op_{\mathtt{L}}(\pi_2) = op_{\mathtt{L}}(\pi_3) = op_{\mathtt{L}}(\pi_4) \implies\\
&\quad op_{\mathtt{H}}(\pi_1) = op_{\mathtt{H}}(\pi_3) \wedge op_{\mathtt{H}}(\pi_2) = op_{\mathtt{H}}(\pi_4) \implies\\
&\quad \pi_1 \approx_{\mathtt{L}} \pi_2 \wedge \pi_3^{(0)} \approx_{\mathtt{L}} \pi_4^{(0)} \implies\\
&\quad \pi_3 \approx_{\mathtt{L}} \pi_4
\end{aligned} \tag{4.3}$$

This version of TPOD is shown in Equation 4.3. This extension is also subset-closed and preserved by refinement.

## 4.4 Formal Modeling of Speculation

We now focus on formulating and reasoning about secure speculation. This requires formalizing a platform that can capture the relevant hardware-software states to model speculation.

We accomplish this by introducing the speculative platform model along with an instruction set assembly intermediate representation associated speculative operational semantics which combines updates to hardware and software state.

### 4.4.1 The Speculative Platform Model

A platform model that can reason about secure speculation must be capable of interfacing a platform's software components with its hardware components. The former instructs how one orchestrates an attack, and the latter makes up the components being exploited. This means that the platform must model programs with at least some notion of assembly language instructions, where the behavior of hardware can be incorporated as a side-effect of the instructions.

The other consideration for our platform model is the level of abstraction it is described. While one can attempt to be precise and implement the model at the RTL or implementation level, this is not scalable and existing literature on generating sound models from RTL has seen limited success in formal verification. Modeling specific architectures can also be cumbersome and gives limited insight into the underlying causes of transient execution vulnerabilities. Moreover, modeling at this level of detail prevents one from capturing secure speculation for a class of platform designs and becomes somewhat ad-hoc.

Thus, we present an abstract instruction interface, assembly intermediate representation (AIR), which ISAs can be lifted into. We encode speculation as part of the semantics of the instructions provided by this interface. We refer to this as *speculative operational semantics*[§].

Formally, the speculative platform model can be defined as a specialization of the state-transition system model in §3.1 where the operation set is $\mathsf{Op} \doteq \{\textsc{RegisterUpdate}, \textsc{Load}, \textsc{Store}, \text{T-}\textsc{Pred}, \text{T-}\textsc{Mispred}, \text{NT-}\textsc{Pred}, \text{NT-}\textsc{Mispred}, \textsc{Goto}, \textsc{SpecFence}, \textsc{Resolve}\}$. These operations correspond to the instructions defined in AIR and are explained below.

#### Assembly Intermediate Representation (AIR)

The AIR is shown in Figure 4.9. A program is a list of instructions. Instructions are one of the following types: updates to registers, loads from memory, stores to memory, conditional and unconditional jumps, and speculation fences.

The first five types of instructions are standard. We introduce a *speculation fence* instruction which causes the processor to not fetch any more instructions until all outstanding branches are resolved. The load fence instructions in Figures 4.4a and 4.4b are modeled as speculation fences because the relevant aspect of these fences is that they stop speculation.

Note that jump targets must be constants in AIR. This is intentional and precludes the verification of programs using indirect jumps and returns in the current version of our verification tool. We do this to simplify the operational semantics for speculative execution.

[§]The AIR is based on the binary analysis platform (BAP) intermediate language (IL) [34]; the speculative operational semantics is the first to be introduced for an instruction-level interface. “Lifters” from x86 and ARM binaries to BAP can be found at [25].

$$
\begin{aligned}
\langle \mathit{program} \rangle &::= \langle \mathit{instr} \rangle^* \\[1ex]
\langle \mathit{instr} \rangle &::= \langle \mathit{reg} \rangle := \langle \mathit{exp} \rangle \\
&\mid \langle \mathit{reg} \rangle := \texttt{mem}[\langle \mathit{exp} \rangle] \\
&\mid \texttt{mem} := \texttt{mem}[\langle \mathit{exp} \rangle \to \langle \mathit{exp} \rangle] \\
&\mid \texttt{if}\ \langle \mathit{exp} \rangle\ \texttt{goto}\ \langle \mathit{const} \rangle \\
&\mid \texttt{goto}\ \langle \mathit{const} \rangle \\
&\mid \texttt{specfence} \\[1ex]
\langle \mathit{exp} \rangle &::= \langle \mathit{const} \rangle \mid \langle \mathit{reg} \rangle \mid \Diamond_u \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle\ \Diamond_b \langle \mathit{exp} \rangle
\end{aligned}
$$

Figure 4.9: The Assembly Intermediate Representation (AIR). $\Diamond_u$ and $\Diamond_b$ are typical unary and binary operators respectively.

Modeling speculative execution of indirect jumps and returns requires modeling indirect branch predictors, branch target buffers and the return address stack. Introducing these structures into our operation semantics is conceptually straightforward but runs into scalability limitations during verification. We plan to extend the operational semantics to include these instructions while addressing scalability in future work.[¶]

$$
\textsc{Const}\ \frac{}{\Delta, n \vdash c \Downarrow c} \qquad \textsc{Reg}\ \frac{\Delta[n, r] = v}{\Delta, n \vdash r \Downarrow v}
$$

$$
\textsc{UnOp}\ \frac{\Delta, n \vdash e \Downarrow v' \qquad \Diamond_u v' = v}{\Delta, n \vdash \Diamond_u e \Downarrow v}
$$

$$
\textsc{BinOp}\ \frac{\Delta, n \vdash e_1 \Downarrow v_1 \qquad \Delta, n \vdash e_2 \Downarrow v_2 \qquad v_1 \Diamond_b v_2 = v}{\Delta, n \vdash e_1 \Diamond_b e_2 \Downarrow v}
$$

Figure 4.10: Semantics of expression evaluation

"Flattening" indirect jumps and returns into a sequence of direct jumps is similar in principle to control-flow integrity (CFI) checks [1, 207]. Since secure programs will likely be implementing CFI anyway, we assert compilers can be modified in straightforward ways to produce code without indirect jumps and returns (with some performance cost).

## Operational Semantics for AIR

In Figures 4.10 and 4.11, we introduce operational semantics for **speculative in-order** processors. We model speculation in the branch predictor for direct conditional branches.

[¶]It is important to note that our exclusion of indirect jumps does not mean our verifier leaves programs vulnerable to Spectre variant 2. In programs without indirect branches, all indirect branch mispredictions will be redirected at decode, long before execution or memory access.

$$\textsc{RegisterUpdate}\ \frac{\begin{array}{ccccc}\Delta, n \vdash e \Downarrow v & \Delta' = \Delta[(n,r) \to v] & \rho = pc[n] & \iota = \Pi[\rho] & pc' = pc[n \to \rho + 1] \\ \rho' = pc'[n] & \iota' = \Pi[\rho'] & \omega' = \omega.\langle \rho, \bot \rangle & \neg \mathit{resolve}(n, \beta, pc) & \end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, r := e \quad \rightsquigarrow \quad \Pi, \Delta', \mu, pc', \omega', \beta, n, \iota'}$$

$$\textsc{Load}\ \frac{\begin{array}{ccccc}\Delta, n \vdash e \Downarrow a & \mu[n,a] = v & \Delta' = \Delta[(n,r) \to v] & \rho = pc[n] & pc' = pc[n \to \rho + 1] \\ \iota = \Pi[\rho] & \rho' = pc'[n] & \iota' = \Pi[\rho'] & \omega' = \omega.\langle \rho, a \rangle & \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, r := \texttt{mem}[e] \quad \rightsquigarrow \quad \Pi, \Delta', \mu, pc', \omega', \beta, n, \iota'}$$

$$\textsc{Store}\ \frac{\begin{array}{ccccc}\Delta, n \vdash e_1 \Downarrow a & \Delta, n \vdash e_2 \Downarrow v & \mu' = \mu[(n,a) \to v] & \rho = pc[n] & pc' = pc[n \to \rho + 1] \\ \iota = \Pi[\rho] & \rho' = pc'[n] & \iota' = \Pi[\rho'] & \omega' = \omega.\langle \rho, a \rangle & \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{mem} := \texttt{mem}[e_1 \to e_2] \quad \rightsquigarrow \quad \Pi, \Delta, \mu', pc', \omega', \beta, n, \iota'}$$

$$\textsc{T-Pred}\ \frac{\begin{array}{ccccc}\Delta, n \vdash e \Downarrow \texttt{true} & \mathit{mispred}(n, \beta, pc) = \texttt{false} & \rho = pc[n] & \iota = \Pi[\rho] & pc' = pc[n \to c] \\ \rho' = pc'[n] & \iota' = \Pi[\rho'] & \omega' = \omega.\langle \rho, \bot \rangle & \beta' = \mathit{update}(n, \rho, \iota, \beta) & \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if } e \texttt{ goto } c \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc', \omega', \beta', n, \iota'}$$

$$\textsc{T-Mispred}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow \texttt{true} \quad \mathit{mispred}(n, \beta, pc) = \texttt{true} \quad \rho = pc[n] \quad \iota = \Pi[\rho] \\ n' = n + 1 \quad pc' = pc[n' \to \rho + 1, n \to c] \quad \rho' = pc'[n'] \\ \iota' = \Pi[\rho'] \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc) \\ \forall m, r \in \mathbb{N}.\ \Delta'[m,r] = \textsc{ite}(m = n', \Delta[n,r], \Delta[m,r]) \\ \forall m \in \mathbb{N}, a \in \mathbb{A}.\ \mu'[m,a] = \textsc{ite}(m = n', \mu[n,a], \mu[m,a])\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if } e \texttt{ goto } c \quad \rightsquigarrow \quad \Pi, \Delta', \mu', pc', \omega', \beta', n', \iota'}$$

$$\textsc{NT-Pred}\ \frac{\begin{array}{ccccc}\Delta, n \vdash e \Downarrow \texttt{false} & \mathit{mispred}(n, \beta, pc) = \texttt{false} & \rho = pc[n] & \iota = \Pi[\rho] & pc' = pc[n \to \rho + 1] \\ \rho' = pc'[n] & \iota' = \Pi[\rho'] & \omega' = \omega.\langle \rho, \bot \rangle & \beta' = \mathit{update}(n, \rho, \iota, \beta) & \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if } e \texttt{ goto } c \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc', \omega', \beta', n, \iota'}$$

$$\textsc{NT-Mispred}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow \texttt{false} \quad \mathit{mispred}(n, \beta, pc) = \texttt{true} \quad \rho = pc[n] \quad \iota = \Pi[\rho] \\ n' = n + 1 \quad pc' = pc[n' \to c, n \to \rho + 1] \quad \rho' = pc'[n'] \\ \iota' = \Pi[\rho'] \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc) \\ \forall m, r \in \mathbb{N}.\ \Delta'[m,r] = \textsc{ite}(m = n', \Delta[n,r], \Delta[m,r]) \\ \forall m \in \mathbb{N}, a \in \mathbb{A}.\ \mu'[m,a] = \textsc{ite}(m = n', \mu[n,a], \mu[m,a])\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if } e \texttt{ goto } c \quad \rightsquigarrow \quad \Pi, \Delta', \mu', pc', \omega', \beta', n', \iota'}$$

$$\textsc{Goto}\ \frac{\begin{array}{cccc}\rho = pc[n] & \iota = \Pi[\rho] & pc' = pc[n \to c] & \rho' = pc'[n] \\ \iota' = \Pi[\rho'] & \omega' = \omega.\langle \rho, \bot \rangle & \beta' = \mathit{update}(n, \rho, \iota, \beta) & \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{goto } c \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc', \omega', \beta', n, \iota'}$$

$$\textsc{SpecFence}\ \frac{n' = 0 \quad \rho' = pc[n'] \quad \iota' = \Pi[\rho'] \quad \neg \mathit{resolve}(n, \beta, pc)}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc, \omega, \beta, n', \iota'}$$

$$\textsc{Resolve}\ \frac{n' = n - 1 \quad n > 0 \quad \rho' = pc[n'] \quad \iota' = \Pi[\rho'] \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \mathit{resolve}(n, \beta, pc)}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc, \omega, \beta', n', \iota'}$$

$$\textsc{havoc}\ \frac{\begin{array}{cccc}n = 0 & pc[0] \notin \mathcal{T}_\rho & \rho' = pc'[0] & \iota' = \Pi[\rho'] \\ \multicolumn{2}{c}{\forall a \in \mathbb{A}.\ a \notin \mathcal{U}_\mu^{wr} \implies \mu'[0,a] := \mu[0,a]} & \multicolumn{2}{c}{\forall m \in \mathbb{N}, a \in \mathbb{A}.\ m > 0 \Rightarrow \mu'[m,a] = \mu[m,a]}\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{havoc } (\Delta,\ \texttt{mem}[\mathcal{U}_\mu^{wr}],\ \beta) \quad \rightsquigarrow \quad \Pi, \Delta', \mu', pc', \omega', \beta', n, \iota'}$$

Figure 4.11: Operational Semantics for Statements in AIR.

Other sources of misspeculation such as value prediction and memory address disambiguation are not considered in this model. Extending the semantics to include these is conceptually straightforward. However, this could result in models that are difficult to analyze using automated verification tools because the verification engine would need to explore exponentially more instruction orderings.

**Types** of the variables in our platform model can be built upon three standard sets of values. These are the set of words $\mathbb{W}$, the set of instructions $\mathbb{I}$, and the set of addresses $\mathbb{A}$. We assume that all such sets are finite.

**Machine state** $s$ is the tuple $\langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$. $\Pi : \mathbb{A} \to \mathbb{I}$ is the program memory: a map from program counter values to instructions. $\Delta : \mathbb{N} \to \mathbb{W}$ and $\mu : \mathbb{A} \to \mathbb{W}$ are the states of the registers and data memory respectively while $pc : \mathbb{A}$ contains the program counter. $\omega : \mathbb{W}^*$ is the trace of program and data addresses accessed so far (i.e., the trace of observations). $\beta$ is the branch predictor state, which we leave abstract and $\iota : \mathbb{I}$ is the instruction that will be executed next.

The main novelty in these semantics is modeling misspeculation. $n$ is an integer that represents *speculation level*: it is incremented each time we misspeculate on a branch and decremented when a branch is resolved. Speculation level 0 corresponds to architectural (non-speculative) execution. $\Delta$, $\mu$ and $pc$ – registers, memory and program counter respectively – are also indexed by the speculation level. $\Delta[n, r]$ refers to the value of the register $r$ at speculation level $n$. $\Delta[(n, r) \to v]$ refers to a register state which is identical to $\Delta$ except that register $r$ at speculation level $n$ has been assigned value $v$. We adopt similar notation for $\mu$ and $pc$.

**Expression Semantics** are shown in Figure 4.10. Expressions are defined over the register state $\Delta$. Notation $\Delta, n \vdash e \Downarrow v$ means that the expression $e$ evaluates to value $v$ given register state $\Delta$ at speculation level $n$. These are standard except for the additional wrinkle of the speculation level.

**Statement Semantics** are shown in Figure 4.11. The semantics of a transition from the machine state $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$ to the machine state $s' = \langle \Pi', \Delta', \mu', pc', \omega', \beta', n', \iota' \rangle$ by executing an operation from AIR is described using the big-step operational semantics judgment $\rightsquigarrow$: $\langle \Pi, \Delta, \mu, \ pc, \omega, \beta, n, \iota \rangle \ \rightsquigarrow \ \langle \Pi', \Delta', \mu', pc', \omega', \beta', n', \iota' \rangle$. We now describe the judgment rules shown in Figure 4.11.

The RegisterUpdate rule models the execution of statements of the form $r := e$, where expression $e$ is written to the register $r$. This involves: (i) updating the value of register $r$ at speculation level $n$ to have the value of the expression $e$: $\Delta' = \Delta[(n, r) \to v]$, (ii) incrementing the $pc$ at speculation level $n$: $pc' = pc[n \to \rho + 1]$ and (iii) appending $\langle \rho, \perp \rangle$ to the trace of memory addresses accessed by the program: $\omega' = \omega.\langle pc, \perp \rangle$. The $\perp$ in the second element of the tuple indicates that no data memory access is performed by this instruction. This rule is only executed when a branch is not being resolved: $\neg resolve(n, \beta, pc)$ and the next instruction to be executed is $\iota' = \Pi[\rho']$ where $\rho' = pc[n]$.

The LOAD and STORE rules are similar. LOAD updates the register state with value stored at memory location $a$ at speculation level $n$: $v = \mu[n, a]$ while STORE leaves register state $\Delta$ unchanged and updates memory address $a$ at speculation level $n$: $\mu' = \mu[(n, a) \to v]$. Both LOAD and STORE append $\langle \rho, a \rangle$ to the trace of memory addresses signifying accesses to program address $\rho$ and data address $a$. As with REGISTERUPDATE, these rules only apply when a branch is not being resolved in this step: $\neg resolve(n, \beta, pc)$.

The T-PRED rule applies when a conditional jump `if` $e$ `goto` $c$ should be taken and is also predicted taken. In the semantics, we model misspeculation through an uninterpreted function $mispred(n, \beta, pc)$ where $\beta$ is the branch predictor state (left abstract in our model), $n$ is the speculation level and $pc$ is a map from speculation levels to program counter values. This rule only applies when *mispred* evaluates `false`. The rule sets the program counter at speculation level $n$ to $c$: $pc' = pc[n \to c]$ and updates the branch predictor state $\beta'$ using the uninterpreted function *update*. Just like the other rules discussed so far, this applies only when the predicate *resolve* does not hold.

The T-MISPRED rule applies when a conditional jump `if` $e$ `goto` $c$ should be taken but is predicted not taken (*mispred* evaluates to `true`). This rule changes system state in the following ways. First, the speculation level is incremented: $n' = n + 1$. Second, the state of the registers at level $n$ in $\Delta$ is now copied over to level $n'$ in $\Delta'$ while all other levels are identical between $\Delta$ and $\Delta'$. The memory state $\mu$ is also modified in a similar way. The program counter at level $n$ gets the correct target $c$, while the program counter at level $n'$ gets the mispredicted fall-through target $\rho + 1$. Execution continues at speculation level $n'$.

NT-PRED, NT-MISPRED handle the case when the conditional branch should *not* be taken. These are similar to T-PRED and T-MISPRED. GOTO applies to direct jumps. Note we do not consider misprediction of direct jumps as they have constant targets and will be redirected at decode.

The rule SPECFENCE resolves all outstanding speculative branches by setting the speculation level back to zero. Note that $pc$, $\Delta$ and $\mu$ at level zero already have the "correct" values, so nothing further needs to be done.

The rule RESOLVE applies when a mispredicted branch is resolved. Resolution occurs when the uninterpreted predicate $resolve(n, \beta, pc)$ holds. At the time of resolution, branch predictor state $\beta'$ is updated using the uninterpreted function *update* and the speculation level $n'$ is decremented. As in SPECFENCE, nothing else need be done as the other state variables have the correct values at the decremented level.

Rule HAVOC will be described in § 4.5.4.

## 4.5 Formulating Secure Speculation

This section formulates the secure speculation property. First, we formalize an adversary model that captures arbitrary transient execution attacks. Next, we present the secure speculation property. Violations of this property correspond to transient execution vulnerabilities.

### 4.5.1 Formal Adversary Model

Recall system state $s$ is the tuple $\langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$[‖] and evolves according to the transition relation $\delta$, where the semantics are defined in Figure 4.11. As discussed in § 4.3, the system has an untrusted low-security component and a trusted high-security component that execute concurrently. Our verification objective is to prove that confidential states of a specified trusted program are indistinguishable to an arbitrary untrusted program. This verification task requires the definition of: (i) the trusted program to be verified and the family of untrusted adversary programs, (ii) confidential states of the trusted program, (iii) how the adversary *tampers* with system state, and (iv) what parts of state are adversary *observable*.

#### The Trusted and Untrusted Programs

We assume that the trusted program resides in the set of instruction memory addresses denoted by $\mathcal{T}_\rho$. The trusted program itself is defined by $\Pi[\rho]$ for each $\rho \in \mathcal{T}_\rho$. Every address $\rho \notin \mathcal{T}_\rho$ is part of the untrusted component and $\Pi[\rho]$ is unconstrained for these addresses to model all possible adversarial programs.

We assume that untrusted code can invoke trusted code only by jumping to a specific entrypoint address $\mathcal{EP} \in \mathcal{T}_\rho$. $\mathcal{T}_\rho$, $\mathcal{EP}$ and instructions $\Pi[\rho]$ for all $\rho \in \mathcal{T}_\rho$ are known to the adversary. Note that the adversary may speculatively attempt to invoke addresses other than the entrypoint, only the non-speculative invocations are restricted. Without such an assumption, the adversary may be able to jump past defensively placed instructions.

In the example shown in Figure 4.4a, $\mathcal{T}_\rho$ contains all instruction addresses that are part of the function `foo`. The entrypoint $\mathcal{EP}$ is the address of the first instruction in `foo`. Note that if the adversary can directly jump to line 6 (i.e., skip the fence on line 5), the program is vulnerable – this is why the restriction on invocation of only the entrypoint is required.

Such restrictions are implemented in all typical scenarios: system calls, software fault isolation, etc. To consider another example, if we are verifying secure speculation for system calls in an operating system kernel, $\mathcal{T}_\rho$ contains all kernel text addresses and the entrypoint $\mathcal{EP}$ is the syscall trap address.

#### Public States

Given the above definitions, the low operation executed in a state $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$ is defined by the projection of memory that is publicly visible and the low instruction being executed. Public memory is defined as the projection of memory down to the parts that are not in the set of public addresses $\mathcal{A}_\mathcal{P}$, i.e., $\mathcal{P}(\mu) \doteq \lambda a.\ \text{ITE}(a \in \mathcal{A}_\mathcal{P}, \mu[0, a], \bot)$. These regions of memory are adversary accessible. We define the low instruction as $inst_\mathcal{P}(s) \doteq s.\Pi[s.pc[0]]$ if $s.pc[0] \notin \mathcal{T}_\rho$ and $\bot$ otherwise. This definition refers to the non-speculative state – we are looking at $pc[0]$, not higher speculation levels. The instruction being speculatively executed

[‖]Note: We will use the notation *s.field* to refer to elements of the tuple.

may be different, and may in fact be from the trusted component. This is important because we use $op_{\mathtt{L}}$ to constrain adversary actions to be identical across traces, and these constraints can only refer to non-speculative state. Together, these define the low component $op_{\mathtt{L}}(s) \doteq \langle inst_{\mathcal{P}}(s), \mathcal{P}(\mu)\rangle$.

### Confidential States

The secret states that need to be protected from an adversary are the values stored in memory addresses $a$ that belong to the set $\mathcal{A}_{\mathcal{S}} \doteq \mathcal{A}/\mathcal{A}_{\mathcal{P}}$, where $\mathcal{A}$ refers to the set of all data memory addresses. For Figure 4.4, $\mathcal{A}_{\mathcal{S}}$ contains all addresses that are outside of the arrays `a1` and `a2`. Similar to public memory, confidential memory is defined as the projection of memory to the values at secret addresses $\mathcal{S}(\mu) \doteq \lambda a.\ \textsc{ite}(a \in \mathcal{A}_{\mathcal{S}}, \mu[0,a], \perp)$.

The high instruction executed in a state is denoted $inst_{\mathcal{S}}(s)$ and has the value $s.\Pi[s.pc[0]]$ when $pc[0] \in \mathcal{T}_{\rho}$ and $\perp$ otherwise. Intuitively, the high instruction is most recently non-speculative instruction being executed by the trusted program. The high operation executed in state $s$ is then defined as a tuple of the high instruction and the public memory: $op_{\mathtt{H}}(s) \doteq \langle inst_{\mathcal{S}}(s), \mathcal{S}(s.\mu)\rangle$. We include the values of secret memory in this tuple because the high-program may be non-deterministic and we need to constrain the non-determinism to be identical across certain traces.

### General Adversary Tampering ($\mathcal{G}$)

The adversary $\mathcal{G}$ tampers with system state by executing an unbounded number of operations from `Op` to modify architectural and microarchitectural state. Adversary tampering is constrained in only two ways.

1. (**Conformant Store Addresses**) For every non-speculative state in which an untrusted store is executed, the target address of the store must belong to the set of adversary-writeable addresses: $\mathcal{U}_{\mu}^{wr}$. We denote a trace $\pi$ where every state satisfies this condition by the predicate *conformantStoreAddrs*$(\pi)$, defined as follows.

$$\begin{aligned}
&\forall i.\ \pi^i.n = 0 \land \pi^i.pc[0] \notin \mathcal{T}_{\rho} \implies \\
&\quad \pi^i.\iota = \mathtt{mem} := \mathtt{mem}[e_1 \to e_2] \land \pi^i.\Delta[0, e_1] \Downarrow a \implies \\
&\quad a \in \mathcal{U}_{\mu}^{wr}
\end{aligned}$$

   Constraining adversary stores is necessary in order to prevent the adversary from changing the trusted program's architectural (non-speculative) state arbitrarily.

2. (**Conformant Entrypoints**) Non-speculative adversary jumps to trusted code must target the entrypoint $\mathcal{EP}$. A trace $\pi$ where every transition from untrusted to trusted

code satisfies this condition is denoted by the predicate *conformantEntrypoints*$(\pi)$. This is defined as follows:

$$\begin{aligned}
&\forall i, j.\ i < j \land \pi^i.n = \pi^j.n = 0 \implies \\
&\qquad (\forall k.\ i < k < j \implies \pi^k.n \neq 0) \implies \\
&\qquad \pi^i.pc[0] \notin \mathcal{T}_\rho \land \pi^j.pc[0] \in \mathcal{T}_\rho \implies \\
&\qquad \pi^j.pc[0] = \mathcal{EP}
\end{aligned}$$

The above constraints says that if $\pi^i$ and $\pi^j$ are non-speculative states, all states between $\pi^i$ and $\pi^j$ are speculative, and $\pi^i$ is part of the untrusted component while $\pi^j$ is part the trusted component, then $\pi^j$ must necessarily be at the entrypoint. Note this *does not* preclude speculative execution of "gadgets" in the trusted code that do not begin at the entrypoint.

The condition conformant store addresses captures the fact that the adversary cannot write to arbitrary memory locations. Conformant entrypoints ensures that execution of the trusted code starts at the entrypoint.

## Conformant Traces

A trace $\pi$ where: (i) $\pi^{(0)}$ is a non-speculative state and the trusted component has been initialized: $\pi^{(0)}.n = 0 \land init_{\mathcal{T}}(\pi^{(0)})$, (ii) every state $\pi^{(i)}$ satisfies the conformant stores condition and (iii) every pair of states $\pi^{(i)}$ and $\pi^{(j)}$, where $i < j$, satisfy the conformant entrypoints condition is called a conformant trace, denoted by *conformant*$(\pi)$.

$$\begin{aligned}
conformant(\pi) \doteq\ & \pi^{(0)}.n = 0 \land init_{\mathcal{T}}(\pi^{(0)}) \land \\
& conformantStoreAddrs(\pi) \land \\
& conformantEntrypoints(\pi)
\end{aligned} \tag{4.4}$$

## Adversary Observations

We model an adversary who can observe all architectural state and most microarchitectural state when executing; i.e. when $n = 0$ and $pc[0] \notin \mathcal{T}_\rho$. Specifically, the adversary can observe the following:

1. non-speculative register values: $\Delta[0, r]$ for all $r$.
2. non-speculative values stored at all memory addresses in the set $\mathcal{U}_\mu^{rd}$: $\mu[0, a]$ for all $a \in \mathcal{U}_\mu^{rd}$.
3. the trace of instruction and data memory accesses: $\omega$.

4. the branch predictor state $\beta$.

The above implies that two states $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$ and $s' = \langle \Pi, \Delta', \mu', pc', \omega', \beta', n', \iota' \rangle$ are low-equivalent, denoted $s \approx_{\mathtt{L}} s'$, iff $(n = 0 \wedge pc[0] \notin \mathcal{T}_\rho) \implies (\forall r.\ \Delta[0, r] = \Delta'[0, r]) \wedge (\forall a.\ a \in \mathcal{U}^{rd}_\mu \implies \mu[0, a] = \mu'[0, a]) \wedge \omega = \omega' \wedge \beta = \beta'$.

We do not allow the adversary to observe $\Delta[n, r]$ and $\mu[n, a]$ for $n > 0$ because there is no way to "output" speculative state except through a microarchitectural side-channel. These side-channels are captured by the trace of memory accesses $\omega$ which models leaks via caches, prefetches, DRAM and well as other structures in the memory subsystem. The branch predictor state $\beta$ captures all leaks caused by the branch predictor side-channel. Note that the adversary *can* observe the non-speculative values stored in memory for the addresses in the range $\mathcal{U}^{rd}_\mu$, and non-speculative values of the registers when adversary code is being executed.

## 4.5.2 Formalization of the Security Property

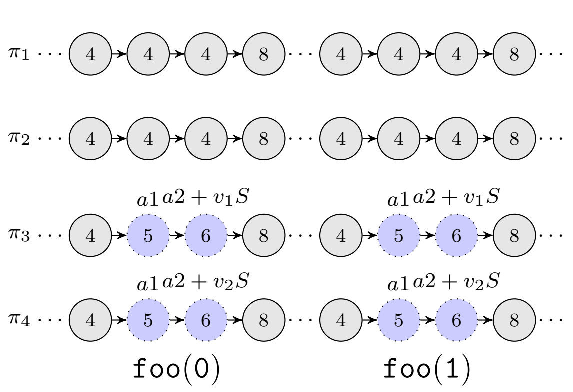
$$\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4. \\
&\quad \mathit{conformant}(\pi_1) \wedge \mathit{conformant}(\pi_2) \implies \\
&\quad \mathit{conformant}(\pi_3) \wedge \mathit{conformant}(\pi_4) \implies \\
&\quad \forall i.\ \neg \mathit{mispred}(\pi_1^i.n, \pi_1^i.\beta, \pi_1^i.pc) \implies \\
&\quad \forall i.\ \neg \mathit{mispred}(\pi_2^i.n, \pi_2^i.\beta, \pi_2^i.pc) \implies \\
&\quad \exists i.\ \mathit{mispred}(\pi_3^i.n, \pi_3^i.\beta, \pi_3^i.pc) \implies \\
&\quad \exists i.\ \mathit{mispred}(\pi_4^i.n, \pi_4^i.\beta, \pi_4^i.pc) \implies \\
&\quad op_{\mathtt{L}}(\pi_1) = op_{\mathtt{L}}(\pi_2) = op_{\mathtt{L}}(\pi_3) = op_{\mathtt{L}}(\pi_4) \implies \\
&\quad op_{\mathtt{H}}(\pi_1) = op_{\mathtt{H}}(\pi_3) \wedge op_{\mathtt{H}}(\pi_2) = op_{\mathtt{H}}(\pi_4) \implies \\
&\quad \pi_1 \approx_{\mathtt{L}} \pi_2 \wedge \pi_3^0 \approx_{\mathtt{L}} \pi_4^0 \implies \\
&\quad \pi_3 \approx_{\mathtt{L}} \pi_4
\end{aligned} \tag{4.5}$$

Using the above definitions, we are now ready to formalize the secure speculation property, shown in Equation 4.5.

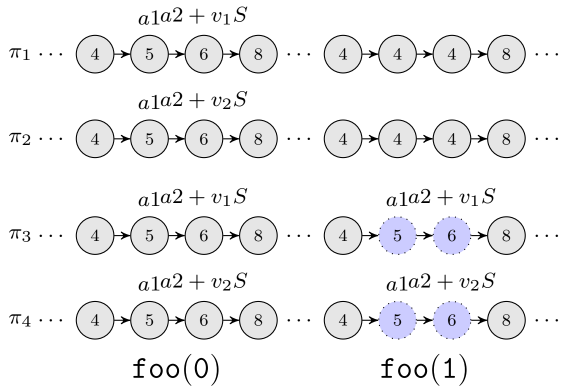
This is an instantiation of the TPOD property shown in Equation 4.3. The trace property $T$ is satisfied when no misspeculation occurs: $\pi \in T \iff \forall i.\ \neg \mathit{mispred}(\pi^i.n, \pi^i.\beta, \pi^i.pc)$.[**] The trace property $U$ requires that all traces be conformant as defined in Equation 4.4. This ensures we only search for violations among traces representing valid executions of our system/adversary model.

A violation of Equation 4.5 occurs when there exists a sequence of adversary instructions such that traces $\pi_1$ and $\pi_2$ are low-equivalent, but $\pi_3$ and $\pi_4$ are not low-equivalent. In other words, we have an information leak that only occurs on a speculative processor; i.e. a transient execution vulnerability.

[**]Or equivalently in linear temporal logic: $\pi \models \Box \neg \mathit{mispred}$.



(a) Violation of secure speculation when N=0 in Figure 4.5.



(b) Secure speculation satisfied when N=1 in Figure 4.5.

Figure 4.12: Illustrating the secure speculation property for the code in Figure 4.5. The numbers within each state refer to program counter values (shown as line numbers from the figure). A label above each state indicates the data memory address accessed by that instruction (if any). States shown in dotted circles are specuative states. Note the non-speculative traces "stutter" when the other traces are speculating. The values $v_1$ and $v_2$ refer to the contents of memory address $a_1$ in their respective traces. Note that traces $\pi_1$ and $\pi_2$ do not speculate while traces $\pi_3$ and $\pi_4$ do.

### 4.5.3 Illustrating Violation/Satisfaction of Secure Speculation

Let us consider the conditionally vulnerable Spectre variant shown in Figure 4.5. A quadruple of traces for this program is shown in Figure 4.12a. Two calls to `foo` are made with arguments $\texttt{i} = 0$ and $\texttt{i} = 1$. The two non-speculative traces $\pi_1$ and $\pi_2$ do not execute the if statement and so they have the same adversary observations (i.e., are low-equivalent). However, traces $\pi_3$ and $\pi_4$ speculatively execute the if statement and the adversary can observe differences in the memory addresses corresponding to the second array access: $a2[v_1 * S]$ and $a2[v_2 * S]$. All traces have the same adversary operations with one pair low-equivalent and non-speculative while the other pair is not low-equivalent and speculative. This is a violation of secure speculation.

Now consider the scenario when $N > 0$, say $N = 1$. This is shown in Figure 4.12b. The key difference here is that the non-speculative traces also make the second array access when $\texttt{i} = 0$. The second memory access reads from the addresses $a2 + v_1 * S$ and $a2 + v_2 * S$ in traces $\pi_1$ and $\pi_2$ respectively. There are two scenarios possible. Either $v_1 = v_2$ or $v_1 \neq v_2$. Suppose $v_1 = v_2$, then traces $\pi_1$ and $\pi_2$ are low-equivalent, but so are traces $\pi_3$ and $\pi_4$! Conversely, if $v_1 \neq v_2$, then the $\pi_1$ and $\pi_2$ are not low-equivalent and the secure speculation property holds vacuously.

### 4.5.4 Adversary Reduction Lemma

The general adversary's tampering described in § 4.5.1 allows the adversary to execute an unbounded number of arbitrary instructions. While this is fully general, directly modeling this makes formal verification infeasible. In addition, finding inductive invariants to characterize these changes is also not feasible in general for an automated approach. To address this problem, we introduce a simpler "havocing adversary" $\mathcal{H}$ and prove that this adversary is as powerful as the general adversary $\mathcal{G}$. We denote the platform model executing with the general adversary as $M^{\mathcal{G}} = \langle Q, I, \delta^{\mathcal{G}}, \mathsf{Op}\rangle$ and the platform model executing with the havocing adversary as $M^{\mathcal{H}} = \langle Q, I, \delta^{\mathcal{H}}, \mathsf{Op}^{\mathcal{H}}\rangle$. The difference between these two models is that the former model executes operations from $\mathsf{Op}$ and the latter executes operations from $\mathsf{Op}^{\mathcal{H}} \doteq \mathsf{Op} \cup \{\textsc{Havoc}\}$. To indicate a transition from state $p \in Q$ to $q \in Q$ in $M^{\mathcal{G}}$, we write $\delta^{\mathcal{G}}(p, \mathsf{op}, q)$. We similarly define $\delta^{\mathcal{H}}(p, \mathsf{op}, q)$ for $M^{\mathcal{H}}$. The goal is to simplify $M^{\mathcal{G}}$ using $M^{\mathcal{H}}$ by introducing an additional operation Havoc for the adversary, which abstracts away the unbounded execution of $\mathcal{G}$.

$\mathcal{H}$ executes only one instruction that modifies non-speculative state, $\texttt{havoc}\ (\Delta, \texttt{mem}[\mathcal{U}_{\mu}^{wr}], \beta)$. The semantics of this instruction are shown in Figure 4.11; it sets the registers, program counter, data at adversary writable memory addresses, trace of observations, and branch predictor to unconstrained values (i.e. "havocs" them). The set of writable memory addresses is denoted by $\mathcal{U}_{\mu}^{wr}$ which is a subset of the set of all addresses, denoted as $\mathbb{A}$. The speculative platform model executing with the $\mathcal{H}$ is defined with the transition relation $\delta^{\mathcal{H}} \subset Q \times \mathsf{Op}^{\mathcal{H}} \times Q$. In other words, the adversary has access to an additional – and more abstract – Havoc operation.

We begin by proving a simpler result, that the execution of an operation in Figure 4.11 can be *simulated* by the Havoc operation. In other words, each operation $\mathsf{op} \in \mathsf{Op}$ executed by the adversary $\mathcal{G}$ in $M^{\mathcal{G}}$ can be replaced by the more abstract operation Havoc. We later use this result to prove that any number of operations can be simulated by a single Havoc operation and consequently, that $M^{\mathcal{G}}$ can be simulated by $M^{\mathcal{H}}$.

**Lemma 4.2.** *A non-speculative adversary execution in $M^{\mathcal{G}}(\mathsf{op})$ where $\mathsf{op} \in \mathsf{Op}$, can be simulated by an adversary execution in $M^{\mathcal{H}}(\textsc{Havoc})$. More specifically, for all reachable states $p \in Q$ of $M^{\mathcal{G}}$, if $n = 0$, $inst_{\mathcal{P}}(q) \neq \bot$ and $\delta^{\mathcal{G}}(p, \mathsf{op}, q)$, then there exists some $q'$ such that $\delta^{\mathcal{H}}(p, \mathsf{op}, q')$ and $q = q'$.*

*Proof.* Assume that $n = 0$ and $\delta^{\mathcal{G}}(p, \mathsf{op}, q)$ and let $q'$ be the post state after executing Havoc, i.e., $\delta^{\mathcal{H}}(p, \textsc{Havoc}, q')$. We also assume that during the adversary execution of $\mathsf{op}$, the only writable and readable memory locations are $a \in \mathcal{U}_{\mu}^{wr}$. Note that since we assume $n = 0$, we can ignore the operations T-Mispred, NT-Mispred, and Resolve, as the premises in the operations' respective judgments require that $n' \neq 0$. Define $\mathcal{Q}_{\mathcal{A}}^{\mathsf{op}} : (p, v) \mapsto \{q.v \mid \delta^{\mathcal{A}}(p, \mathsf{op}, q)\}$ as the set of values variable $v$ can assume after adversary $\mathcal{A}$ executes operation $\mathsf{op}$ in state $p$. To prove simulation, it suffices to show that $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}}(p, v) \subset \mathcal{Q}_{\mathcal{H}}^{\textsc{Havoc}}(p, v)$ for all variable $v \in V$, operation $\mathsf{op} \in \mathsf{Op}$ and reachable state $p$. Since Havoc allows the variables $\Delta, \omega, pc$ and $\beta$ to be any value (i.e., $q'.v \in \mathbb{D}(v), \forall v \in \{\Delta, \omega, pc, \beta\}$), then $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}}(p, q, v) \subset \mathbb{D}(v) = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, v), \forall v \in \{\Delta, \omega, pc, \beta\}$. Thus, it remains to show that $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}}(p, v) \subset \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, v)$ for $v \in \{\mu, \iota\}$. Since the execution of Store is only to the set $\mathcal{U}_{\mu}^{wr}$ during adversary execution, then $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}}(p, \mu) = \mathcal{Q}_{\mathcal{G}}^{\text{Store}}(p, \mu) \subset \{\mu[(0, a) \to c] \mid a \in \mathcal{U}_{\mu}^{wr}, c \in T(\mu[a])\} \subset \{\mu' \mid \forall a \in \mathbb{A}.\ a \notin \mathcal{U}_{\mu}^{wr} \Rightarrow \mu'[0, a] = \mu[0, a], \forall m \in \mathbb{N}, a \in \mathbb{A}.\ m > 0 \Rightarrow \mu'[m, a] = \mu[m, a]\} = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, \mu)$. In other words, the execution of Store only sets one memory location $\mu[0, a]$ inside the adversary space to a new value and Havoc unconstrains all values in the adversary space. Lastly, all operations assign $\iota$ to $\mu[pc'[0]]$, so we have $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}}(p, \iota) = \{\Pi[pc[n]] \mid pc[n] \in \mathbb{A}\} = \{\Pi[pc[0]] \mid pc[0] \in \mathbb{A}\} = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, \iota)$. □

**Lemma 4.3.** *Every sequence of states $q_i, \ldots, q_k$ with $inst_{\mathcal{P}}(q_j) \neq \bot$ and $q_j.n = 0$ for every $i \leq j \leq k$ can be simulated by a single* `havoc` $(\Delta, \mathtt{mem}[\mathcal{U}_{\mu}^{wr}], \beta)$ *instruction. That is, $\mathcal{G}$ can be simulated by $\mathcal{H}$.*

*Proof.* Lemma 4.2 states that each operation $\mathsf{op} \in \mathsf{Op}$ can be simulated by Havoc, thus it suffices to show that a sequence of Havoc operations can be simulated by a single Havoc operation. By transitivity, this implies Lemma 4.3.

Let $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}}, \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}$ be defined as in the proof of Lemma 4.2 and define the set of values of a variable $v$ after executing $k - i$ operations as $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}_i, \ldots, \mathsf{op}_{k-1}} : (p, v) \mapsto \{q_k.v \mid \exists q_i, \ldots, q_k \in Q.\ p = q_i \wedge \bigwedge_{j=i}^{j=k-1} \delta^{\mathcal{G}}(q_j, \mathsf{op}_j, q_{j+1})\}$. We show that $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}_i, \ldots, \mathsf{op}_{k-1}}(p, v) \subset \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, v)$ for all $v \in V$. By assumption, the platform is not speculating and so $\forall j \in \{i, \ldots, k\}.\ q_j.n = 0$. Thus, $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}_i, \ldots, \mathsf{op}_{k-1}}(p, n) = \{0\} = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, v)$. For all $v \in \{\Delta, \omega, pc, \beta\}$, we know that $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}_i, \ldots, \mathsf{op}_{k-1}}(p, v) = \{q_k.v \mid q_k.v \in \mathbb{D}(v)\} = \mathbb{D}(v) = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, v)$. For $\mu$, realize that the conjunction of constraints imposed by the premises related to $\mu$ is identical to a single constraint. In other words, if we let $P(\mu) \doteq \big((\forall a \in \mathbb{A}.\ a \notin \mathcal{U}_{\mu}^{wr} \implies \mu'[0, a] := \mu[0, a] \wedge (\forall m \in \mathbb{N}, a \in \mathbb{A}.\ m > 0 \Rightarrow \mu'[m, a] = \mu[m, a])\big)$, the final constraint imposed on $\mu$ gives the set of values $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}_i, \ldots, \mathsf{op}_{k-1}}(p, \mu) = \{q_k.\mu \mid P(q_i.\mu) \wedge \ldots \wedge P(q_k.\mu)\} = \{p.\mu \mid P(p.\mu)\} = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, \mu)$. Lastly, $\mathcal{Q}_{\mathcal{G}}^{\mathsf{op}_i, \ldots, \mathsf{op}_{k-1}}(p, \iota) = \{\Pi[pc[0]] \mid pc[0] \in \mathbb{A}\} = \mathcal{Q}_{\mathcal{H}}^{\text{Havoc}}(p, \iota)$. □

The adversary reduction Lemma 4.2 lets us replace all sequences of non-speculative instructions executed by the adversary with `havoc`'s and helps scale verification. It is important to note that we cannot replace instruction sequences that contain speculative instructions because these may contain exploitable transient execution gadgets.

### 4.5.5 Discussion and Limitations

An important implication of the secure speculation property is that if a program satisfies Equation 4.5, then all observational determinism properties where low-equivalence is defined over $\omega$, $\mu$ and $\beta$ that hold for non-speculative execution of the program also hold for speculative executions. For instance, a tool like CacheAudit [61, 62] can be used to verify that the cache accesses of a program are independent of some secret. Note that even though a program's non-speculative execution may not leak information through cache (this is what CacheAudit verifies) that *does not mean* that its speculation execution will have the same properties. This is because CacheAudit does not model speculative execution. However, if we do prove Equation 4.5 for a program, then all properties proven by tools like CacheAudit also apply to the program's speculative execution.

Our operational semantics are for in-order processors only. Nevertheless, the secure speculation property can be used to analyze out-of-order execution and other speculation (e.g., memory address disambiguation) in a conceptually straightforward way by extending the semantics to model these features.

Specific programs may need additional constraints on the traces to avoid spurious counterexamples, especially if the set of secrets $\mathcal{S}_\mathcal{T}$ is over-specified. For example, in Figure 4.4(b), a tuple of traces where the $\texttt{i} < \texttt{N}$ never occurs would cause a violation of Equation 4.5 if $\mathcal{S}_\mathcal{T}$ also contained the addresses that point to `a1` and `a2`.

## 4.6 Verification Approach

We have implemented an automated verifier to answer the following question: Given a program (e.g., C code) as input, does it satisfy the secure speculation property in Equation 4.5?

Our approach is fairly standard, based on the method of *self-composition* (see, e.g., [21]). For lack of space, we present only the essential aspects. Given the input program, we translate it into a transition system based on the adversary model and operational semantics presented in the previous section. The secure speculation property is a 4-safety property, meaning that we can turn it into a safety property to be checked on a 4-way self-composition of the transition system. We use term-level model checking [35] based on satisfiability modulo theories (SMT) solving to check whether the safety property holds for this 4-way self-composition. The model checker uses either bounded model checking (to find violations of the property) or k-induction (to prove the property).

The main new aspect of our verifier is the implementation of the transformation of the program into a transition system. We rely on two tools: the Binary Analysis Platform (BAP) [34] to translate x86 binaries into an intermediate format called BIL, and UCLID5 [193], an SMT-based model checking tool supporting both bounded model checking (BMC) and k-induction. BIL is an assembly-like intermediate language similar to AIR (described in Sec. 4.4). Overall, our workflow for each input program is as follows:

1. Compile C source code containing the victim function into an x86 binary file.
2. Translate the x86 binary file using BAP into the BIL intermediate language.
3. Translate the BIL into UCLID5 models and check the secure speculation property via self-composition. For each program, we first obtain a counterexample via BMC

demonstrating the vulnerability; then, we insert an lfence at an appropriate point and prove the secure speculation property via k-induction.

We note that this workflow may be abstracted to a more general TPOD property.

The translation from BIL to UCLID5 implements the operational semantics given earlier, with the following key steps:

1. Datatypes in the BIL program such as addresses, memories, and words are converted to uninterpreted types for more scalable analysis and to obtain a more portable model that is not specific to 32-bit/64-bit architectures.

2. Each basic block of the BIL program is considered an atomic step of the transition system in the UCLID5 model after which the safety property is checked on the 4-way self-composition. This suffices as the deviations in behavior between the 4 traces happen at branch points.

3. At any speculative transition step, the program can resolve a misspeculation as per the RESOLVE rule.

4. All state variables are initialized to symbolic constants with the exception of the memory, where it is initialized to have the same value at every address except the program-specific secret address that stores the secret.

Given the model, the implication chain of the secure speculation property is translated into a number of assumptions and invariants. The invariants which we wish to check are whether the speculative program traces diverge in control flow, branch prediction or memory access observations, but only in the cases that they do not for the non-speculative traces. Proofs by induction require a few additional auxiliary invariants, whereas bounded model checking does not.

For a particular proof of the secure speculation property on one of the examples from Section 4.7 in UCLID5, we first instantiate four programs as instances. In UCLID5, this is a composition of four transition systems within the main proof script. Next, we define a speculation flag that determines if a program is allowed to speculate or not and instantiate the program pairs `t1`, `t2` and `t3`, `t4` with their speculation flags turned off and on respectively. For initialization, we set the program counters, registers, and rollback states of the programs to be the same. We also set the memories of the program pairs `t1`,`t3` and `t2`, `t4` to be the same except for a confidential memory address represented by a symbolic constant, which corresponds to the conformant conditions. Note that only allowing one (symbolic) memory address to differ is sufficient because any counterexample with a larger difference can be extended to one in the single address case. Additionally, we make the assumption that the program counter and observational states, such as prior memory read addresses and the branch predictor state, are initially equal across the non-speculating programs `t1` and `t2`. During a transition step of the main proof, we step both of the non-speculative traces `t1` and `t2` only if `t3` and `t4` are not currently speculating and they "stutter" otherwise.

This is to prevent any spurious counter-examples from divergent observational states caused by mismatching steps in the non-speculating and speculating programs. At each of these program transitions, a basic block of the program is executed. This abstraction is sound in the sense that the traces considered are a strict subset of the permissable traces of an out of order processor and if out of order execution is the cause of the observational determinism violation, then it should not be captured in our property. The secure speculation property is then encoded as equality across the program counter and observational states in the speculating programs `t3` and `t4`. This property was proven inductively and through bounded model checking in all of our examples. For inductive invariant checking, various auxiliary invariants were required to constrain the attacker input, rollback states, entry points, memories, and speculating states of the programs.

As a remark on the formulation of the 4-safety property, traces `t1` and `t2` are used to constrain which addresses are public and private. This allows us to generalize over the various examples for variant 1 of the spectre attack instead of encoding this specifically for each program example. More concretely, in the $N > 0$ case of Figure 4.4c, this constrains the value at the confidential memory address to be the same.

## 4.7 Case Studies

We used our verifier for a proof-of-concept demonstration to detect whether or not a snippet of C code is vulnerable to the Spectre class of attacks. As benchmarks, we rely on Paul Kocher's list of 15 victim functions vulnerable to the Spectre attack [115] in addition to the examples we presented earlier.

In particular, we show here the results on examples 1, 5, 7, 8, 10, 11, and 15 from Paul Kocher's list, along with the example from Figure 4.4 (c), and an example with nested if statements. We chose these based on what we believe are illustrative of a wide range of victim functions that are not easily detectable using the current static analysis tools such as Qspectre [147], which was only able to detect the first two examples in Kocher's list. In each example, we also add a fence at an appropriate location to illustrate how one can mitigate the attack. We begin with a brief explanation of some of the examples and then discuss the results from applying bounded model checking and induction with our secure speculation property on our UCLID5 models.

### Bounds-Check-Bypass Variations

*Example 1 (Figure 4.13)*: The first example of Paul Kocher's list is the original bounds check bypass variant of the Spectre attack discovered [116].

```
void victim_fn_v01(unsigned x) {
  if (x < array1_size) {
    __mm_lfence();
    temp &= array2[array1[x] * 512];
  }
}
```

Figure 4.13: Example 1: Original Spectre BCB (bounds check bypass) example.

*Example 5 (Figure 4.14)*: This example is similar to the first variant but implemented within a for loop. The untrusted argument `x` may be larger than the array size, which causes the vulnerability, but if `x` is within the bounds of the array, note that condition `i > 0` is also potentially vulnerable to the attack.[††] Inserting a fence at line 4 prevents the subsequent memory accesses at each iteration of the loop and mitigates the attack. Alternatively, inserting the fence between lines 2 and 3 will also mitigate the attack.

```
1 void victim_fn_v05(unsigned x) {
2   if (x < array1_size)
3     for (size_t i = x - 1; i > 0; i--)
    {
4      _mm_lfence();
5      temp &= array2[array1[i] * 512];
6     }
7 }
```

Figure 4.14: Example 5: BCB with a for loop.

*Example 7 (Figure 4.15)*: This example is interesting because it depends on the value of a static variable updated from a previous call of the function `victim_function_v07`. Every call to the function should not make the second array access at line 5 unless `x == last_x`. An attacker can exploit this function by first calling with `x < array1_size` repeatedly to train the predictor to predict true at line 3, and then subsequently make a call with an arbitrary value of `x`. This would then trigger the execution of the second array access. Inserting a fence at line 4 requires the condition at line 3 to be evaluated and thus prevents speculatively executing the second array access.

```
1 void victim_fn_v07(unsigned x) {
2   static unsigned last_x = 0;
3   if (x == last_x) {
4     _mm_lfence();
5     temp &= array2[array1[x] * 512];
6   }
7   if (x < array1_size)
8     last_x = x;
9 }
```

Figure 4.15: Example 7: BCB with unsafe static variable check.

*Example 8 (Figure 4.16)*: The ternary operator is interesting because the program counter is allowed to jump to two different basic blocks for the computation of the second array memory access as opposed to one block as in Example 1. Eventually, it returns to the basic block represented by line 5. The placement of the fence is also right before the second array access, similar to Figure 4.13.

```
1 void victim_fn_v08(unsigned x) {
2   result = (x < array1_size);
3   unsigned ind = result ? (x + 1) : 0;
4   _mm_lfence();
5   temp &= array2[array1[ind] * 512];
6 }
```

Figure 4.16: Example 8: BCB with the ternary conditional operator.

[††]Kocher's code has the condition `x >= 0` which causes an infinite loop.

*Example 10 (Figure 4.17)*: This is the first example where a second load dependent on a secret is not required for a leak. This is because whether `array2[0]` is cached is dependent on the result of `array1[x] == k`. To accomplish determine the value of `array1[x]`, the attacker can repeatedly call the function and iterate over the values of `k`. Thus, knowing whether or not `array2[0]` was accessed is enough to leak the secret at `array1[x]`.

```
void victim_fn_v10(
    unsigned x, unsigned k
) {
  if (x < array1_size) {
    __mm_lfence();
    if (array1[x] == k)
      temp &= array2[0];
  }
}
```

Figure 4.17: Example 10: BCB using an additional attacker-controlled input.

*Example 11 (Figure 4.18)*: Ideally, an approach to checking secure speculation should also be able to capture leakage through various function calls. This example uses a function call to `memcpy` to leak the secret. However, because of the single byte access, it gets optimized into a single load and store.

```
void victim_fn_v11(unsigned x) {
  if (x < array1_size) {
    _mm_lfence();
    temp = memcmp(&temp, array2 + (
   array1[x] * 512), 1);
  }
}
```

Figure 4.18: Example 11: BCB using the memory comparison function.

*Example 15 (Figure 4.19)*: This example is interesting because it passes a pointer instead of an integer as the attacker-controlled input. We assume the value stored in the pointer is constant across traces to ignore cases where the attacker forces a secret dependent branch during non-speculative execution.

```
void victim_fn_v15(unsigned *x) {
  if (*x < array1_size) {
    _mm_lfence();
    temp &= array2[array1[*x] * 512];
  }
}
```

Figure 4.19: Example 15: BCB using attacker-controlled pointer.

*Example NI (Figure 4.20)* In this example, nested if statements cause the attack to occur even without a second address load dependent on a secret. If the programs speculatively choose not to execute the second if statement, the value `array2[0]` will not be cached, but if a program eventually executes the second if statement as a result of a resolution, then a leak can occur as determined by the cached `array2[0]` value.

```
void victim_fn_nested_ifs(unsigned x) {
  unsigned val1, val2;
  if (x < array1_size) {
    val1 = array1[x];
    if (val1 & 1) {
      _mm_lfence();
      val2 = array2[0];
    }
  }
}
```

Figure 4.20: Example NI: BCB with nested if statements.

| | ex1 | ex5 | ex7 | ex8 | ex10 | ex11 | ex15 | Fig. 4.5 | NI |
|---|---|---|---|---|---|---|---|---|---|
| BMC | 6.6 | 9.0 | 10.2 | 5.7 | 9.6 | 6.4 | 5.8 | 6.6 | 12.9 |
| Ind | 5.0 | 5.0 | 5.7 | 4.6 | 5.8 | 5.9 | 4.8 | 4.8 | 5.4 |

Table 4.1: Runtime (sec.) of each example using 5 steps for bounded model checking to find vulnerabilities and 1 step induction to prove correctness after inserting a memory fence. These experiments were run on a machine with an 2.20GHz Intel(R) Core(TM) i7-2670QM CPU with 5737MiB of RAM.

**Runtime Results**

Table 4.1 lists the runtime (in seconds) required for each verification task with the memory fences implemented. As can be seen, the verifier is able to prove the correctness of these programs within a few seconds. Although these programs are small, this exercise gives us confidence that the method could be useful on larger programs. We assert that with the use of a stronger software model-checking engine and the development of TPOD-specific abstractions, it will be possible to prove secure speculation for larger programs.

## 4.8 Related Work

The most closely related past work to ours is CheckMate [209] which uses happens-before graphs to analyze transient execution vulnerabilities. The insight in CheckMate is that happens-before graphs encode information about the orders in which instructions can be executed. By searching for patterns in the graph where branches are followed by dependent loads, an architectural model can be analyzed for susceptibility to Spectre/Meltdown. A key difference between CheckMate and our approach is that we are not matching patterns of vulnerable instructions. Our verification is semantic, not pattern-based. In particular, the example showing conditional vulnerability in Figure 4.4(c) cannot be precisely captured by CheckMate.

Another closely related effort is by McIlroy et al. [140] who introduce a formal model of speculative execution in modern processors and analyze it for transient execution vulnerabilities. Similar to our work, they too introduce speculative operational semantics and their model includes indirect jumps and a timer. An important difference between their semantics and ours is that their semantics are based on a microarchitectural model of execution. In contrast, our semantics capture an abstract notion of speculation that: (i) does not prescribe any specific microarchitectural implementation and (ii) is more amenable to verification due to its abstract nature. Further, they do not present an automated verification approach for finding transient execution vulnerabilities.

In concurrent work to ours, Guarnieri et al. [82] introduce SPECTECTOR which is also a principled verification methodology for the detection of Spectre-like vulnerabilities. They introduce the notion of speculative non-interference which is defined as follows: for every pair of initial configurations of the program, if these configurations are low-equivalent and their non-speculative traces have the same observations, then their speculative traces must also have the same observations. This is similar to our secure speculation property. Note that we also introduce TPOD which is a generalization of secure speculation/speculative non-interference and could be used to reason about the interaction between arbitrary microarchitectural side-channels: e.g. prefetching and value prediction.

One difference between our work and SPECTECTOR is that the latter only considers terminating programs while our methodology is applicable to non-terminating programs. This is because SPECTECTOR analyzes only finite-length traces. This also implies that in the case of non-terminating programs, SPECTECTOR can only find violations, not prove the absence of vulnerabilities. In contrast, our verification methodology can indeed prove the absence of vulnerabilities. A important insight in the SPECTECTOR work is that is that non-speculative traces are sub-sequences of the speculative traces *for in-order processors*. The allows SPECTECTOR to convert the 4-safety secure speculation property into into a 2-safety property. While this is an important and useful optimization that improves scalability, it does not appear to be applicable to out-of-order speculative semantics. While our current implementation and semantics do not model out-of-order execution, they are built to be extensible to this scenario.

The Spectre vulnerability was discovered by Kocher et al. [116] while Meltdown was discovered by Lipp et al. [133]. Their public disclosure has triggered an avalanche of new transient execution vulnerabilities, notable among which are Foreshadow [215] which attacked enclave platforms and virtual machine monitors, SpectreRSB [117] and Ret2Spec [137]. A thorough study of transient execution vulnerabilities was done by Canella et al. [39]. These vulnerabilities build on the rich literature of microarchitectural side-channel attacks [2, 79, 80, 105, 128, 134, 162, 165, 166, 191, 197]. Verification of mitigations to these "traditional" side-channel attacks is well-studied [4, 5, 6, 19, 22, 26, 61, 62, 65, 66, 173].

TPOD in general and secure speculation in particular are examples of hyperproperties [51]. A large body of work has studied hyperproperties that encode secure information flow. Influential exemplars of this line of work include noninterference [76], separability [180] and observational determinism [142, 178, 233]. Our verification method is based on self-composition which has been well-studied; see, for example, Barthe et al. [20, 21]. While we take a straightforward approach to using self-composition, more sophisticated approaches are also possible in some cases (e.g., [201]).

## 4.9 Summary

This chapter presented a formal approach for secure speculative execution on modern processors, a key part of which is a formal specification of secure speculation that abstracts away from the particulars of specific vulnerabilities. Our secure speculation formulation is an instance of trace property-dependent observational determinism, a new class of information flow security properties introduced by this work. We introduced an adversary model and an automated approach to verifying secure speculation and demonstrated the approach on several programs that have been used to illustrate the Spectre class of vulnerabilities. To the best of our knowledge, the material from this chapter presents the first effort to formalize and automatically prove secure speculation.

# Chapter 5

# Compositional Proofs of Information Flow Properties for Hardware-Software Platforms

Chapter §4 presented a methodology to prove secure speculation of programs for select variants of speculative attacks. This means that the speculative platform model only contains components relevant to those attacks. It does not explore the malleability of the approach to extend to classes of hardware attacks. Related existing work [68, 82, 83] is similar in that regard. Thus, it is not clear how one would extend the platform model beyond the specific abstraction and platform semantics introduced in prior work. Moreover, existing approaches have not developed an approach that prevents the path explosion problem to which instruction-level program models and microprocessor models are prone.

This chapter describes a proof system for compositional information flow analysis and formalism for compositional models to tackle this problem. More specifically, we focus on exploiting taint- and symbolic-based representations of information flow in our compositional methods.

## 5.1 Introduction

The rise of user-friendly, high-level hardware design frameworks [14, 92, 118] has made agile development and optimization of hardware computation platforms more accessible. These designs range from general-purpose computers and domain-specific computation engines (accelerators), to platforms designed with security as the guiding principle [10, 52, 69, 70, 101, 111, 112, 127]. The usage of ever more efficient hardware systems, however, has been plagued by the existence of hardware execution attacks [29, 40, 41, 116, 133, 186, 215, 216, 224]. Formal methods can provide strong security guarantees about system behaviour in the context of these attacks, thus building trust in the system.

Most hardware execution attacks exploit *microarchitectural* features such as caches, branch predictors, and load/store buffers. While reasoning over the architectural state (e.g., program counter, registers) suffices for proving functional correctness of software, one also has to account for the microarchitectural state when proving security properties. Since the microarchitectural state is more detailed than the architectural state, software semantics at the microarchitectural result in especially challenging verification queries. This problem is made more severe given that vulnerable software fragments are typically nested in large pieces of code. In this work, we develop a concerted approach to make verification of information-flow-style properties scale. On the software side, our approach uses Hoare-style reasoning tailored to security properties, while on the hardware side we leverage compositionality of the platform model.

Fundamental contributions such as Hoare-logic [85], interpolants [136] have enabled techniques such as interpolation-based reasoning [144], and invariant inference (e.g. [48, 60]) which have improved the scalability of software verification. These techniques have been predominantly used for checking single-trace properties such as safety and functional correctness. Security properties such as non-interference on the other hand are *hyperproperties* [51] defined over sets of traces. While some hyperproperties can be compiled to single-trace safety properties (over the self-composition), and hence permit the above approaches, such encodings do not make use of the specialized nature of security properties. This informs our first research question:

**(RQ1)** How can we tailor Hoare-style proof techniques to better scale verification of information-flow-based properties such as non-interference?

We answer this question by developing a proof system called ***SymboTaint***, which combines the symbolic representation of state, with taint-like equivalences over system variables. These equivalences align with security properties such as non-interference which prescribe equivalence between two executions w.r.t. certain variables. The symbolic state allows for more precise reasoning than pure taint analysis.

Security verification of software is closely tied with the microarchitectural semantics of the underlying hardware. Additionally, the capabilities of an adversary also vary with the microarchitecture, as some microarchitectural features create new side channels [205], leading to modified security specifications. While microarchitectural semantics are much more detailed, verification can benefit from frameworks that leverage compositionality of hardware. The choice of modelling framework also impacts whether one is able to easily instrument the model with proofs. Hence, we ask:

**(RQ2)** What modeling formalisms allow compositionality and parameterizability, and connect better with the software-side proof techniques?

We answer this question by developing an abstract operational model called the ***Information Flow State Machine*** (IFSM). Intuitively, an information flow state machine augments the underlying platform model with the joint symbolic-taint analysis from the proof system. This enables better interoperability between the proof and the model. Additionally, under some conditions, the transition relation of an IFSM can be decomposed. This allows the proof to reason about only those components of the platform that are relevant to the security property.

To evaluate the efficacy of our approach, we introduce a speculative platform model and verify the security of several safe and vulnerable programs. These are representative of a broad class of transient execution attacks [40, 89] targeting various microarchitectural features. Our verification approach is based on IFSMs which instrument the speculative platform with the proof. We compare the performance of our technique with prior work on verifying transient execution attacks [46] and observe improved performance across safe and unsafe examples. In summary, we make the following main contributions in this chapter:

1. **SymboTaint Proof System**. We introduce SymboTaint, a sound proof system that specializes pre- and post-conditions from Hoare-style proofs to capture invariants common to security proofs for programs.

2. **Information Flow State Machines**. We introduce IFSMs, an operational model that allows us to connect proofs from SymboTaint with the platform model. We develop conditions under which IFSMs can be decomposed for more efficient analysis of security properties.

3. **Speculative Abstract Platform Model**. We introduce the SAP parameterized platform model which abstractly models a speculative microprocessor. The model captures a wide combination of microarchitectural features and attack vectors, beyond what models in the literature capture. We use IFSMs in this modelling, demonstrating the compositionality of microarchitectural features.

4. **Evaluation on Transient Execution Attacks**. We evaluate our methodology - the model and proof system - by verifying transient execution attacks on the SAP model. We check a broad class of transient execution attack examples against the secure speculation [46, 82] property. We observe performance improvements over the monolithic proof approach from [46].

**Outline**. In §5.2 we motivate the problem by considering an example of an attack vector. In §5.3 we introduce the platform and attacker model, and security properties of interest. In §5.4 we develop the SymboTaint proof-system which enables Floyd-Hoare-style proofs for verification of security properties. In §5.5, we introduce IFSMs, an operational formalism that allows instrumenting the platform model with proofs written in SymboTaint. We also develop a notion of composition for IFSMs that enable concise proofs. In §5.6 we present a speculative abstract platform (SAP) model that is capable of capturing a broad class of transient execution attacks from [40, 89] and perform experimentation on this model in §5.7. We discuss related work in §5.9 and conclude in §5.10.

## 5.2 Motivation

We motivate our methodology and abstractions with the problem of verifying classes of transient execution attacks. In recent literature, secure speculation [46] has been consistently

```
1  int victim_func(int x) {
2      // A: Init secret pointer
3      int* a = &secret;
4      // B: Secret dependent code
5      ...
6      // C: Spectre V1 / BCB
7      a = arr1;
8      if (x < N)
9          tmp = arr2[a[x] * 512];
10 }
```

Figure 5.1: Victim program executing in the trusted user's domain with input x which is adversary controlled. This function is vulnerable to Spectre V1 (BCB), Spectre V4 (store-bypass), their combination and leakage from segment B.

```
// C1: lines 7-8
addi a3, gp, -88
store a3, s0, -24
bge a0, a1, END
// C2: line 9, a[x]
load a3, s0, -24;
load a3, a3, 0
add a4, a0, a3
load a4, a4, 0
// C3: line 9, arr2[a[x]*512]
muli a4, a4, 512
addi a3, gp -48
load a3, a3, 0
add a4, a4, a3
load a5, a4, 0
```

Figure 5.2: Instruction level translation of lines 4-6 of the program in Figure 5.1.

used to capture speculation dependent vulnerabilities in micro-architectures. This property is an extension of observational determinism [233] which itself is a flavor of the non-interference property [51]. Our work is based on using non-interference style properties to identify a broad class of transient execution attacks. We begin by presenting a motivating example of such an attack.

Figure 5.1 illustrates `victim_func`, a function that is owned by a victim process and is callable by an adversary process. Ignoring for the moment the first two segments labeled A and B (lines 2-5), segment C (lines 7-9) shows the first discovered transient execution attack called Spectre V1 (bounds check bypass) [116]. The way Spectre V1 works is that the adversary can train the branch predictor to mispredict the condition `(x < N)`, thereby coercing the processor to transiently execute line 9. This results in a memory access to a potential victim's secret using `a[x]` and then a secret dependent access `arr2[a[x] * 512]`. This access leaves observable side effects in the data cache covert channel. The adversary can observe these side effects and hence infer the secret.

**Complexity in software.** A vulnerable code segment such as the one above does not often appear in isolation. It may appear alongside other complex code segments, such as A and B in Figure 5.1. Segment A may non-trivially interact with segment C, with potential security implications. For example, even if the adversary was unable to mistrain the branch predictor, a faulty store-to-load forwarding of the secret address `a` from line 3 to line 9 could result in a variation of the Spectre V4 (speculative store bypass) attack [40]. Similarly, segment B can contribute its own secret dependent effects to an exploitable side channel, such as the line fill buffer [186], resulting in data leakage. This example illustrates the challenges in verifying large pieces of code monolithically and motivates approaches that decompose the proof. In §5.4 we develop such an approach which takes the form of a proof system. In §5.6

we apply this proof system to an abstract speculative microprocessor model that can execute assembly-like code.

**Complexity in hardware**. Another aspect that complicates the verification of software such as `victim_func` is the necessity to model the hardware at the microarchitectural level, resulting in a massive model. Hence, developing modeling approaches that facilitate compositionality is desirable. While this is true, reasoning over subsets of components in an unguarded manner can lead to false guarantees.

Figure 5.3, illustrates such an example inspired by the CacheOut attack [187], This attack exfiltrates in-flight data (potentially containing secrets) from the line-fill-buffer (LFB) into the cache, which then serves as the side channel. An analysis on the model in Figure 5.3A, sans the LFB, can be imprecise. Hence, compositionality though useful requires care. In §5.5.1, we present an operational modeling framework that enables composition under certain conditions. The speculative platform model from §5.6.1 has a wide range of microarchitectural components and satisfies these conditions, resulting in efficient verification.

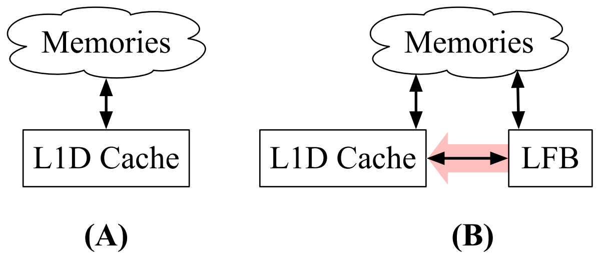


Figure 5.3: Different levels of modeling detail.

## 5.2.1 Approach Overview: Efficient Proofs with Interpolants

Security properties such as non-interference [77] are based on a notion of observation that identifies when two states are considered to be equivalent. Non-interference in particular requires that two executions which start in equivalent states must also end in equivalent states. One prominent approach [46] used to verify such properties for a program is bounded model checking (BMC) [145]. This is presented in Eq. 5.1, which is a relation over traces of two instances of the same transition system. It states that if the two instances of the system start in some initial states (say $q_1^{(0)}$ of the first instance and $q_2^{(0)}$ of the second) that are *low-equivalent* ($q_1^{(0)} \approx_{\mathrm{L}} q_2^{(0)}$), and each instance takes $k$ steps (i.e., $\delta^k(q^{(0)}, q^{(k)}) = \delta(q^{(0)}, q^{(1)}) \wedge ... \wedge \delta(q^{(k-1)}, q^{(k)})$) to transition from the initial states to the final states ($q_1^{(k)}, q_2^{(k)}$), then the final states should be low-equivalent ($q_1^{(k)} \approx_L q_2^{(k)}$).

$$q_1^{(0)} \approx_L q_2^{(0)} \wedge \delta^k(q_1^{(0)}, q_1^{(k)}) \wedge \delta^k(q_2^{(0)}, q_2^{(k)}) \Rightarrow q_1^{(k)} \approx_L q_2^{(k)} \tag{5.1}$$

However, a monolithic proof of non-interference (such as a direct translation of the Eq. 5.1 into a BMC query) results in a large verification query. One way to address the complexity of a monolithic proof is to decompose it into smaller proofs, using intermediate properties, or what we refer to as interpolants, to connect them. This can be intuitively conceptualized through the following three equations.

$$q_1^{(0)} \approx_L q_2^{(0)} \land \delta(q_1^{(0)}, q_1^{(1)}) \land \delta(q_2^{(0)}, q_2^{(1)}) \Rightarrow J_1(q_1^{(1)}, q_2^{(1)}) \tag{5.2}$$

$$\forall i \in \{1, ..., k-1\}.\ J_i(q_1^{(i)}, q_2^{(i)}) \land \delta(q_1^{(i)}, q_1^{(i+1)}) \land \delta(q_2^{(i)}, q_2^{(i+1)}) \Rightarrow J_{i+1}(q_1^{(i+1)}, q_2^{(i+1)}) \tag{5.3}$$

$$J_k(q_1^{(k)}, q_2^{(k)}) \Rightarrow q_1^{(k)} \approx_L q_2^{(k)} \tag{5.4}$$

Instead of a single query, one may break down the proof into a set of smaller verification conditions. This is possible by identifying intermediate conditions $J_1, \cdots, J_k$ such that: (a) the initial conditions and transition constraints implies $J_1$ (Eq. 5.2), (b) $J_i$ and transition constraints imply $J_{i+1}$ (Eq. 5.3), and (c) the final interpolant $J_k(q_1^k, q_2^k)$ implies that the final states are low-equivalent (Eq. 5.4).

So far this is just standard interpolant-based reasoning. The crux of effectively using this approach to address the complexity issues mentioned lies in the shape of the interpolants $J_i$ used, and how the system model (i.e., $\delta$) is represented. One type of property that can naturally serve as part of these interpolants $J_i$ for non-interference-style properties, is the class of relational properties of variables between the two system instances in a non-interference proof. Namely, this is the information described by the set of equality constraints $\{q_1^{(i)}(v) = q_2^{(i)}(v)\}_{v \in V_i}$ for $V_i \subseteq V$, where $q(v)$ denotes the value of variable $v$ in state $q$. One can view this as a summary of which variables between the two instances are still low-equivalent after the $i$-th step. This intuition is formalized in §5.4 as the *SymboTaint* proof system. On the modeling side, it is highly desirable to be able to decompose or separate a system into simpler components. Exploiting the simpler proof obligations from Eq. 5.2-5.4, one can then hope that a smaller system recomposed from the decomposed components is sufficient for sound analysis of the property. This intuition is materialized in §5.5.1 as *Information State Flow Machines*.

## 5.3 Security Model

In this section we begin by introducing our programming model in §5.3.1 followed by the attacker model in §5.3.2. We provide background on security properties in §5.3.3. We consider in this work the following security properties: non-interference [77], observational-determinism [233], and trace-property observational determinism [46].

### 5.3.1 Programming Model

We start by developing the system model based on which security properties are defined. We adopt a standard state-transition system, $M = \langle Q, I, \delta, \mathsf{Op}\rangle$ with states $Q$, initial states $I \subset Q$ and transition relation $\delta \subseteq Q \times \mathsf{Op} \times Q$ with transitions labelled by operations from $\mathsf{Op}$. It is often useful to view states as assignments to the variables in the system. In this view, $q \in Q$ is map from variables $v \in V$ to values from some domain $\mathbb{D}(v)$, i.e., $q : V \to \mathbb{D}(v)$.

A program is a word over $\mathsf{Op}$ (i.e. $P \in \mathsf{Op}^*$) which generates an execution. The execution on a program $P = \mathsf{op}_1 \cdots \mathsf{op}_n$ is a trace of states $\pi_P = q^{(0)}...q^{(n)} \in \mathit{Tr}(M)$, where the initial state belongs to $I$ and (b) consecutive states $q^{(i)}, q^{(i+1)}$ have a valid transition under $\mathsf{op}_{i+1}$: $\delta(q^{(i)}, \mathsf{op}_{i+1}, q^{(i+1)})$. For trace $\pi$, we write $\pi^{(i)}$ for the $i$-th state of the trace. $\mathit{Tr}(M)$ represents valid traces of $M$ (across programs in $\mathsf{Op}^*$), while $\mathit{Tr}_P(M)$ represents traces corresponding to program $P$. A program can lead to several traces (e.g. if the system is *non-deterministic*).

**Example 5.1** (Simple platform model)**.** *Figure 5.4 illustrates a simple platform model. The model state variables consist of the register file and the memory. The model consists of operations* `add` *and* `load`*. The* `load` *operation accesses the memory through the* `mem.load` *function. Throughout our exposition, we add more detail to this model (e.g. a cache, branch predictor, etc.), leading up to the SAP model in §5.6.1. The modeling syntax loosely follows* UCLID5 *[169] which we use to implement the verification techniques discussed later.*

## 5.3.2 Adversary Model

In this section, we characterize the *capabilities* of the adversary/attacker by defining a parameterized adversary model. The adversary model determines which behaviors constitute a vulnerability and hence influence the security specification. A common adversary model [46, 119, 204] is one that can passively observe and actively write to subsets of variables. Our approach additionally endows the adversary with the ability to transmit data between variables. This choice is motivated by microarchitectural mechanisms that move data between variables without necessarily making it visible, as illustrated in Example 5.2.

```
// Core variables and operations
core {
    // System state variables
    var pc : word_t;
    var regs : [regindex_t]word_t;
    ...

    // Operations
    operation add (rs1, rs2, rd) {
        regs[rd] = regs[rs1] + regs[rs2];
    }

    operation load(rs1, rd, imm) {
        // regs[rd] = mem[regs[rs1]+imm];
        var addr = regs[rs1]+imm;
        regs[rd] = mem.load(addr);
    }
}
```

Figure 5.4: A simple platform model.

**Example 5.2.** *In Figure 5.5, we illustrate how a transmit operation can abstractly model the effect of adversary code in the case of the Lazy-FP [202] vulnerability. In Lazy-FP, the flow of information from the secret to cache state made possible by the adversary's capability to leak information from the* `xmm` *register to the general purpose register* `rax` *using the* $\mathsf{adv}_{\mathit{flow}}(\texttt{xmm}, \texttt{rax})$ *operation (and eventually the observable cache) indicated by the solid red arrows. The victim program enables information flow from the secret to the* `xmm` *register (indicated by the dotted line).*

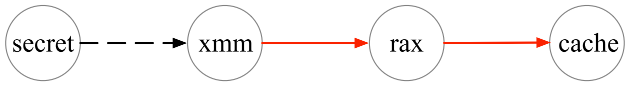


Figure 5.5: Information flow in the Lazy-FP vulnerability.

Formally, we characterize our adversary as a triple, $\mathcal{A} = \langle V_O, V_T, F \rangle$. The components indicate the set of observable state variables $V_O \subseteq V_{\mathrm{L}}$, tamperable variables $V_T \subseteq V$ and a set of *transmitting pairs* $F \subseteq V \times V$. Observable variables determine when two states are considered to be distinguishable. The tampering operations $\mathsf{Op}_T = \{\mathsf{op}_{tamp}(v) \mid v \in V_T\}$ change the value of the tampered variable to an arbitrary value: $\delta(q, \mathsf{op}_{tamp}(v), q') \iff \exists x \in \mathbb{D}.\ \ q' = q[v \leftarrow x]$. Our adversary model augments these tampering operations from [125, 204] with transmitting operations $\mathsf{Op}_{\mathit{flow}} = \{\mathsf{adv}_{\mathit{flow}}(v_1, v_2) \mid (v_1, v_2) \in F\}$. A transmitting operation $\mathsf{adv}_{\mathit{flow}}(v_1, v_2)$ establishes a flow of data from $v_1$ to $v_2$: $\delta(q, \mathsf{adv}_{\mathit{flow}}(v_1, v_2), q') \iff q' = q[v_2 \leftarrow q(v_1)]$. These adversary operations are a subset of the complete operation set: $\mathsf{Op}_O \cup \mathsf{Op}_T \cup \mathsf{Op}_{\mathit{flow}} \subset \mathsf{Op}$. The adversary executes asynchronously with the system using interleaving semantics[*].

## 5.3.3 Security Properties

**Observation**. We base our security properties on a notion of observation that dictates when two states lead to different observations (e.g. timing/power-based side-channels [205]). Our instantiation of observation identifies a subset of variables $V_{\mathrm{L}} \subseteq V$ denoted as *low* variables. These are required to have equivalent values in the two states:

$$q_1 \approx_{V_{\mathrm{L}}} q_2 \doteq \forall v \in V_{\mathrm{L}}.\ \ q_1(v) = q_2(v) \tag{5.5}$$

In the context of an entire execution, the adversary-visible §5.3.2 should be tagged as low. Using this definition we define the standard non-interference property.

**Definition 5.1** (Non-Interference)**.** *A system $M$ executing program $P$ satisfies non-interference (w.r.t. low variables $V_L$) if*

$$\forall \pi_1, \pi_2 \in \mathit{Tr}_P(M).\ \ \pi_1^{(0)} \approx_{V_L} \pi_2^{(0)} \Rightarrow \pi_1^{(n)} \approx_{V_L} \pi_2^{(n)} \tag{5.6}$$

In words, this property requires that states which start out being observationally equivalent should end up being observationally equivalent after executing $P$.

We also consider observational determinism [233] which states that executing program $P$ from indistinguishable states should result in indistinguishable states at every step. We first extend the definition of low-equivalence to traces: $\pi_1 \approx_{V_{\mathrm{L}}} \pi_2 \doteq \forall i \in \mathbb{N}.\ \ \pi_1^{(i)} \approx_{V_{\mathrm{L}}} \pi_2^{(i)}$. Observational determinism can be formalized as follows:

[*]Note that the adversary and system can take an arbitrary number of steps.

**Definition 5.2** (Observational Determinism)**.** *A system $M$ executing program $P$ satisfies observational-determinism (OD) w.r.t low variables $V_L$ if*

$$\forall \pi_1, \pi_2 \in Tr_P(M).\ \ \pi_1^{(0)} \approx_{V_L} \pi_2^{(0)} \Rightarrow \pi_1 \approx_{V_L} \pi_2 \tag{5.7}$$

An extension of observational determinism [46] captures trace property-dependent violations of observational determinism. While OD requires that any two traces that are initially equivalent be always equivalent, Trace-Property Observational Determinism (TPOD) relaxes this condition in two ways. First, TPOD restricts these traces $(\pi_3, \pi_4)$ to a trace set $T_2$. Secondly, TPOD only enforces their equality when two other traces $(\pi_1, \pi_2)$ from trace set $T_1$ are equivalent. We refer the reader to [46] for details.

**Definition 5.3** (TPOD)**.** *Given trace properties $T_1 \subset Tr_P(M)$ and $T_2 \subset Tr_P(M)$, a system $M$ satisfies trace property-dependent observational determinism when executing program $P$ if*

$$\forall \pi_1, \pi_2 \in T_1 . \pi_3, \pi_4 \in T_2 . \big(\pi_1 \approx_{V_L} \pi_2 \land \pi_3^{(0)} \approx_{V_L} \pi_4^{(0)}\big) \Rightarrow \pi_3 \approx_{V_L} \pi_4 \tag{5.8}$$

**Taint contexts**. Taint analysis [188] is an approach that can perform an approximate (typically over-approximate) analysis of the system w.r.t. security properties. These approaches generally consider a set $\mathcal{L}$ of security labels [58, 183]. A *taint-context* $\Gamma$ maps variables to taint-labels, $\Gamma : V \to \mathcal{L}$. When the label set consists of two labels low and high, $\{\mathtt{L}, \mathtt{H}\}$, we can view $\Gamma(v) = \mathtt{L}$ to mean that the variable $v$ is untainted (by any high - $\mathtt{H}$ - values). Consequently, this interpretation of taint can be related to the notion of equivalence of variables. In particular, when the taint assigned to $v$ is low at some point in the execution, $\Gamma(v) = \mathtt{L}$, $v$ only depends on variables that had initially had $\mathtt{L}$ taint. If the latter were observationally equal then $v$ is as well. Hence, the taint can be thought of as a relational property marking variables that take identical values across executions. We make use of this when developing the SymboTaint proof system. We can define indistinguishability in terms of the taint-context:

$$q_1 \approx_\Gamma q_2 \doteq \forall v \in V.\ \ \Gamma(v) = l \Rightarrow q_1(v) = q_2(v) \tag{5.9}$$

Eq. 5.5 relates to this as: $q_1 \approx_{\{v \ \mid \ \Gamma(v)=\mathtt{L}\}} q_2 \iff q_1 \approx_\Gamma q_2$. We also use $\delta_\tau$ as the transition relation over taint-contexts: $\delta_\tau \subseteq Q \times (V \to \mathcal{L}) \times \mathtt{Op} \times (V \to \mathcal{L})$.

## 5.4 The SymboTaint Proof System

In this section, we develop a methodology to verify a given program running on a platform model with respect to security properties from §5.3.3. Our technique is a proof-system based on Hoare-logic [85] and interpolant-based reasoning [144]. Standard Hoare-logic inductively builds a proof for a program by composing Hoare-triples defined for atomic statements to get a pre-post condition for the full program. The pre-post conditions are typically state sets (predicates) that over-approximate the actual set of states reached. This is adequate when one is concerned with proving properties over individual executions. However, the security properties of interest (§5.3.3) are *hyperproperties* defined over pairs of executions. This requires us to augment the shape of the interpolants used in the proof-system, which we now discuss.

### 5.4.1 Joint Symbolic-Taint Interpolants

Our proof system uses interpolants of the form $\{S, \Gamma\}$ where $S$ is a set of states and $\Gamma$ is a collection of relation equality constraints. While standard Hoare-logic/interpolant-based approaches use formulae that hold over individual states, $\{S, \Gamma\}$ holds on *a pair of states*. This is defined through a judgement $(q_1, q_2) \models \{S, \Gamma\}$ (where $q_1, q_2 \in Q$):

$$(q_1, q_2) \models \{S, \Gamma\} \doteq (q_1 \in S) \wedge (q_2 \in S) \wedge q_1 \approx_\Gamma q_2$$

We define the SymboTaint-triple $\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\}$ over an individual operation $\mathsf{op}$ similar to Hoare-logic. If two states satisfying $\{S, \Gamma\}$ transition on $\mathsf{op}$, then any pair of post-states satisfy $\{S', \Gamma'\}$:

$$\begin{aligned}\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\} \doteq\ & \forall q_1, q_2, q_1', q_2'. \\ & ((q_1, q_2) \models \{S, \Gamma\} \wedge \delta(q_1, \mathsf{op}, q_1') \wedge \delta(q_2, \mathsf{op}, q_2')) \implies (q_1', q_2') \models \{S', \Gamma'\}\end{aligned}$$

We observe that the symbolic-taint interpolants *can be composed*, giving us proof rules for sequential composition and iteration similar to Hoare-logic. We provide the full set of proof rules in the Appendix. Then starting with the triple for a single operation as the base case, we can build proofs for larger programs. This key feature allows us to decompose proofs for large programs into a set of verification conditions (VC) over small sequences of operations. Each of these VCs can be discharged more effectively than a single VC for the full program.

**Self-composition vs. SymboTaint**. Hyper-properties such as non-interference can be thought of as (single trace) safety properties over the self-composition of the system [51]. Following this observation, one could build a self-composition of the system and use interpolants defined over two copies of the variables. However, this approach defines symbolic constraints on twice the variables, putting strain on the underlying model-checker. We develop a different route.

Our choice of interpolants is based on the observation that the properties in §5.3.3 mention equality over pairs of variables from the copies of the system. We encode this as a taint-context $\Gamma : V \to \mathcal{L}$ which marks whether a variable takes equal values. When $\Gamma(v) = \mathtt{L}$ then $v$ takes the same value across both executions.

```
 1 core {
 2   ...
 3   operation load(rs1, rd, imm) {
 4     // mem[regs[rs1]+imm] = rd;
 5     var addr = regs[rs1]+imm;
 6     // Redefined model semantics
 7     if (cache.is_hit(addr, dom))
 8       regs[rd] = cache.load(addr, dom)
 9     else
10       regs[rd] = mem.load(addr);
11   }
12   ...
13 }
14
15 // Cache component
16 cache {
17  var mdata : [set_index_t]word_t;
18  var data : [set_index_t]word_t;
19  // Internal functions
20  get_index(addr, dom): set_index_t =
21    addr[63:22] ++ dom;
22  get_tag(addr) : tag_t = addr[21:8];
23  is_hit(addr,dom): bool = get_tag(addr
      )
24    == mdata[get_index(addr, dom)];
25 }
```

Figure 5.6: Modified platform model from Figure 5.4 with a partitioned cache.

**Example 5.3.** *In Figure 5.6 we depict a snippet from a model extending Ex. 5.1. In this model, before a* `load` *operation invokes* `mem.load` *to fetch an address from the memory it checks for the address in the cache (*`cache.is_hit`*). The model performs cache partitioning [113] by dividing the cache into partitions each of which is only accessible by a single process domain (*`dom`*). The model partitions the cache by set index (line 19); the index depends on the address and the process domain. Suppose the victim domain 0 is allocated indices $\leq k$ while the attacker domain 1 is allocated the rest.*

*An access made by the victim (domain 0) on address* `addr`*, results in the satisfaction of the Hoare-triple $\{Q, \Gamma_0\}\ M(\texttt{load})\ \{S_1, \Gamma_1\}$ where $\Gamma_0 = [\lambda i > k.i \leftarrow \texttt{L}]$ is the taint context with domain 1 accessible indices marked low. Additionally, the symbolic state captures the fact that domain 0 is making the access. Then following the partitioning semantics encoded in $M(\texttt{load})$, we can infer a post-judgement where $\Gamma_1 = \Gamma_0$. That is after performing the load operation, domain 1 visible state remains low-equivalent. This proves that a victim executed* `load` *does not modify attacker-visible state.*

### 5.4.2 Connecting the Proof System with Security Properties

In this section, we connect the proof system developed in §5.4.1 with security properties. This connection is based on the fact that we can develop SymboTaint-triples which, under some conditions, are sound with respect to the properties defined in §5.3.3. We now discuss these conditions, starting with non-interference.

**Non-interference**. When proving non-interference with respect to the low variables $V_{\texttt{L}}$ and program $P$, we generate a valid triple of the form: $\{I, \Gamma_0\}\ M(P)\ \{Q, \Gamma_f\}$ with the following condition, denoted as CondNI:

$$\text{CondNI} : \quad \forall v \notin V_{\texttt{L}}.\ \ \Gamma_0(v) \neq \texttt{L} \quad \wedge \quad \forall v \in V_{\texttt{L}}.\ \ \Gamma_f(v) = \texttt{L} \tag{5.10}$$

In the pre-condition, we allow a low ($\mathtt{L}$) assignment to variables in $V_{\mathtt{L}}$. This ensures that the antecedent of the non-interference property is implied. We require that the final taint context $\Gamma_f$ assign $\mathtt{L}$ to $V_{\mathtt{L}}$, which implies the consequent of non-interference.

**Observational determinism**. The difference between observational determinism and non-interference is that in the former, all taint contexts must assign $\mathtt{L}$ to $V_{\mathtt{L}}$. Hence, in this case, we want to generate valid triples where CondObsDet holds:

$$\begin{aligned}&\{I, \Gamma_0\}\ M(\mathsf{op}_1)\ \{S_1, \Gamma_1\}\ M(\mathsf{op}_2)\ \cdots \{S_n, \Gamma_n\}\\ &\text{CondObsDet} : \forall v \notin V_{\mathtt{L}}.\ \ \Gamma_0(v) \neq \mathtt{L}\ \wedge\ \forall i \in [1..n].\ \ \forall v \in V_{\mathtt{L}}.\ \ \Gamma_i(v) = \mathtt{L}\end{aligned} \tag{5.11}$$

The first judgement (pre-condition) is identical to the case of non-interference. However, in this case, we need to choose the intermediate interpolants such that each of the taint-contexts assign the $\mathtt{L}$ label to the variables in $V_{\mathtt{L}}$.

**Trace property-dependent observational determinism**. While non-interference and observational determinism are properties over two traces, TPOD is over four traces. This requires us to consider a *self-composition* of the platform transition system. Additionally, TPOD only enforces observational-equivalence when the traces belong to $T_1$ and $T_2$. This allows us to strengthen the proof system for TPOD with auxiliary invariants that over-approximate $T_1$ and $T_2$. We call these invariants *cover-invariants*. We now briefly discuss these concepts.

**Self composition of** $M$. The self-composition of $M$ is the transition system $M^2 = \langle Q^2, \delta^2, I^2 \rangle$. The new state space is $Q^2 = Q \times Q$, the transition relation $\delta^2$ enforces $\delta$ on both the first and second copies of the state, and $I^2 = I \times I$. We also consider the paired state as an assignment to two copies of variables: $V^1 = \{v^1\}_{v \in V}$ and $V^2 = \{v^2\}_{v \in V}$.

**Cover invariants for trace-properties**. In order to capture the trace-properties $T_1, T_2$ that TPOD enforces, we allow invariants $I_1^{tpod}, I_2^{tpod} \subseteq Q$ that are implied by $T_1$ and $T_2$ respectively. That is, if $\pi \in T_1$ then $\forall i.\ \pi^i \in I_1^{tpod}$, and similarly for $T_2$ and $I_2^{tpod}$. While cover invariants can just be *True*, tighter invariants can lead to stronger proofs.

For TPOD, we require the following valid triples to hold over the self-composed system $M^2$ and program $P = \mathsf{op}_1 \cdots \mathsf{op}_n$ such that CondTPOD holds:

$$\begin{aligned}&\{S_0, \Gamma_0\}\ M(\mathsf{op}_1)\ \{S_1, \Gamma'_1\}, \{S_1, \Gamma_1\}\ M(\mathsf{op}_2)\ \{S_2, \Gamma'_2\}, \cdots, \{S_{n-1}, \Gamma_{n-1}\}\ M(\mathsf{op}_n)\ \{S_n, \Gamma'_n\}\\ &\text{CondTPOD} : S_0 = (I \cap I_1^{tpod}) \times (I \cap I_2^{tpod})\ \wedge\\ &\quad \forall i.\ \forall v \notin V_{\mathtt{L}}^1.\ \ \Gamma_i(v) \neq \mathtt{L}\ \wedge\ \forall v \notin V_{\mathtt{L}}^2.\ \ \Gamma_0(v) \neq \mathtt{L}\ \wedge\\ &\quad \forall i \in [1..n].\ \forall v \in V_{\mathtt{L}}^2.\ \ \Gamma'_i(v) = \mathtt{L}\ \wedge\ \forall i \in [1..(n-1)].\ \forall v \in V^2.\ \ \Gamma_i(v) = \Gamma'_i(v)\end{aligned} \tag{5.12}$$

**Theorem 5.1 (Soundness).** *If there is a Hoare-triple for $M$ under program $P$ that is valid w.r.t. conditions* CondNI *(resp.* CondObsDet, CondTPOD*) then the system $M$ satisfies non-interference (resp. observational-determinism, TPOD) on program $P$.*

**Proof rules**. We provide the full set of proof rules in Figure 5.7. In addition to the sequential composition of the proof rules, we also define a proof rule for strengthening the pre-condition and weakening the post-condition of judgments. This is analogous to the consequence rule in the classical Floyd-Hoare system (e.g. [85]). We denote this as rule (3) in Figure 5.7.

$$\textsc{RBase}\ \frac{\begin{array}{l}\forall q_1, q_2, q_1', q_2'.\ \big((q_1, q_2) \models \{S, \Gamma\} \wedge \\ \delta(q_1, \mathsf{op}_1 \cdots \mathsf{op}_n, q_1') \wedge \\ \delta(q_2, \mathsf{op}_1 \cdots \mathsf{op}_n, q_2')\big) \implies (q_1', q_2') \models \{S', \Gamma'\}\end{array}}{\{S, \Gamma\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S', \Gamma'\}}$$

$$\textsc{RSeq}\ \frac{\begin{array}{c}\{S, \Gamma\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_k)\ \{S'', \Gamma''\} \\ \{S'', \Gamma''\}\ M(\mathsf{op}_{k+1} \cdots \mathsf{op}_n)\ \{S', \Gamma'\}\end{array}}{\{S, \Gamma\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S', \Gamma'\}}$$

$$\textsc{RCons}\ \frac{\begin{array}{c}(S_1 \subseteq S_1'\ \wedge\ \Gamma_1 \sqsubseteq \Gamma_1') \\ (S_2' \subseteq S_2\ \wedge\ \Gamma_2' \sqsubseteq \Gamma_2) \\ \{S_1', \Gamma_1'\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S_2', \Gamma_2'\}\end{array}}{\{S_1, \Gamma_1\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S_2, \Gamma_2\}}$$

Figure 5.7: Proof rules for joint symbolic-taint judgments.

With these rules, we can now prove the soundness theorem 5.1.

*Proof.* In the case of non-interference (NI), if 5.10 holds then, we also have

$$\{I, \Gamma^*\}\ M(P)\ \{Q, \Gamma_f\} \tag{5.13}$$

where $\Gamma^* = [V_{\mathtt{L}} \to \mathtt{L}, V_{\mathtt{H}} \to \mathtt{H}]$ (by RCons). Now consider any pair of traces $\pi_1, \pi_2 \in Tr_P(M)$. If $\pi_1^{(0)} \approx_{V_{\mathtt{L}}} \pi_2^{(0)}$ (precondition of NI, Defn. 5.6) then $(\pi_1^{(0)}, \pi_2^{(0)}) \models \{I, \Gamma^*\}$. Consequently (by Eq. (5.13), every pair of final states satisfy $(\pi_1^{(n)}, \pi_2^{(n)}) \models \{Q, \Gamma_f\}$. Then, by the condition on $\Gamma_f$ (in Eq. (5.10)), we get $\pi_1^{(n)} \approx_{V_{\mathtt{L}}} \pi_2^{(n)}$ as desired. The proof for OD is similar, however, we use the intermediate taint-contexts to show equivalence $\pi_1^{(i)} \approx_{V_{\mathtt{L}}} \pi_2^{(i)}$ for each step $i$.

We now provide a proof for TPOD. Suppose there exist traces $\pi_1, \pi_2, \pi_3, \pi_4 \in Tr_P(M)$ that satisfy the preconditions of TPOD. That is (A) $\pi_1, \pi_2 \in T_1$, (B) $\pi_3, \pi_4 \in T_2$, (C) $\pi_1 \approx_{V_\mathtt{L}} \pi_2$ and (D) $\pi_3^{(0)} \approx_{V_\mathtt{L}} \pi_4^{(0)}$. Then we proceed by induction to show that $((\pi_1^{(i)}, \pi_3^{(i)}), (\pi_2^{(i)}, \pi_4^{(i)})) \models \{S_i, \Gamma_i\}$ holds for each $i$. The base case follows by (Eq. (5.12)) since $\pi_1^{(0)}, \pi_2^{(0)} \in I \cap I_1^{tpod}$ (since $I_1^{tpod}$ is a cover invariant for $T_1$), and similarly $\pi_3^{(0)}, \pi_4^{(0)} \in I \cap I_2^{tpod}$.

Now assume (inductive case) that it holds for some $i$. The $((\pi_1^{(i+1)}, \pi_3^{(i+1)}), (\pi_2^{(i+1)}, \pi_4^{(i+1)})) \models \{S_{i+1}, \Gamma'_{i+1}\}$ holds by (F) and RBase. Now, the fact that $\Gamma_i$ and $\Gamma'_i$ agree on $V_\mathtt{L}^2$ and that $(\pi_1^{(i)} \approx_{V_\mathtt{L}^1} \pi_2^{(i)})$ implies $((\pi_1^{(i+1)}, \pi_3^{(i+1)}), (\pi_2^{(i+1)}, \pi_4^{(i+1)})) \models \{S_{i+1}, \Gamma_{i+1}\}$. This shows the inductive case. Finally, this implies $\pi_3^{(i)} \approx_{V_\mathtt{L}^2} \pi_4^{(i)}$ for each $i$ since $\Gamma_i(v) = \mathtt{L}$ for $v \in V_\mathtt{L}^2$. □

## 5.5 IFSMs: Operational Encoding of SymboTaint

In this section, we discuss an operational approach to encode the proof-based reasoning developed in §5.4. This allows us to represent proofs in the form of executions of a standard symbolic transition system. This has the following prominent advantages. Firstly, it is easier to connect (by way of instrumentation) an operationally encoded proof with a platform model that is represented as a transition system. Then, off-the-shelf model-checking tools can be used to perform verification on the proof-instrumented platform model (e.g. bounded/unbounded model checking, invariant inference, etc.). We apply this to secure and insecure cases in §5.7 where we analyze an abstract platform model. Additionally, an operational encoding allows us to perform structural composition (§5.5.2) of parts of the platform. In particular, this allows projecting away components that are not relevant to the proof of a certain property. This is advantageous since platforms (over which we evaluate our techniques) are built hierarchically.

### 5.5.1 Information Flow State Machine

At a high level, an IFSM is a state-transition system that encodes a joint symbolic-taint ($\{S, \Gamma\}$) judgement in its state. This encoding is performed by augmenting the state from the system §5.3.1 with a taint-context ($\Gamma : V \to \mathcal{L}$). Consequently, IFSM creation can be thought of as instrumenting a platform model with taint-tracking variables.

The transitions of the IFSM update the system state following the transition relation $\delta \subset Q \times \mathsf{Op} \times Q$ from §5.3.1, and update the taint-context following its transition relation, $\delta_\tau$. If a transition is allowed in the IFSM, then the corresponding Hoare-triple holds in the proof system of §5.4. This key feature implies the soundness of safety proofs that use IFSMs. The IFSM is also parameterized by the initial taint-context, $\Gamma_0$. The choice of initial taint context depends on the property being proved. For example, if one is concerned with proving non-interference with $V_\mathtt{L}$ as the set of low variables, the initial taint-context assigning $\mathtt{H}$ to all non-low variables provides the correct antecedent: $\Gamma_0 = [V_\mathtt{L} \to \mathtt{L}, V_\mathtt{H} \to \mathtt{H}]$. We now formally define an IFSM.

**Definition 5.4** (Information Flow State Machine)**.** *An IFSM is a transition system* $\langle \mathsf{Q}, \Delta, \mathsf{I}\rangle$*, with a set of configurations* $\mathsf{Q}$*, the transition relation* $\Delta$*, and a set of initial states* $\mathsf{I}$*. Each configuration pairs a platform state with a taint context:* $\mathsf{Q} = Q \times (V \rightarrow \mathcal{L})$*. The transition relation combines the platform variable updates with taint transitions:* $\Delta((q_1, \Gamma_1), \mathsf{op}, (q_2, \Gamma_2)) \iff \delta(q_1, \mathsf{op}, q_2) \wedge \delta_\tau((q_1, \Gamma_1), \mathsf{op}, \Gamma_2)$*. where* $\delta$ *is the platform transition relation and* $\delta_\tau$ *is the taint-context transition relation. Finally the set of initial configurations is defined as:* $\mathsf{I} = I \times \Gamma_0$*.*

### 5.5.2 Composing IFSMs

In this section, we discuss structural compositionality of IFSMs. Our notion of composition is based on the observations that (a) hardware platforms are typically hierarchical in nature and (b) only some components in the design hierarchy transition for certain operations. We define a composition of two IFSMs as follows.

**Definition 5.5** (Composition of IFSMs)**.** *The composition of* $M_1 = \langle \mathsf{Q}, \Delta_1, \mathsf{I}_1\rangle$ *and* $M_2 = \langle \mathsf{Q}, \Delta_2, \mathsf{I}_2\rangle$ *is* $M_1||M_2 = \langle \mathsf{Q}, \Delta, \mathsf{I}\rangle$*, where:* $\Delta = \Delta_1 \wedge \Delta_2$ *and* $\mathsf{I} = \mathsf{I}_1 \wedge \mathsf{I}_2$*.*

The composition conjoins transition relations and starting states of $M_1$ and $M_2$. The new transition relation enforces constraints from both component IFSMs. This may lead to the new IFSM not having any valid transitions (e.g. when $M_1$, $M_2$ require conflicting updates to a variable). We identify the conditions under which this does not happen.

**Separability.** To allow composability, we require that the overall transition relation is *separable* into per-variable components: $\delta(q, \mathsf{op}, q') \iff \wedge_v\ \delta^v(q, \mathsf{op}, q'(v))$. Intuitively, the relation $\delta^v \subseteq Q \times \mathsf{Op} \times \mathbb{D}$ localizes the effect of $\delta$ on an individual variable $v$. The post-values admitted by all individual $\delta^v$s can be combined to generate valid next-state assignments. We define $\delta^v_\tau$ similarly, by replacing $\delta$ with $\delta_\tau$, $q$ with $\mathsf{q} \doteq (q, \Gamma)$ and $q'(v)$ with $\Gamma'(v)$ in the above equation: $\delta_\tau(\mathsf{q}, \mathsf{op}, \Gamma') \iff \bigwedge_v\ \delta^v_\tau(\mathsf{q}, \mathsf{op}, \Gamma'(v))$.

**Definition 5.6** (Separability)**.** *An IFSM* $M$ *is separable if the following hold:*

$$\forall q, q' \in Q, \forall v \in V, \exists \delta^v,\ \textit{s.t.}\ \delta(q, \mathsf{op}, q') \iff \delta^v(q, \mathsf{op}, q'(v)) \tag{5.14}$$

$$\forall \mathsf{q} \in \mathsf{Q}, \Gamma' \in (V \rightarrow \mathcal{L}), \forall v \in V, \exists \delta^v,\ \textit{s.t.}\ \delta_\tau(\mathsf{q}, \mathsf{op}, \Gamma') \iff \delta^v_\tau(\mathsf{q}, \mathsf{op}, \Gamma'(v)) \tag{5.15}$$

**Projection Consistency.** In addition to separating the effects of a transition, we also need to identify when a particular component drives a certain variable. We denote this by the *guard predicate* $C^v(q, \mathsf{op})$ which is true when $v$ is driven by the component and false otherwise. Formally, we can write this as follows:

**Definition 5.7** (Projection Consistent)**.** *A decomposition of $\delta$ into a collection of localized $\{\delta^v\}_{v\in V}$ is projection consistent if*

$$C^v(q, \mathsf{op}) \vee \forall x \in \mathbb{D}(v).\ \delta^v(q, \mathsf{op}, x) \tag{5.16}$$

When $C^v(q, \mathsf{op})$ is true, the component enforces specific next values for variable $v$. Otherwise, $\delta^v$ holds for all the next values, i.e., the component does not enforce any constraints on the new value. This allows the localized effects of $v$, i.e., $\delta^v$, to be *consistent* under the composition definition such that if a system was decomposed into sub-systems, the composition of the sub-systems is equivalent to the original system. When composing two or more components, we require that at each step the guard of at most one component be true. This ensures non-conflicting updates.

### 5.5.3 Compositional Verification with IFSM

Compositionality allows us to view the full platform model as a set of separable constraints imposed on each variable. We can make full use of this separability, by composing constraints from only necessary models. Consider the task of verifying the composition $M = M_1||...||M_N$. Then it suffices to compose only those components that drive some variable $v$. For operation $\mathsf{op}$, we denote such components as $\mathbf{I}(\mathsf{op}) \doteq \{i \in [N] \mid \exists q \in Q, v \in V.\ \ C_i^v(q, \mathsf{op})\}$. Then we have the following corollary.

**Corollary 5.1** (Minimal Composition)**.** *If each IFSM $\{M_i\}_{i\in[N]}$ of a composition $M = M_1|| \cdots ||M_N$ is separable, is projection consistent and the guards of each $\{M_i\}_{i\in[N]}$ are pairwise-disjoint, that is $\forall i, j \in [N], v \in V, q \in Q.\ i \neq j \Rightarrow \neg\big(C_i^v(q, \mathsf{op}) \wedge C_j^v(q, \mathsf{op})\big)$, then we have (where $\mathbf{I}(\mathsf{op}) = \{j_1, \cdots, j_k\}$):*

$$\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\} \iff \{S, \Gamma\}\ \ (M_{j_1}||M_{j_2}|| \cdots ||M_{j_k})\,(\mathsf{op})\ \{S', \Gamma'\} \tag{5.17}$$

*Proof.* Let $M = \langle \mathsf{Q}, \Delta, I\rangle$, $M' = (M_{j_1}||M_{j_2}|| \cdots ||M_{j_k}) = \langle \mathsf{Q}, \Delta', \mathsf{I}'\rangle$ be the *minimal* composition as defined above and let $M_i = \langle \mathsf{Q}, \Delta_i, \mathsf{I}_i\rangle$. We show that if $\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\}$ and $\{S, \Gamma\}\ M'(\mathsf{op})\ \{S'', \Gamma''\}$, then $\{S', \Gamma'\} = \{S'', \Gamma''\}$.

By Eq. (5.14) and Eq. (5.15), it suffices to prove that the execution of $\mathsf{op}$ in $M$ and $M'$ both permit or forbid the transition from state $q$ to the next state with variable $v$ assigned to $x$ in the same way, i.e., $\Delta^v(\mathsf{q}, \mathsf{op}, x) \iff \Delta'^v(\mathsf{q}, \mathsf{op}, x)$. Without loss of generality, consider an arbitrary $v \in V$. By assumption, each pairwise guard satisfies $\neg(C_i^v(q, \mathsf{op}) \wedge C_j^v(q, \mathsf{op})$ for any $q \in Q$ and $i \neq j$. Thus, for a given $v \in V$ and $q \in Q$, $\neg C_1^v(q, \mathsf{op}) \wedge \cdots C_i^v(q, \mathsf{op}) \cdots \neg C_N^v(q, \mathsf{op})$. This means that only one guard is true for a given execution of operation $\mathsf{op}$ in any state $q \in Q$. With this, the assumption Eq. (5.16), and definition of $\mathbf{I}(\mathsf{op})$, we know that only one component of $M$ assigns to $v$, giving us $\Delta^v(\mathsf{q}, \mathsf{op}, x) = \Delta_1^v(\mathsf{q}, \mathsf{op}, x) \wedge \cdots \wedge \Delta_N^v(\mathsf{q}, \mathsf{op}, x) = \mathit{true} \wedge \cdots \wedge \Delta_i^v(\mathsf{q}, \mathsf{op}, x) \wedge \cdots \wedge \mathit{true} = \Delta_i^v(\mathsf{q}, \mathsf{op}, x)$. Since the right-hand side of the preceding equation is precisely the behavior of $\Delta'$ for variable $v$, i.e., $\Delta'^{v}(\mathsf{q}, \mathsf{op}, x) \doteq \Delta_i^v(\mathsf{q}, \mathsf{op}, x)$, this completes the proof. $\square$

In the following sections, we utilize this notion of minimal composition when verifying transient execution attacks on the platform model of a microprocessor.

## 5.6 Verifying Speculative Platforms with IFSMs

Existing works on modeling and verifying security properties for processor platforms commonly develop the platform model based on *speculative semantics* [46, 68, 82, 83]. These platform models specify instruction semantics at the microarchitectural level. These mappings from instructions to micro-architectural effects are often hard-coded making extensions to these semantics difficult. A modeling language that allows flexible specification of microarchitectural features can address these challenges. We present the Speculative Abstract Platform (SAP), which models a general class of microprocessor designs. The SAP model is implemented using the modeling and verification language UCLID5 [169] which allows for parameterization and composition of model components. While we model a wide range of microarchitectural features, for space reasons, we only present the cache and branch prediction components of the SAP model in this section. For full details, we refer the reader to the repository at `https://github.com/ifsm-sp2023/sap`.

### 5.6.1 The Speculative Abstract Platform

The SAP model consists of several abstract components represented as IFSMs. The model includes a CPU core, a data cache, a line fill buffer, load and store buffers, a page table, a translation lookaside buffer, a pattern history table, a branch target buffer and a power state. We note that the SAP model serves as an initial abstraction which one may use for the analysis of speculative programs. To exemplify our methodology, we describe the following three IFSM components of the SAP in more detail and describe their composition: the CPU, cache and branch predictor models. We omit details about virtualization from this example for simplicity.

| **Model** | **State Var.** | Description |
|---|---|---|
| CPU | `pc` | The program counter. |
| | `regs` | Registers. |
| | `mem` | Physical memory. |
| | `excp` | Exception register. |
| | `pid` | Current executing process. |
| Cache | `cache_valid` | Cache index to valid bit. |
| | `cache_tag` | Cache index to entry tag. |
| Branch prediction | `pht` | Pattern history table. |
| | `btb` | Branch target buffer. |

Figure 5.8: CPU, cache, and branch prediction components of the SAP model.

**Abstract CPU model**. The CPU model $M_{core}$, is an abstraction of architectural states, which include the variables in the first row of Table 5.8. The program counter, registers,

```
// Branch prediction
branch_prediction {
  var pht : [addr_t]counter_state_t;

  // internal functions
  ctr_to_dir(ctr_state) : boolean;
  predict_dir(addr) : boolean =
    ctr_to_dir(pht[addr]);
  ...

  operation bge(rs1, rs2, addr) {
    // speculatively predict branch
    if (predict_dir(pc)) {
      pc = pc + 4;
    } else {
      pc = addr;
    }
  }

  guard bge() { return *; }
  ...
}
```

Figure 5.10: Branch prediction in the SAP model

physical memory, and exception registers[†] are denoted by `pc`, `regs`, `mem`, `excp` respectively. We use `pid` to denote the domain of the executing process (adversary or victim). The set of operations (`Op`) define the transition relation semantics $\delta$. For the CPU, these operations are load (`load`), store (`store`), conditional branch (`bge`), add (`add`), jump (`jmp`) and other instructions containing only the ISA level semantics. This model was described earlier as Figure 5.4.

**Abstract cache model**. The second row in Table 5.8 describes the cache model $M_{cache}$, which contains a map of cache indexes to valid bits and tags of cache lines (this is abstracted away in Figure 5.6). In addition to the internal functions as shown in Figure 5.6, the cache model contains the `load` operation which reads from `data` variable. The cache model redefines the semantics of the CPU's `load` operation and thus we associate the guard $C^{\texttt{regs[rd]}}(q, \texttt{load}) := \texttt{is\_hit(addr,pid)}$, where $\texttt{addr} := \texttt{q.regs[rs1]} + \texttt{imm}$. Figure 5.9 expands on Figure 5.6 to illustrate this.

```
// Cache model continued
cache {
  ...
  operation load(addr, dom) {
    return data[get_index(addr, dom)];
  }
  guard load(addr, dom) {
    return is_hit(addr, dom);
  }
}
```

Figure 5.9: Continued cache model from Figure 5.6 with the load operation and guard.

**Abstract branch predictor model**. The abstract branch predictor $M_{BP}$, redefines branch instructions with speculative semantics by using a pattern history table (`pht`). It takes the conventional *always mispredict* semantics common in existing models [68, 82]. For example, the branch-if-greater-equal (`bge`) operation makes a branch decision based on the `pht` using an uninterpreted function `ctr_to_dir`. This takes as argument the state of the counter for a given address and returns a branch direction. The guard associated to this operation,

[†]We note that the number of exceptions is not comprehensive and only includes page faults, abort pages and device-not-available exceptions to accommodate the variants we verify.

$C^{pc}_{BP}(q, \texttt{bge}) := *$, allows the model to non-deterministically choose (indicated by $*$) between executing `bge` in the `branch_prediction` or the `core`. Figure 5.10 further expands on Figure 5.9 to illustrate the extended model with the abstract branch predictor.

## 5.6.2 Composing IFSM Models

**Composing SAP components**. To verify our properties efficiently, we compose the abstract models from §5.6.1 using Def. 5.5. In addition, the models are designed such that the transitions satisfy Eq. 5.16 and the collection of guards are disjoint. Subsequently, we make use of Cor. 5.1 to verify "*minimal*" compositions with the interpolant-based approach. To illustrate this, consider the task of verifying non-interference for Figure 5.2. Figure 5.11 shows the program component that contains three operations corresponding to the blocks in Figure 5.2. Each instruction level operation within the block corresponds to a composition of the operations from $M_{core}$, $M_{cache}$ and $M_{BP}$.

```
// Branch prediction
program {
  operation block_C1() {
    addi(a3, gp, -88);
    store(a3, s0, -24);
    bge(a0, a1, END);
  }
  ...
  operation block_C3() {
    ...
    load(a3, a3, 0);
    add(a4, a4, a3);
    load(a5, a4, 0);
  }
}
```

Figure 5.11: Program composed with the SAP model.

**Executing the victim program on the SAP model**. First, by construction, our models $M_{core}$, $M_{cache}$, and $M_{BP}$ satisfy Eq. 5.16 because when the guard of a variable $v$ evaluates to false, we do not update variable $v$. Second, each guard is written to be disjoint. This allows us to use Corollary 5.1 to compose only the necessary models for computing the symbolic-taint interpolants. For example, computing the post-interpolant for `block_C1()`, the composition $M_{core}||M_{cache}||M_{BP}$ is used because of the `store` and `bge` instructions. On the other hand, computing the post-interpolant of `block_C3()` only requires $M_{core}||M_{cache}$. This results in a shorter compilation time of the models for each computation of an interpolant and a smaller model to reason about.

## 5.7 Case Studies

While several approaches [46, 82, 153] perform verification w.r.t. secure speculation, vulnerabilities such as ÆPIC [29] transcend secure speculation, due to which we check observational determinism (Eq. 5.11), which in turn implies secure speculation.

**Transient execution attacks and their adversary model parameterization.**

| | Execution | | $V_O$ | | $V_T$ | | | | | | | $F$ | | Spec. Feature Exploited | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Vulnerability*** | ***Asynchronous*** | ***Entry Points*** | ***L1D*** | ***AVX2 Power*** | ***PHT*** | ***RSB/BTB*** | ***Page Tables*** | ***Store Buffer*** | ***L1D Cache*** | ***Load Port*** | ***LFB*** | $v_1$ | $v_2$ | ***Branch Pred.*** | ***BTB/RSB Pred.*** | ***STL Forward*** | ***Reg. Perm.*** | ***Mem. Perm.*** |
| Spectre v1 [116] | | × | × | | × | | | | × | | | | | × | | | | |
| Spectre v2 [116] | | × | × | | | × | | | × | | | | | | × | | | |
| Spectre v4 [116] | | × | × | | | | | × | × | | | | | | | × | | |
| Meltdown [133] | | × | × | | | | | | × | | | | | | | | | × |
| Foreshadow [215] | | × | × | | | | × | | × | | | | | | | | | × |
| LVI [216] | | × | × | | × | × | × | × | × | × | × | | | | | × | | × |
| NetSpectre [190] | | × | × | × | × | | | | × | | | | | × | | | | |
| LazyFP [202] | | × | × | | | | | | × | | | `XMM` | `L1D` | | | | × | |
| RIDL [186] | × | | × | | | | | | × | × | × | `LFB` | `L1D` | | | | × | |

Table 5.1: The execution column indicates whether the adversary is allowed to execute asynchronously or only before and after the victim program (at entry points). $V_O$, $V_T$ and $F$ represent the observable states, tamperable states and the flow transmit pairs (§5.3.2) of the adversary.

## 5.7.1 Speculative Examples

Table 5.1 shows the list of transient execution attacks that we check for observational determinism when executing on the SAP model. The execution column indicates the setting in which the vulnerabilities may occur: either when the adversary executes asynchronously (e.g., using simultaneous multi-threading) or only before and after the victim program execution (entry points). We try to choose asynchronous execution whenever possible because it is a more restrictive model. The three columns $V_O$, $V_T$, and $F$ describe our parameterization of the adversary model (§5.3.2) used to capture the attacks. $V_O$ shows two covert channels that are used in the list of attacks, which include the L1D cache and the AVX2 power state. $V_T$ indicates the states that we consider tamperable. Lastly, $F$ describes the adversary's capability to leak information from the state in column $v_1$ to the state in column $v_2$. For example, in the case of the LazyFP vulnerability, we assume the setting in which the adversary can leak information from the XMM registers to the L1D cache, and for the MDS-based attacks, we assume that the adversary can leak information from the line fill buffer (`LFB`) to the L1D cache. The remaining column indicates the speculative features being exploited at a high level. We note that these columns are not intended to be exhaustive of the speculative features, buffers, and covert channels that can potentially be exploited. We emphasize that the table stresses the need for a more holistic approach that considers all components of the platform and an adversary model that can parameterize the system state. Each vulnerability shown only exploits a specific combination of speculative features and covert channels, yet there is potentially a combinatorial space of features one could exploit and thus sound analysis would require reasoning about all these combinations in a scalable manner.

**Verifying Observational Determinism on the SAP Model**

| **Vulnerability** | **OD** (BMC) | | **TOD** (BMC) | | **TOD** (Interpolants) | |
|---|---|---|---|---|---|---|
| | Insecure | Partitioned Cache | Insecure | Partitioned Cache | Insecure | Partitioned Cache |
| Spectre v1 | 150.7 | 125.9 | 7.1 | 35.3 | 5.2 | 6.2 |
| Spectre v2 | 223.0 | 242.8 | 8.2 | 25.3 | 4.2 | 4.3 |
| Spectre v4 | 20.8 | 49.2 | 5.6 | 6.872 | 4.2 | 4.0 |
| Meltdown | 12.7 | 92.1 | 4.1 | 4.1 | 3.8 | 3.7 |
| Foreshadow | 11.9 | 81.7 | 4.5 | 9.3 | 3.9 | 3.8 |
| LVI | 15.1 | 33.6 | 7.0 | 5.1 | 4.0 | 4.3 |
| NetSpectre | 17.6 | - | 7.0 | - | 3.7 | - |
| LazyFP | 6 | - | 3.7 | - | 3.8 | - |
| RIDL | 14.37 | 20.2 | 4.3 | 3.5 | 4.3 | 3.9 |
| Spectre v1 (St.=4) | TLE | TLE | 94.7 | 155.0 | 4.8 | 4.3 |
| Spectre v1 (St.=5) | TLE | TLE | 466.2 | 297.2 | 5.8 | 4.7 |
| Spectre v1 (St.=6) | TLE | TLE | 874.8 | TLE | 6.1 | 5.7 |

Table 5.2: Time (in seconds) to verify OD using the 2-safety encoding with BMC, the trace property encoding of OD (TOD) with BMC and TOD with interpolants. Examples are checked using UCLID5, and marked with TLE (time limit exceeded) if it takes longer than 15 minutes.

### 5.7.2 Verification Results

We model instruction-level program snippets representative of the transient execution attack vulnerabilities from Table 5.1 and verify the programs by composing it with the SAP. The results of the approaches used are described in Table 5.2. Specifically, we first verify that the program snippets violate the secure speculation property using (1) bounded model checking (BMC) with the 2-safety observational determinism-based encoding (OD), (2) using bounded model checking with the symbolic taint analysis (TOD) encoding (which is a trace property with taint contexts modeled using ghost variables as explained in §5.5.1) and (3) using the interpolant-based approach with the symbolic taint analysis encoding. For the interpolant-based approach, we prove pre-post properties corresponding to the taint context. These results are listed under the *Insecure* columns. We also verify that the programs are secure with an abstract partitioned cache [113] model, with the exception of NetSpectre and LazyFP because the model of the former leaks to the AVX2 side-channel and the latter leaks secrets into cache after the execution of the victim. These results are listed under the columns labeled *Partitioned Cache*. We note that the disparity in runtimes between the different vulnerabilities can be explained by the number of atomic blocks we separate the program $P$ into (and hence require more steps for BMC), the number of instructions and varying platform model complexity. The last three examples are extensions of the Spectre v1 attack where the system takes a varying number of steps (annotation (St.=$i$) means the system takes $i$ steps). As expected, the interpolant-based approach outperforms BMC because each check is localized to small sequences of instructions. The experiments were run on a 2.6 GHz 6-Core Intel Core i7 machine with 16 GB RAM.

## 5.8 Discussion

### 5.8.1 Limitations

**Compositional Verification of CondTPOD**. While CondNI (Eq. 5.10) and CondObsDet (Eq. 5.11) can be checked by checking each Floyd-Hoare triple locally, CondTPOD (Eq. 5.12) requires checking trace properties over entire traces. Consequently, this means we lose the ability to check each Floyd-Hoare triple locally for general trace properties. However, trace properties that can be expressed as invariants over the triples naturally does allow our proof system to also check whether the system satisfies the trace properties in a localized manner. In fact, many useful properties, including the ones used to instantiate TPOD to derive secure speculation [46], can indeed be written as invariants (e.g. whether a program is allowed to speculate). Thus we defer the exploration of compositionally checking trace properties to existing and future work.

**In-order Execution of Programs**. Similar to existing approaches [46, 68, 82] that use symbolic execution, our verification methodology using the SAP model only considers in-order program instruction fetch. Thus, there may exist vulnerabilities on an out-of-order microprocessor that are not captured using our model alone. Proving properties about fully out-of-order processors would thus require modeling a component akin to a reorder buffer. Alternatively, one could potentially synthesize sound abstractions such that any violation of an information flow property in the out-of-order implementation model is preserved by the abstraction.

**Soundness of Abstractions**. While the SAP model is capable of capturing a broad class of attacks, we emphasize that every component is necessary for sound verification of any system. Thus while our SAP model is capable of capturing a broad class of vulnerabilities on a class of micro-architectures, ideally each micro-architecture should tailor the model to accommodate all components that could potentially be exploited. For more fine-grain analysis, one should use sound abstract models derived from the RTL implementation, as direct formal verification of RTL often does not scale.

## 5.9 Related Work

Our main contribution in this work is demonstrating two forms of composition: (a) temporal composition which builds Hoare-style proofs for sequences of instructions, and (b) spatial composition which allows reasoning only over relevant slices of the hardware design. This is most closely related to work on information flow checking; more specifically, lazy self-composition [227]. The SymboTaint proof system used in temporal composition combines symbolic state with taint-based relational atoms. Lazy self-composition develops an abstraction-refinement approach, also using relational atoms. However, their core focus is on performing symbolic reasoning lazily (by default relying only on taint-based relational atoms). In contrast, our focus is on proof decomposition. Consequently, a lazy self-composition-based approach can be used with ours for identifying optimal interpolants. Additionally, lazy self-composition does not consider the modeling and compositional reasoning of the hardware platform.

Other related works that combine program and platform models to verify security include Covern [154] which defines composition over a specific type of shared resource system with locks, compositional information flow-aware refinement [23] which introduces the notion of ignorance-preservation, is developed over an abstract system which can be used our formalisms to develop more accurate models. Lastly, work on modeling hardware platforms using happens-before graphs [153, 209] proposes a pattern-based approach for checking security, however, non-interference is beyond the scope of this work.

The emergence of transient execution attacks has also led to the use of information flow checking for proving the security of programs [81, 82, 83], but they are limited in their ability to extend to different attacker and platform models and lack a systematic method of spatial composition. While previous work [68] provides a systematic approach to combine speculative attacks, they are attack-centric and retrospective, requiring knowledge about the precise attack mechanisms, and are limited in their ability to combine attacks.

## 5.10 Summary

In this work, we considered the problem of verifying information-flow-based security properties for software running on hardware platforms. This is challenging owing to complex microarchitectural-level system models and vulnerable code fragments nestled within large software. We introduced *SymboTaint*, a proof-system that specializes Hoare-style reasoning to properties such as non-interference and observational determinism. We developed *Information Flow State Machines* as an operational framework that allows parameterizable modeling of microarchitectural features. Additionally, IFSMs allow instrumenting the platform model with *SymboTaint* based proofs. We presented an abstract model of a speculative microprocessor called the *Speculative Abstract Platform* (SAP) with several microarchitectural features. We use our methodology to verify observational- determinism for a broad class of transient execution attacks beyond what is possible with existing approaches.

# Chapter 6

# Designing Secure and Efficient Trusted Execution Environments

While proving properties such as non-interference, observational determinism, secure speculation, and TPOD described in chapters 4 and 5 can assist in proving memory isolation. It is sometimes impractical to prove this on-demand for programs that are constantly changing on a platform. For these programs that require additional security, often in the form of data confidentiality, secure computation is desirable. One method to provide secure computation is through the use of TEEs. However, TEEs are limited in the number of features they support and their programming model. More importantly, TEEs support little to no memory sharing for the enclave programs they manage, which is one of their most inhibiting limitations.

In this chapter, we present the first formally verified TEE design to support memory sharing to expand the use case of enclave programs. Together with the previous chapters, this aims to provide secure information flow from the hardware level, to the firmware and software levels of hardware platforms.

## 6.1 Overview

The hardware enclave [10, 53, 70, 111, 112, 127, 141] is a promising method of protecting a program [157, 171, 172, 210] by allocating a set of physical addresses accessible only from the program. The key idea of hardware enclaves is to isolate a part of physical memory by using hardware mechanisms in addition to a typical memory management unit (MMU). The isolation is based on a *disjoint memory assumption*, which constrains each of the isolated physical memory regions to be owned by a specific enclave. A hardware platform enforces the isolation by using additional in-memory metadata and hardware primitives. For example, Intel SGX maintains per-physical-page metadata called the Enclave Page Cache Map (EPCM) entry, which contains the enclave ID of the owner [52]. The hardware looks up the entry for each memory access to ensure that the page is accessible only when the current enclave is the owner.

However, the disjoint memory assumption also significantly limits enclaves in terms of their performance and programmability. First, the enclave needs to go through an expensive initialization whenever it launches because the enclave program cannot use shared libraries in the system nor clone from an existing process [131]. Each initialization consists of copying the enclave program into the enclave memory and performing *measurements* to stamp the initial state of the program. The initialization latency proportionally increases depending on the size of the program and the initial data. Second, the programmer needs to be aware of the non-traditional assumptions about memory. For instance, system calls like `fork` or `clone` no longer rely on efficient copy-on-write memory, resulting in significant performance degradation [171, 210].

A few studies have proposed platform extensions to allow memory sharing of enclaves. Yu *et al.* [231] proposes Elasticlave, which modifies the platform such that each enclave can own multiple physical memory regions that the enclave can selectively share with other enclaves. An enclave can map other enclaves' memory regions to its virtual address space by making a request, followed by the owner granting access. Elasticlave improves the performance of enclave programs that relies on heavy inter-process communication (IPC). Li *et al.* [131] proposes Plug-In Enclave (PIE), which is an extension of Intel SGX. PIE enables faster enclave creation by introducing a *shared enclave region*, which can be mapped to another enclave by a new SGX instruction `EMAP`. `EMAP` maps the entire virtual address space of a pre-initialized *plug-in* enclave. PIE improves the performance of enclave programs with large initial code and read-only data (e.g., serverless workloads). Although the prior work shows that memory sharing can substantially improve performance, they do not provide formal guarantees about security.

Unsurprisingly, the disjoint memory assumption of enclaves is crucial for the security of the enclave platforms. Previous studies [70, 156, 184, 204] formally prove high-level security guarantees of enclave platforms such as non-interference properties, integrity, and confidentiality based on the disjoint memory assumption. However, to our best knowledge, no model formally verifies the security guarantees under the weakened assumption that the enclaves can share memory.

Practical formal verification requires choosing the right level of abstraction to model and apply automated reasoning. Verification on models that conform to the low-level implementation [156] or source-level code [8, 47, 114, 194, 198] is often platform-specific in that it only provides security guarantees to those implementations and thus does not apply generally. If one seeks to verify that a memory-sharing approach on top of a family of enclave platforms is secure, it is not easy to reuse verification efforts for specific implementations. We seek an approach that is incremental and also applicable to existing platforms.

Moreover, there are many ways one could design a memory-sharing model, each varying in complexity and flexibility. Complex models can provide more flexibility to optimize the applications for performance, but this often comes at the cost of increasing the complexity of formal verification. However, if memory sharing is too restrictive, it also becomes hard for programmers to leverage it for performance improvements. Thus, we seek a simple sharing model with a balance between flexibility and ease of verification.

To this end, this chapter presents *Cerberus*, a formal approach to secure and efficient enclave memory sharing. Cerberus chooses *single-sharing* model with *read-only* shared memory, which allows each enclave to access only one read-only shared memory. We show that this design decision significantly reduces the cost of verification by simplifying invariants, yet still provides a big performance improvement for important use cases. We formalize an enclave platform model that can accurately capture high-level semantics of the extension and formally verify a property called *Secure Remote Execution* (SRE) [204]. We perform *incremental verification* by starting from an existing formal model called Trusted Abstract Platform (TAP) [204] for which the SRE property is already established. Finally, we show the feasibility of Cerberus by implementing it in an existing platform, RISC-V Keystone [127]. Cerberus can substantially reduce the initialization latency without incurring significant computational overhead.

To summarize, the contributions of this chapter include the following:

- Provide a *general* formal enclave platform model with memory sharing that weakens the disjoint memory assumption and captures a family of enclave platforms
- Formally verify that the modified enclave platform model satisfies SRE property via automated formal verification
- Provide programmable interface functions that can be used with existing system calls
- Implement the extension on an existing enclave platform and demonstrate that Cerberus reduces enclave creation latency

## 6.2 Motivation and Background

### 6.2.1 Use Cases of Memory Sharing in Enclaves

Many programs these days take advantage of sharing their memory with other programs. For example, *shared libraries* allow a program to initialize faster with less physical memory than static libraries because the operating system can reuse in-memory shared libraries for multiple processes. Similarly, sharing large in-memory objects (e.g., an in-memory key-value store) can be shared across multiple processes. Running a program inside an enclave disables memory sharing because of the disjoint memory assumption. This section introduces a few potential use cases of memory sharing in enclaves to motivate Cerberus. Memory sharing can significantly improve the performance of enclave programs that require multiple isolated execution contexts with shared initial code and data.

**Serverless workloads**. Serverless computing is a program execution model where the cloud provider allocates and manages resources for a function execution on demand. In the model, the program developer only needs to write a function that runs on a language runtime, such as a specific version of Python. Many serverless frameworks [71, 160] reduce the cold-start latency of the execution with pre-initialized *workers* containing the language runtime. As described earlier by Li *et al.* [131], the workers will suffer from an extremely long initialization latency (e.g., a few seconds) when they run in enclaves, as the language runtimes are typically a few megabytes (e.g., Python is 4 MB). Because the difference between worker memories (e.g., heap, stack, and the function code) can be as small as a few kilobytes, a large amount of initialization latency and memory usage can be saved by sharing memory.

**Inference APIs**. Machine learning model serving frameworks [13, 90, 175] allow users to send their inputs and returns the model's inference results. Serving different users with separate enclaves will have a longer latency as the model size increases. As of now, the five most popular models in Huggingface [90] have a number of parameters ranging from a few hundred million to a few hundred billion, which would occupy at least hundreds of megabytes of memory. Sharing memory will drastically reduce the latency and memory usage of such inference APIs in enclaves.

**Web servers**. Multi-processing web servers handle requests with different execution contexts while sharing the same code and large objects. For example, a web server or an API server that provides read access to a large object (e.g., front-end data or database) will suffer from long latency and memory usage running in an enclave. If enclaves can share a memory, they can respond with lower latency and smaller memory usage.

### 6.2.2 The Secure Remote Execution Property

As mentioned earlier, much of the prior work identifies integrity and confidentiality as key security properties for enclave platforms. As a result, we aim to prove a property that is at least as strong as these two, which is the SRE property [204]. To provide intuition behind the property, the typical setting for an enclave user is that the user wishes to execute their enclave program securely on a remote enclave platform. The remote platform is largely untrusted, with an operating system, a set of applications, and other enclaves that may potentially be malicious. Thus it is desirable to create a secure channel between the enclave program and the user in order to set up the enclave program securely. Consequently, in order to have end-to-end security, we need to ensure that the enclave platform behaves in the following three ways:

- The measurement of an enclave on the remote platform can guarantee that the enclave is set up correctly and runs in a deterministic manner
- Each enclave program is integrity-protected from the untrusted entities and thus executes deterministically,
- Each enclave program is confidentiality-protected to avoid revealing secrets to untrusted entities.

These three behaviors manifest as the secure measurement, integrity, and confidentiality properties as defined in Section §6.5 and are ultimately what we guarantee for our platform model extended with Cerberus.

### 6.2.3 Formal Models of Enclave Platforms

Prior work has formally modeled and verified enclave platform models for both functional correctness and adherence to safety properties similar to the SRE property. While verification at the source code level (e.g., Komodo [70]) provides proofs of functional correctness and noninterference of enclaves managed by a software *security monitor*, existing verification efforts are often closely tied to the implementation, making it difficult to apply existing work to our extension. A binary- or instruction-level verification (e.g., Serval [156]) on the other hand, focuses on automating the verification of the implementation. Working with binary-level models is often difficult and tedious because the binary often lacks high-level program context (e.g., variable names). However, this chapter aims to verify the enclave memory sharing on general enclave platforms and thus binary-level verification is not a goal. Our approach complements binary-level verification by reducing the problem to showing refinement from our model the the binary-level model.

The *Trusted Abstract Platform model* [204] is an abstraction of enclave platforms that was introduced with the SRE property. The SRE property states that an enclave execution on a remote platform follows its expected semantics and is confidentiality-protected from a class of adversaries defined along with the TAP model. This property provides the end-to-end verification of integrity and confidentiality for enclaves running on a remote platform. It has also been formally proven that the state-of-the-art enclave platforms such as Intel's SGX [52, 141] and MIT's Sanctum [53, 124] refine the TAP model and hence satisfy SRE against various adversary models. To our best knowledge, the TAP is the only model for formal verification that has been used to capture enclave platforms in a general way. The level of abstraction also makes it readily extensible. For these reasons, we extend the TAP model.

## 6.3 Design Decisions: Memory Sharing in Enclave Platforms

Several design decisions were made in our approach to conform to our design goals. The memory-sharing model and interface designs are crucial for modeling, verification, and implementation. This section discusses the details of how we chose to design the memory-sharing model and interface.

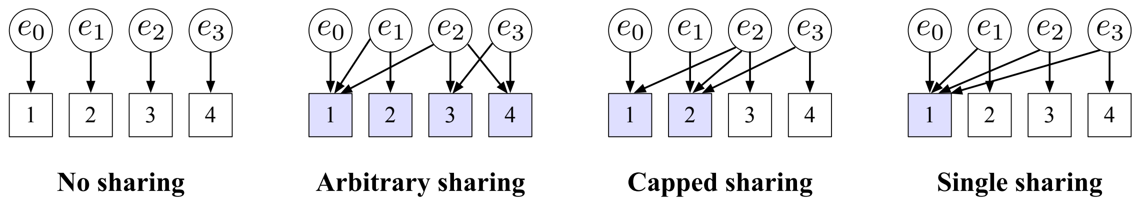


Figure 6.1: Memory sharing models with varying flexibility. Blue (and white) boxes indicate shareable (and non-shareable) physical memory regions, and circles indicate enclaves. An edge from an enclave to physical memory is an *access relation* stating that an enclave can access the memory it points to. The figure only depicts cases where the number of memory regions $m$ is the same as that of enclaves $n$, but $m$ can be greater than $n$ in practice.

## 6.3.1 Writable Shared Memory

Some programs use shared memory for efficient inter-process communication (IPC), which requires any writes to the shared memory to be visible to the other processes. Elasticlave [231] allows an enclave to grant write permissions for a memory region to the other enclaves such that they can communicate without encrypting or copying the data. However, the authors also show that such *writable shared memory* requires the write permission to be dynamically changed to prevent interference between enclaves. As formal reasoning on memory with dynamic permission will introduce a non-trivial amount of complexity, we leave this direction as future work. Thus, Cerberus does not support use cases based on IPC or other mutable shared data. Similarly, PIE [131] also only enables read-only memory sharing among enclaves.

## 6.3.2 Memory Sharing Models

Figure 6.1 shows four different memory-sharing models with varying levels of flexibility. We discuss the implications for the implementation and the feasibility of formal verification for each model. For this discussion, we use the number of *access relations* between enclaves and memory regions as a metric for the complexity of both verification and implementation.

**No sharing**. We refer to the model that assumes the disjoint memory assumption as the *no-sharing* model, which is implemented in state-of-the-art enclaves [70, 127, 141]. The no-sharing model strictly disallows sharing memory and assigns each physical address to only one enclave. As a result, the number of access relations is $O(max(m, n)) = O(m)$, where $m$ is the number of physical memory regions, and $n$ is the number of enclaves. Thus, implementations with no-sharing will require metadata scaling with $O(m)$ to maintain the access relations. For instance, each SGX EPC page has a corresponding entry in EPCM, which contains the owner ID of the page. The no-sharing model has been formally verified at various levels [70, 156, 204].

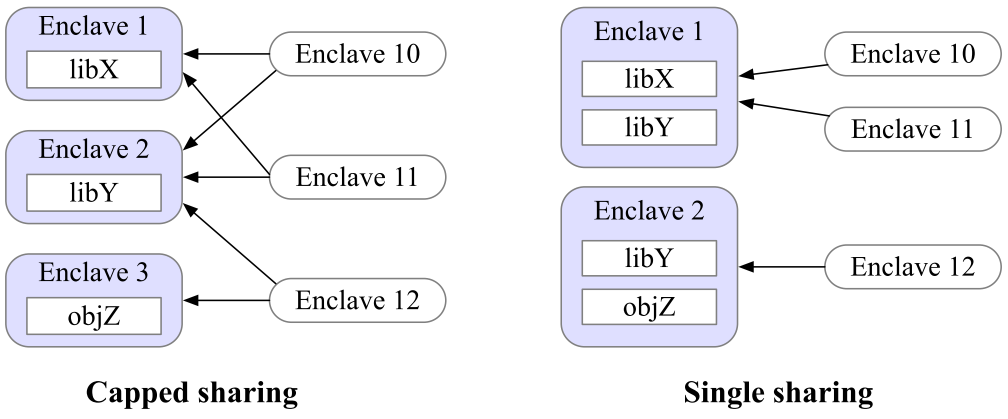


Figure 6.2: Difference between capped- and single-sharing models in use cases. `libX`, `libY`, and `objZ` are the large libraries or objects that enclaves want to share. Enclave 10 and 11 relies on `libX` and `libY`, while Enclave 12 relies on `libY` and `objZ`.

**Arbitrary sharing**. One can completely relax the sharing model and allow any arbitrary number of enclaves to share memory (as in Elasticlave). We refer to this sharing model as the *arbitrary-sharing* model. In this case, the number of access relations between enclaves is $O(mn)$. Consequently, arbitrary sharing requires metadata scaling with $O(mn)$.

**Capped sharing**. To achieve scalability in the number of access relations, one can constrain the sharing policy such that each enclave can only access a limited number of shared physical memory regions. We refer to this sharing model as the *capped-sharing* model. In Figure 6.1, capped sharing shows an example where each enclave is only allowed to access at most two additional shared physical memory regions. As an example, PIE [131] introduces a new type of enclave called *plug-in* enclave, which can be mapped to the virtual address space of a normal enclave. This reduces the number of relations to $O(kn+m)$, where $k$ is the number of shared physical memory regions that are allowed to be accessed by an enclave.

**Single sharing**. The *single-sharing* model is a special case of the capped sharing with $k = 1$. Thus, the model only allows enclaves to access the shared memory regions of a particular enclave. Single sharing reduces the complexity to $O(m)$.

## 6.3.3 Formal Verification of Sharing Models

Formally verifying arbitrary- or capped-sharing models are challenging due to the flexibility of the models. Verifying security properties such as SRE requires reasoning about safety properties with multiple traces and platform invariants with nested quantifiers. In our experience, modeling an arbitrary number of shared memory would add to this complexity. For example, one inductive invariant needed to prove SRE on TAP is that if a memory region is accessible by an enclave, the region is owned by the enclave. To allow an arbitrary number of memory regions to be shared, the invariant should be extended such that it existentially quantifies over all relations, for example, stating that the owner of the memory is one of the enclaves that shared their memory with the enclave (See §6.5 and Eq. (6.8) for details). The encoding of the invariant in TAP uses first-order logic with the theory of arrays and, in general, is not decidable [27]. As a result, the introduction of this quantifier further complicates the invariant. Despite the limiting constraint in *capped sharing*, a formal model capturing any arbitrary limit $k$ would still require modeling an arbitrary number of the shared memory as in the *arbitrary sharing* scheme and face the same complication.

In contrast, the single-sharing model significantly reduces the efforts of formal reasoning and implementation. First, the formal reasoning no longer requires the complex invariant because the memory accessible to an enclave either belongs to the enclave itself or only another enclave that is sharing memory. Second, the implementation becomes much simpler as it requires only one per-enclave metadata to store the reference to the shared memory. The platform modification also becomes minimal as it only checks one more metadata per memory access.

Despite its simplicity, the single-sharing model can still improve the performance of programs by having all of the shared contents (e.g., shared library, initial code, and initial data) in a shared enclave. Figure 6.2 depicts the difference between capped- and single-sharing models. With the capped-sharing model, each shareable content can be initialized with a separate enclave, allowing each enclave to map up to $k$ different enclave memory regions (i.e., *Plug-In* enclaves in PIE). Single-sharing model only allows each enclave to map exactly one other enclave, leaving potential duplication in memory when heterogeneous workloads have shared code (e.g., `libY`). We claim that the benefit of the model's simplicity outweighs the limitation, as the single sharing does not have notable disadvantages over capped-sharing when there is no common memory among heterogeneous workloads.

## 6.3.4 Interface

Enclave programs need interface functions to share memory based on the sharing model. Elasticlave and PIE introduce explicit operations to *map* or *unmap* the shareable physical memory region to the virtual address space of the enclave. For example, Elasticlave requires an enclave program to explicitly call `map` operation to request access to the region, which will be approved by the owner via `share` operation. Similarly, PIE allows an enclave to use `EMAP` and `EUNMAP` instructions to map and unmap an entire plug-in enclave memory to the virtual address of the enclave.

Elasticlave and PIE allow an enclave program to map shareable physical memory regions to its virtual address space. However, there are a few downsides to the approaches. First, the programmers must manually specify which part of the application should be made shareable. In most cases, the programmers must completely rewrite a program such that the shareable part of the program is partitioned into a separate enclave memory. Second, a dynamic map or unmap requires local attestation, which verifies that the newly-mapped memory is in an expected initial state. Thus, the measurement property of a program relies on the measurement property of multiple physical memory regions.

Cerberus takes an approach similar to a traditional optimization technique, which clones an address space with copy-on-write, as in system calls like `clone` and `fork`. This approach fits Cerberus use cases where the shareable regions include text segments, static data segments, and dynamic objects (e.g., a machine learning model). In general, programmers expect such system calls to copy the entire virtual address space of a process – no matter what it contains – to a newly-created process. A similar interface will allow the programmers to write enclave programs with the same expectation. Also, such an interface will not require additional properties or assumptions on measurements of multiple enclaves. Since the initial code of an enclave already contains when to share its entire address space, the initial measurement implicitly includes all memory contents to be shared.

To this end, Cerberus introduces two enclave operations, which are `Snapshot` and `Clone`. `Snapshot` freezes the entire memory state of an enclave, and `Clone` creates a logical duplication of an enclave. We make `Snapshot` only callable from the enclave itself, allowing the enclave to decide when to share its memory. The adversary can call `Clone` any time, which does not break the security because it can be viewed as a special way of launching an enclave (See §6.4.4). When the adversary calls `Clone` on an existing enclave, a new enclave is created and resumes with a copy-on-write (CoW) memory of the snapshot. Thus, any changes to each of the enclaves after the `Clone` are not visible to each other. The following sections formally discuss the sharing model and the interface of Cerberus.

## 6.4 Formal Model of An Enclave Platform with Memory Sharing

We first introduce a threat model in Section §6.4.1 that is consistent with these goals and the current state-of-the-art enclave threat models. Then, we list and justify our assumptions in Section §6.4.2, introduce our formal models of the platform and adversary based on these assumptions in Section §6.4.3 and then introduce the two new operations `Snapshot` and `Clone` of Cerberus in Section §6.4.4.

In section §6.5, we use these formal models to define the SRE [204] property, which is a critical security property used to prove that enclaves executing in the remote platform are running as expected and confidentially. These properties are then formally verified using incremental verification on TAP. In other words, our formal models extend the TAP model introduced by Subramanyan *et al.* [204]. While SRE has been proven on the extended TAP model, Cerberus design weakens the disjoint memory assumption to allow memory sharing. In addition, it is not immediately clear that the two additional operations clearly preserve SRE. Thus, we prove that SRE still holds under our extended model with the operations. For the rest of the literature, we refer to the original formal platform model defined by Subramanyan *et al.* [204] as TAP and our extended model as $\text{TAP}_C$.

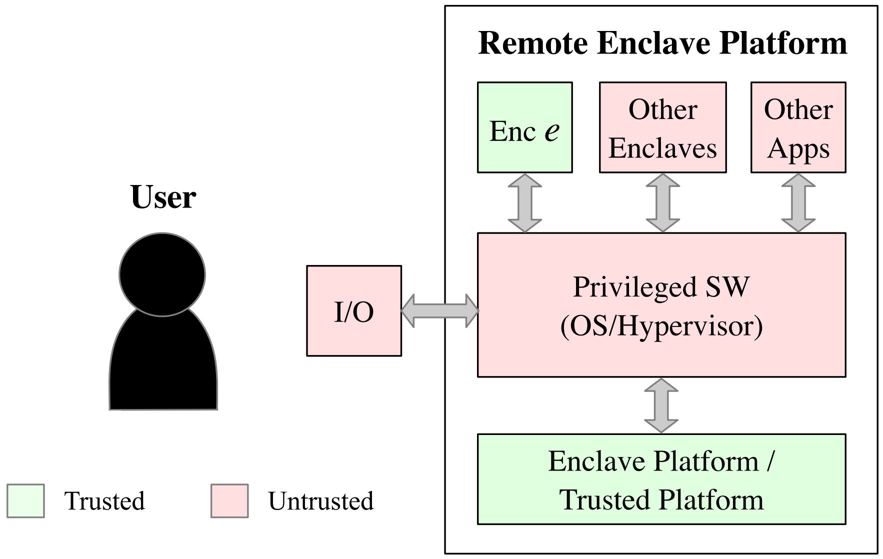


Figure 6.3: A user provisions their (protected) enclave $e$ in the remote enclave platform isolated from untrusted software. Green/red boxes indicate trusted/untrusted components.

### 6.4.1 Threat Model

Our extension follows the typical enclave threat model where the user's enclave program $e$ is integrity- and confidentiality-protected over the enclave states (e.g. register values and data memory owned by the enclave program) against any software adversary running in the remote enclave platform. The software adversaries of an enclave include the untrusted operating system, user programs, and the other enclaves as shown in Figure 6.3.

With Cerberus, enclaves may share data or code that were common between enclaves before the introduction of the `Clone`. We assume that the memory is implicitly not confidential among these enclaves with shared memory. However, each enclave's memory should not be observable by the operating system or other enclaves and applications. We ensure that the enclaves are still *write-isolated*, which means that any modification to the data from one enclave must not be observable to the other enclaves, even to the enclave that it cloned from. Thus, any secret data needs to be provisioned after the enclave is cloned. It is the enclave programmer's responsibility to make sure that the parent enclave does not contain any secret data that can be leaked through the children.

We do not consider the program running in the enclave to be vulnerable or malicious by itself. For example, a program can generate a secret key in the shared memory, and encrypt the confidential data of the child with the key. This would break confidentiality among children enclaves write-isolated from each other because the children will have access to the key in the shared memory. We do not consider such cases, but this could be easily solved by having programs load secrets to their memory after they have created the distrusting children.

Since our main goal is to design a generic extension, we also do not consider any type of side-channel attack or architecture-specific attack [41, 116, 126, 133, 140, 151, 186, 191, 215, 216, 219, 226]. We leave side-channel resilient interface design as future work. We note that since the base TAP model has also been used to prove side-channel resiliency on some enclave platforms [53, 127], it is not impossible to extend our proofs to such adversary models. Denial of service against the enclave is also out of the scope in this thesis; this is consistent with the threat models for existing state-of-the-art enclave platforms.

A formal model of the threat model is described in more detail in Section §6.4.3 after the formal definition of the platform.

### 6.4.2 TAP$_C$ Model Assumptions

Below, we summarize a list of assumptions about the execution model of TAP$_C$ that we make for the purpose of simplifying and abstracting the modeling. These assumptions are consistent with the adversary model described above:

- TAP$_C$ inherits the assumptions and limitations of TAP [204], which include assuming that every platform and enclave operation is atomic relative to one another, assuming the DRAM is trusted, no support for demand paging, assuming a single-core and single-process model, and assuming properties of cryptographic functions used for measurement.
- If an enclave operation returns with an error code, we assume that the states of the platform are entirely reverted to the state prior to the execution of that operation.
- State continuity of enclaves is out-of-scope in our models, consistent with prior work TAP [204], and can be addressed using alternative methods [106, 163].
- The memory allocation algorithm (e.g. for copy-on-write) is deterministic given that the set of unallocated memory is the same. This means that given any two execution sequences of a platform, as long as the page table states are the same, the allocation algorithm will return the same free memory location to allocate.

Next, we introduce our formal models describing the platform which extends the existing TAP model with `Snapshot` and `Clone` under these assumptions.

### 6.4.3 Formal TAP$_C$ Platform Model Overview

As mentioned, a user of an *enclave platform* typically has a program and data that they would like to run securely in a remote server, isolated from all other processes as shown in Figure 6.3. Such a program can be run as an enclave $e$. The remote server provides isolation using its hardware primitives and software for managing the enclaves, where the software component is typically firmware or a security monitor. This software component provides an interface for the enclave user through a set of operations, denoted by $\mathcal{O}$, for managing

$e$. The goal is to guarantee that this enclave $e$ is protected from all other processes on the platform and running as expected. For the purpose of understanding the proofs, we refer to the enclave we would like to protect as the *protected enclave* $e$. We make this distinction to differentiate it from adversary-controlled enclaves.

### Platform and Enclave State

The platform can be viewed as a transition system $M = \langle S, I, \rightsquigarrow \rangle$ that is always in some state denoted by $\sigma \in S$. Alternatively, $\sigma$ can be viewed as an assignment of values to a set of state variables $V$. The platform starts in an initial state in the set $I$ and transitions between states defined by a transition relation $\rightsquigarrow \subset S \times S$. We write $(\sigma, \sigma') \in \rightsquigarrow$ to mean a valid transition of the platform from $\sigma$ to $\sigma'$. An execution of the platform therefore emits a (possibly infinite) sequence of states $\pi = \langle \sigma^0, \sigma^1, ... \rangle$, where $(\sigma^i, \sigma^{i+1}) \in \rightsquigarrow$ for $i \in \mathbb{N}$. We write $\pi^i = \sigma^i$ interchangeably, but will usually write $\pi^i$ whenever referencing a specific trace. When an enclave is initially launched, it is in the initial state prior to enclave execution, which we indicate using the predicate $init_e(\sigma) : S \rightarrow Bool$. We describe the set of variables $V$ and enclave state $E_e(\sigma)$ for TAP$_C$ in the following Section §6.4.3.

### TAP$_C$ State Variables

Each of the variables $V$ in TAP$_C$ are shown in Table 6.1. $pc : VA$[*] is an abstraction of the program counter whose value is a virtual address from the set of virtual addresses $VA$. $\Delta_{rf} : \mathbb{N} \rightarrow W$ is a register file that is a map[†] from the set of register indices (of natural numbers) $\mathbb{N}$ to the set of words $W$. $\Pi : PA \rightarrow W$ is an abstraction of memory that maps the set of physical addresses $PA$ to a set of words. We write $\Pi[a]$ to represent the memory value at a given physical address $a \in PA$. A page table abstraction defines the mapping of virtual to physical addresses $a_{PA}$ and access permissions $a_{perm} : VA \rightarrow ACL$, where $ACL$ is the set of read, write, and execute permissions. $ACL$ can be defined as the product $VA \rightarrow Bool \times Bool \times Bool$, where $Bool \doteq \{true, false\}$ and the value of the map corresponds to the read, write, and execute permissions for a given virtual address index[‡]. $e_{curr} : \mathcal{E}_{id}$ represents the current enclave that is executing. $\mathcal{E}_{id} = \mathbb{N} \cup \{\mathcal{OS}\} \cup \{e_{inv}\}$ is the set of enclave IDs represented by natural numbers and a special identifier $\mathcal{OS}$ representing the untrusted operating system. We reserve the identifier $e_{inv}$ to refer to the invalid enclave ID which can be thought of as a default value that does not refer to any valid enclave. For the ease of referring to whether an enclave is valid and launched, we define the predicate $valid(e_{id}) \doteq e_{id} \neq e_{inv} \wedge e_{id} \neq \mathcal{OS}$ that returns whether or not an ID is a valid enclave ID. The *active* predicate returns true for an enclave $e$ whenever it is launched or cloned and not yet destroyed in state $\sigma$. $o$ is a map that describes the ownership of physical addresses, each

[*]We write $v$: $T$ to mean variable $v \in V$ has type $T$

[†]of type $L \rightarrow R$, where the index type is $L$ and value type is $R$.

[‡]We use $\doteq$ to mean *by definition* to differentiate between the *equality* symbol $=$.

| **State Var.** | **Type** | Description |
|---|---|---|
| $pc$ | $VA$ | The program counter. |
| $\Delta_{rf}$ | $\mathbb{N} \rightarrow W$ | General purpose registers. |
| $\Pi$ | $PA \rightarrow W$ | Physical memory. |
| $a_{PA}$ | $VA \rightarrow PA$ | Page table abstraction; virtual to physical address map. |
| $a_{perm}$ | $VA \rightarrow ACL$ | Page table abstraction; virtual to their permissions. |
| $e_{curr}$ | $\mathcal{E}_{id}$ | Current executing enclave ID (or $e_{curr} = \mathcal{OS}$ if the OS is executing). |
| $o$ | $PA \rightarrow \mathcal{E}_{id}$ | Map from physical addresses to the enclave that owns it. |
| $\mathcal{M}$ | $\mathcal{E}_{id} \rightarrow \mathcal{E}_M$ | Map of enclave IDs to enclave metadata. `emd`$[\mathcal{OS}]$ stores a checkpoint of the OS. |

Table 6.1: TAP$_C$ State Variables $V$.

of which can be owned by an enclave (with the corresponding enclave ID) or the untrusted operating system.

Lastly, each enclave $e$ has a set of enclave metadata $\mathcal{M}$, which is a record of variables described in Table 6.2. We abuse notation and write $\mathcal{M}^{pc}[e]$ to represent the program counter $\mathcal{M}^{pc}$ of $e$ in the record stored in the metadata map $\mathcal{M}$. We use the enclave index operator $[\cdot]$ similarly for the other metadata fields defined in Table 6.2 to refer to a particular enclave's metadata. $\mathcal{M}^{EP}[e]$ is the entry point of the enclave that the enclave $e$ starts in after the `Launch` and before `Enter`. $\mathcal{M}^{AM}_{PA}[e]$ is the virtual address map of the enclave program. $\mathcal{M}^{AM}_{perm}[e]$ is the map of address permissions for each virtual address. $\mathcal{M}^{EV}[e]$ is the map from virtual addresses to Boolean values representing whether an address is allocated to the enclave. $\mathcal{M}^{pc}[e]$ is the current program counter of the enclave. $\mathcal{M}^{regs}[e]$ is the saved register file of the enclave. $\mathcal{M}^{paused}[e]$ is a Boolean representing whether or not the enclave has been paused and is initially false at launch.

These variables were introduced in the base TAP model and are unmodified in TAP$_C$. We introduce the remaining four metadata variables required for Cerberus in Section §6.4.4, which are additional state variables in TAP$_C$ that are not defined in the base TAP model.

The state $E_e(\sigma)$ is a projection of the platform state to the enclave state of $e$ that includes $\mathcal{M}^{EP}[e]$, $\mathcal{M}^{AM}_{PA}[e]$, $\mathcal{M}^{AM}_{perm}[e]$, $\mathcal{M}^{EV}[e]$, $\mathcal{M}^{pc}[e]$, $\mathcal{M}^{regs}[e]$, and the projection of enclave memory $\lambda v \in VA.ITE(\mathcal{M}^{EV}[e][v], \Pi[\mathcal{M}^{AM}_{PA}[v]], \perp)$. In the last expression, $\lambda v \in VA.E$ is the usual lambda operator over the set of virtual addresses $v$ and expression body $E$, $ITE(c, expr_1, expr_2)$ is the if then else operator that returns $expr_1$ if condition $c$ is true and $expr_2$ otherwise. $\perp$ is the constant bottom value which can be thought of as a don't-care or unobservable value. This projection of memory represents all memory accessible to enclave $e$, including shared memory and memory owned by enclave $e$ as referenced by the virtual address map $\mathcal{M}^{AM}_{PA}$.

### Enclave Inputs and Outputs

Communication between an enclave $e$ and external processes for a given state $\sigma$ are controlled through $e$'s inputs $I_e(\sigma)$ and its outputs $O_e(\sigma)$. $I_e(\sigma)$ includes the arguments to the opera-

| State Var. | Type | Description of each field |
|---|---|---|
| $\mathcal{M}^{EP}$ | $VA$ | Enclave entrypoint. |
| $\mathcal{M}^{AM}_{PA}$ | $VA \to PA$ | Enclave's virtual address map. |
| $\mathcal{M}^{AM}_{perm}$ | $VA \to ACL$ | Enclave's address permissions. |
| $\mathcal{M}^{EV}$ | $VA \to Bool$ | Set of private virtual addresses. |
| $\mathcal{M}^{pc}$ | $VA$ | Saved program counter. |
| $\mathcal{M}^{regs}$ | $\mathbb{N} \to W$ | Saved registers. |
| $\mathcal{M}^{paused}$ | $Bool$ | Whether enclave is paused. |
| $\mathcal{M}^{IS\dagger}$ | $Bool$ | Whether the enclave is a snapshot. |
| $\mathcal{M}^{CC\dagger}$ | $\mathbb{N}$ | Number of children enclaves. |
| $\mathcal{M}^{RS\dagger}$ | $\mathcal{E}_{id}$ | Enclave's root snapshot. |
| $\mathcal{M}^{PAF\dagger}$ | $PA \to Bool$ | Map of free physical addresses. |

Table 6.2: Record of TAP$_C$ enclave metadata $\mathcal{E}_M$. Additional state variables were added to the TAP model support `Snapshot` & `Clone`, as indicated by the † superscript.

tions that manage enclave $e$, areas of memory outside of the enclave that the enclave may access and an untrusted attacker may write to, and randomness from the platform. $O_e(\sigma)$ contains the outputs of enclave $e$ that are writable to by $e$ and accessible to the attacker and the user.

## Platform and Enclave Execution

An execution of an enclave $e$ is defined by the set of operations from $\mathcal{O}$, in which the execution of an operation is deterministic up to its input $I_e(\sigma)$ and current state $E_e(\sigma)$. This means that given the same inputs $I_e(\sigma)$ and enclave state $E_e(\sigma)$, the changes to enclave state $E_e(\sigma)$ is deterministic. The set of operations for the base TAP model is $\mathcal{O}_{base} \doteq \{$ `Launch`, `Destroy`, `Enter`, `Exit`, `Pause`, `Resume`, `AdversaryExecute`$\}$. TAP$_C$ extends the base set with two additional operations: $\mathcal{O} \doteq \mathcal{O}_{base} \cup \{$`Snapshot`, `Clone`$\}$. We use the predicate $curr(\sigma) = e$ to indicate that enclave $e$, which may be adversary controlled, is executing at state $\sigma$ and $curr(\sigma) = \mathcal{OS}$ to indicate that the operating system is executing.

## Formal Adversary Model

In our model, untrusted entities such as the OS and untrusted enclaves are represented by an adversary $\mathcal{A}$ that can make arbitrary modifications to state outside of the protected enclave $e$, denoted by $A_e(\sigma)$. Consistent with the base TAP model, the untrusted entities and protected enclave $e$ takes turn to execute under interleaving semantics in our formal TAP$_C$ model, as illustrated in Figure 6.4. Under these semantics, the adversary is allowed to take any arbitrary number of steps when an enclave is not executing. Likewise, the protected enclave is allowed to take any number of steps when the adversary is not executing without being observable to the adversary.

Conventionally, we define an adversary with an observation and tamper function that describes what the adversary can observe and change in the platform state during its execution to break integrity and confidentiality. The execution of the adversary is the operation `AdversaryExecute` in $\mathcal{O}_{base}$ during which either the tamper or observation functions can be used by the adversary. Figure 6.4 describes these two functions for our model.

$$\begin{tikzcd}
\pi_1^0 \arrow[r, "op^0"] \arrow[d, phantom, "\approx_{\mathcal{L}}" rotate=90] & \cdots & \pi_1^i \arrow[r, "\mathcal{A}"] \arrow[d, phantom, "\approx_{\mathcal{L}}" rotate=90] & \pi_1^{i+1} \arrow[r, "op^{i+1}"] \arrow[d, phantom, "\approx_{\mathcal{L}}" rotate=90] & \pi_1^{i+2} \arrow[d, phantom, "\approx_{\mathcal{L}}" rotate=90] & \cdots \\
\pi_2^0 \arrow[r, "op^0"] & \cdots & \pi_2^i \arrow[r, "\mathcal{A}"] & \pi_2^{i+1} \arrow[r, "op^{i+1}"] & \pi_2^{i+2} & \cdots
\end{tikzcd}$$

Figure 6.4: Illustrating the execution of two traces of the platform in the secure measurement, integrity and confidentiality proofs. Proof obligations for each property are checked as indicated by $\approx_{\mathcal{L}}$ and equal initial condition indicated as $\approx_{\mathcal{L}}$. $op^i$ indicates enclave execution of an operation from $\mathcal{O}$ at step $i$ and $\mathcal{A}$ indicates an adversary execution.

**Tamper Function**. The tamper function is used to model these malicious modifications to the platform state by the adversary and is defined over $A_e(\sigma)$ which includes any memory location that is not owned by the protected enclave $e$ and page table mappings. The semantics of the model allows the adversary to make these changes whenever it is executing. We allow all tampered states to be unconstrained in our models, which means they can take on any value. This type of adversary tamper function over-approximates what the threat model can change and is typically referred to as a havocing adversary [46, 204].

**Observation Function**. The adversary's observation function is denoted $obs_e(\sigma)$. In our model, we allow the adversary to observe locations of the memory that are not owned by the protected enclave $e$, described by the set $obs_e(\sigma) \doteq O_e(\sigma) \doteq \lambda p \in PA.ITE(\sigma.o[p] \neq e, \sigma.\Pi[p], \perp)$. Intuitively, $obs_e$ is a projection of the platform state that is observable by the adversary whose differences should be excluded by the property. For example, if the same enclave program operating over different secrets reveals secrets through the output, that is a bug in the enclave program and we do not protect from this. The adversaries may try to modify or read the enclave state during the lifetime of the enclave.

Under this threat model, we prove that the TAP$_C$ model still satisfies the SRE property described in Section §6.5.

### 6.4.4 The Extended Enclave Operations

Cerberus is the extension of enclave platforms with two new operations `Snapshot` and `Clone` to facilitate memory sharing among enclaves. Intuitively, `Snapshot` converts the enclave executing the operation into a read-only enclave and `Clone` creates a child enclave from the

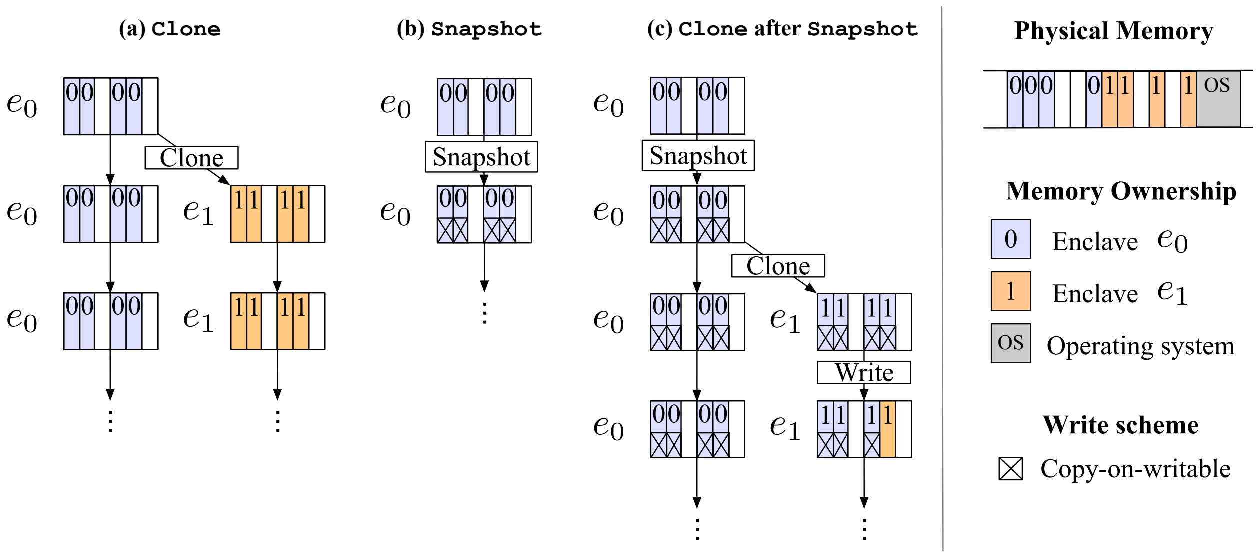


Figure 6.5: The physical memory layout of the three scenarios: using (a) `Snapshot`-only, (b) `Clone`-only, and (c) `Clone`-after-`Snapshot`. Each column represents a region of memory allocated to enclave $e_0$, enclave $e_1$, or the operating system (OS). Each region either respects the permissions based on the regular platform semantics or is snapshot memory which is writable only using the copy-on-write scheme.

parent enclave being cloned so that the child enclave can read and execute the same memory contents as the parent at the time of clone.

This extension requires four new metadata state variables that are indicated in Table 6.2 with the † symbol. $\mathcal{M}^{IS}[e]$ is a Boolean valued variable indicating whether or not `Snapshot` has been called on the enclave $e$. $\mathcal{M}^{CC}[e]$ is the number of children $e$ has, or in other words, the number of times a clone has been called on the enclave $e$ where $e$ is the parent of `Clone`. $\mathcal{M}^{RS}[e]$ is a reference to the root snapshot of $e$ if one exists, and $\mathcal{M}^{PAF}[e]$ is a map of addresses that have been assigned to $e$ but are not yet allocated memory.

We now define the semantics of the two new operations introduced in Cerberus.

## Clone

`Clone` creates a clone of an existing logical enclave such that there exist two enclaves with identical enclave states. `Clone` alone provides a functionality similar to `fork` and `clone` system calls, no matter whether the platform enables memory sharing. More concretely, the `Clone` takes in three arguments: the ID of the existing parent enclave $e^p_{id} \in \mathcal{E}_{id}$ to clone, the enclave ID of the child enclave $e^c_{id} \in \mathcal{E}_{id}$ and a set of physical addresses assigned to the child enclave $x_p \subset PA$. The assigned physical addresses are marked as free (i.e., $\mathcal{M}^{PAF}[e_c][p] = true, \forall p \in x_p$) so that the parent's memory can be copied to them. The child enclave $e_c$ with corresponding enclave ID $e^c_{id}$ is used to create a clone of the parent $e_p$ such that $E_{e_p}(\sigma) = E_{e_c}(\sigma)$. In other words, the virtual memory of both enclaves is equal. The physical memory ownership of this scenario is depicted in Figure 6.5(a). We write $E_{e_p}(\sigma_0)$ to denote the initial state of the parent such that $init(E_{e_p}(\sigma))$.

We view the `Clone` as a special way of creating an enclave; instead of starting from the initial enclave state $E_{e_p}(\sigma_0)$, we start from an existing enclave $e_p$, which is effectively identical to creating two enclaves with the same initial state and then executing the same sequence of inputs up until the point clone was called.

To prevent the malicious use of clones, we require the condition Eq. 6.1 to hold during state $\sigma$ when `Clone` is called.

$$\begin{aligned} &\sigma.e_{curr} = \mathcal{OS} \wedge valid(e^p_{id}) \wedge active(e^p_{id}, \sigma) \quad \wedge \\ &valid(e^c_{id}) \wedge \neg active(e^c_{id}, \sigma) \quad \wedge \\ &e^c_{id} \neq e^p_{id} \quad \wedge \\ &\forall p \in PA. p \in x_p \Rightarrow \sigma.o[p] = \mathcal{OS} \quad \wedge \\ &sufficient\_mem(\sigma.o) \end{aligned} \tag{6.1}$$

This condition states that the `Clone` succeeds if and only if the operating system (and hence not an enclave) is currently executing, the parent is a valid and active enclave, the child enclave ID is valid but it doesn't point to an active enclave, both the parent and child enclave IDs are distinct, all physical addresses in $x_p$ are owned by the OS (and thus can be allocated to the enclave), and there is *sufficient memory* to be allocated to the enclave.

If the condition passes, `Clone` copies all of the data in the virtual address space of $e_p$ to $e_c$ to ensure write isolation. For each virtual address $v$ mapped by $e_p$ (mapped), `Clone` first selects a physical address $p$ owned by $e_c$, copies the contents from $\Pi[\mathcal{M}^{AM}_{PA}[e_p][v]]$ to $\Pi[p]$, and update the page table of $e_c$ such that $\mathcal{M}^{AM}_{PA}[e_c][v] = p$. This can be implemented in the platform itself (i.e., the security monitor firmware in Keystone) or in a local vendor-provided enclave (i.e., similar to the Quoting Enclave in Intel® SGX).

In Eq. 6.1, $sufficient\_mem : PA \rightarrow \mathcal{E}_{id} \rightarrow Bool$, $sufficient\_mem$ can be viewed as a predicate that determines whether there is enough memory to copy all data. $sufficient\_mem$ is modeled abstractly in the TAP$_C$ model to avoid an expensive computation to figure out whether there is enough memory.

`Clone` is only called from the untrusted OS because it requires the OS to allocate resources for the new enclave. Thus, if an enclave program needs to clone itself, it needs to collaborate with the OS to have it call `Clone` on its behalf. As the newly-created enclave is still an isolated enclave, the SRE property on both parent and child enclaves should hold even with a malicious OS.

### Snapshot

`Clone` by itself still requires copying the entire virtual memory to ensure isolation. To enable memory sharing, `Snapshot` makes the caller enclave $e$ to be an immutable image. After calling `Snapshot`, $e$ becomes a special type of enclave referred to as a *snapshot enclave* or the *root snapshot* of its descendants. $e$ is no longer allowed to execute at this point because all of its memory becomes read- or execute-only (Figure 6.5(b)). On the other hand, $e$ can be *cloned* by `Clone`, where the descendants of $e$ are allowed to read directly from the $e$'s shared data pages. Any writes from the descendants to physical addresses $p \in PA$ owned by $e$ (i.e., $\sigma.o[p] = e$) trigger copy-on-write (CoW). This scheme ensures that the descendant enclaves are still write-isolated from each other.

Like `Clone`, `Snapshot` has a success condition described in Eq. (6.2). The condition checks that the current executing enclave is valid $valid(\sigma.e_{curr})$ and active $active(\sigma.e_{curr}, \sigma)$, $e$ is not already a snapshot, and the enclave cannot have a root snapshot in the current state.

$$\begin{aligned} & valid(\sigma.e_{curr}) \wedge active(\sigma.e_{curr}, \sigma) \quad \wedge \\ & \neg\sigma.\mathcal{M}^{IS}[e] \wedge \neg valid(\sigma.\mathcal{M}^{RS}[\sigma.e_{curr}]) \end{aligned} \tag{6.2}$$

If `Snapshot` is called successfully in a state that satisfies this condition, $e$ is marked as a snapshot enclave. In the formal model, the metadata state $\mathcal{M}^{IS}[e]$ is to *true*.

### Clone after Snapshot

In order to make `Clone` work with `Snapshot`, `Clone` additionally increments $e_p$'s child count $\mathcal{M}^{CC}[e_p]$ by 1, and sets the root snapshot of $e_c$ (i.e., $\mathcal{M}^{RS}[e_c]$) to either $e_p$ or $e_p$'s root snapshot $\mathcal{M}^{RS}[e_p]$ if it has one.

With the single-sharing model, arbitrarily nested calls of `Clone` should still keep only one shareable enclave. As shown in Figure 6.6, there will be only one root snapshot $e_1$, whose memory is shared across all the descendants. This means that even though cloning can be arbitrarily nested, the maximum height of the tree representing the root snapshot to the child enclave is one.

To maintain the same functionality, the virtual address space of the parent and the child should be the same right after `Clone`. Thus, a descendant enclave memory will diverge from the shared memory when the descendant writes. Depicted in Figure 6.5(c), writing to snapshot memory in the child enclave results in a copy-on-write that allocates memory for the write. Unfortunately, there is no better way than to have `Clone` copy the diverged memory from the parent to the child. This is a limitation of Cerberus because the benefit of sharing memory will gradually vanish as the memory of the descendant diverges from the snapshot. However, we claim that Cerberus is very effective when the enclaves mostly write to a small part of the memory while sharing the rest. It is the programmer's responsibility to optimize their program by choosing the right place to call `Snapshot`.

## 6.5 Formal Guarantee of Secure Remote Execution

To recap, one of our goals is to prove that our extension applied to an enclave platform does not weaken the high-level security property SRE. We accomplish this by reproving SRE on TAP$_C$. As per Theorem 3.2 [204] (restated below as Theorem 6.1), it suffices to show that the triad of properties – secure measurement, integrity, and confidentiality – hold on TAP$_C$

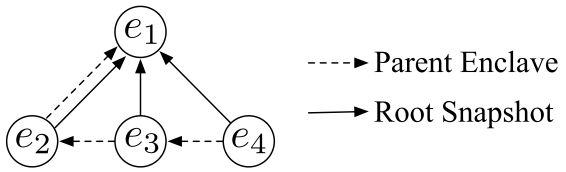


Figure 6.6: Parent-child relationship and root snapshot-child relationship of four enclaves in Cerberus. Enclave $e_1$ is a snapshot and the parent enclave of $e_2$, which is the parent of $e_3$, which is the parent of $e_4$. Despite the nested parent relationship, the root snapshot of $e_2, e_3$, and $e_4$ are $e_1$.

to prove SRE. In this section, we formally define SRE, the decomposition theorem, and each of the properties along with informal justification as to why they hold in the TAP$_C$ model against the adversary described in Section §6.4. Each of these properties has been mechanically proven on the base TAP model [204] without `Snapshot` and `Clone`, and in our work, we extend these proofs to provide the same guarantees for the memory adversary on the extended TAP$_C$ model. For brevity, we leave out some of the model implementation details and refer the reader to the GitHub repository for the proofs. We also provide a list of additional inductive invariants required to prove the properties in the TAP$_C$ model in §6.5.2.

While there are several flavors of non-interference properties, the following properties are based on the observational determinism (OD) [51, 233] definition of non-interference generalized for traces of concurrent systems. OD commonly shows up in several formalisms of confidentiality and integrity including the classic work on separability by Rushby [179] among other work [46, 83, 119, 204]. At a high level, OD states that if the initial states are low-equivalent and low inputs are the same, all states including intermediate states must also be low-equivalent (i.e., observationally deterministic functions of the low state/inputs). Whereas the classic non-interference property has an obligation [233] to prove termination and does not reason about intermediate states. We find OD to be more appropriate because we desire to show that every state of execution is observationally deterministic and indistinguishable.

### 6.5.1 Secure Remote Execution

**Definition 6.1** (Secure Remote Execution). *Let $\pi = \{\sigma_0, \sigma_1, ...\}$ be a possibly unbounded-length sequence of platform states and $\pi' = \{\sigma'_0, \sigma'_1, ...\}$ be the subsequence of $\pi$ containing all of the enclave executing states (i.e. $\forall i \in \mathbb{N}.curr(\sigma'_i) = e$). Then the set $[\![e]\!] = \{\langle I_e(\sigma'_0), E_e(\sigma'_0),\ O_e(\sigma'_0)\rangle, ...| init_e(E_e(\sigma_0))\}$ describes all valid enclave execution traces and represents the expected semantics of enclave $e$. A remote platform performs SRE of an enclave program $e$ if any execution trace of $e$ on the platform is contained within $[\![e]\!]$. In addition, the platform must guarantee that a privileged software attacker can only observe a projection of the execution trace defined by obs.*

To prove SRE, the following theorem from prior work [204] allows us to decompose the proof as follows.

**Theorem 6.1.** *An enclave platform that satisfies secure measurement, integrity, and confidentiality property for any enclave program also satisfies secure remote execution.*

**Secure measurement**. In any enclave platform, the user desires to know that the enclave program running remotely is in fact the program that it intends to run. In other words, the platform must be able to *measure* the enclave program to allow the user to detect any changes to the program prior to execution. The first part of the measurement property stated as Eq. (6.3) requires that the measurements $\mu(e_1)$ and $\mu(e_2)$ of any two enclaves $e_1$ and $e_2$ in their initial states are the same if and only if the enclaves have identical initial enclave states. $\mu$ is defined to be the measurement function that the user would use to check that their enclave $e$ is untampered with in the remote platform.

$$\begin{aligned}\forall \sigma_1, \sigma_2 \in S.\big(\mathit{init}(E_{e_1}(\sigma_1)) \wedge \mathit{init}(E_{e_2}(\sigma_2))\big) \Rightarrow \\ \big(\mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2)\big)\end{aligned} \tag{6.3}$$

The second part of measurement ensures that the enclave executes deterministically given an initial state. This is formalized as Eq. (6.4), which states that any two enclaves $e_1$ and $e_2$ starting with the same initial states, executing in lockstep and with the same inputs at each step, should have equal enclave states and outputs throughout the execution. Together, these properties help guarantee to the user that their enclave is untampered with.

$$\begin{aligned}\forall \pi_1, \pi_2.\Big(&E_{e_1}(\pi_1^0) = E_{e_2}(\pi_2^0) \quad \wedge \\ &\forall i \in \mathbb{N}.(\mathit{curr}(\pi_1^i) = e_1) \iff (\mathit{curr}(\pi_2^i) = e_2) \quad \wedge \\ &\forall i \in \mathbb{N}.(\mathit{curr}(\pi_1^i) = e_1) \Rightarrow I_{e_1}(\pi_1^i) = I_{e_2}(\pi_2^i)\Big) \quad \Rightarrow \\ &\Big(\forall i \in \mathbb{N}.E_{e_1}(\pi_1^i) = E_{e_2}(\pi_2^i) \wedge O_{e_1}(\pi_1^i) = O_{e_2}(\pi_2^i)\Big)\end{aligned} \tag{6.4}$$

With the addition of the `Clone` and `Snapshot`, the measurement of enclaves does not change for two reasons. Equation 6.3 is satisfied because the measurement of a child is copied over from the parent, and has an equivalent state as the parent. In addition, because each enclave child executes in a way that is identical to the parent without `Clone`, the child enclave $e_c$ is still deterministic up to the inputs $I_{e_c}(\sigma)$.

**Integrity**. The second property, integrity, states that the enclave program's execution cannot be affected by the adversary beyond the use of inputs $I_e$ at each step and initial state $E_e(\pi_1^0)$, formalized as Equation 6.5.

$$\begin{aligned}\forall \pi_1, \pi_2.\Big(&E_e(\pi_1^0) = E_e(\pi_2^0) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e) \iff (curr(\pi_2^i) = e) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e) \Rightarrow I_e(\pi_1^i) = I_e(\pi_2^i)\Big) \quad \Rightarrow \\ &\Big(\forall i \in \mathbb{N}.E_e(\pi_1^i) = E_e(\pi_2^i) \wedge O_e(\pi_1^i) = O_e(\pi_2^i)\Big)\end{aligned} \tag{6.5}$$

`Clone` creates a logical copy of the enclave whose behavior matches the parent enclave had it not been cloned and thus clone does not affect the integrity of the enclave. `Snapshot` freezes the enclave state and thus does not affect the integrity vacuously because the state of $e$ after calling snapshot does not change until its destruction.

**Confidentiality**. Lastly, the confidentiality property states that given the same enclave program with different secrets represented by $e_1$ and $e_2$ in traces $\pi_1$ and $\pi_2$ respectively, if the adversary starts in the initial state $A_{e_1}(\pi_1[0])$ and the protected enclave(s) $e_1$ (and $e_2$) is operated with a (potentially malicious) sequence of inputs $I_{e_1}$, the adversary should not learn more than what's provided by the observation function *obs* and hence its state $A_{e_1}(\sigma)$ and $A_{e_1}(\sigma)$ should be the same. The fourth line of Equation 6.6 requires that any changes by the protected enclave $e$ do not affect the observations made by the adversary in the next step. This is to avoid spurious counter-examples where secrets leak through obvious channels such as the enclave output which is a bug in the enclave program as explained in Section 6.4.3.

$$\begin{aligned}\forall \pi_1, \pi_2.\Big(&A_{e_1}(\pi_1^0) = A_{e_2}(\pi_2^0) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = curr(\pi_2^i) \wedge I_{e_1}(\sigma_1^i) = I_{e_2}(\sigma_2^i)) \quad \wedge \\ &\forall i \in \mathbb{N}.(curr(\pi_1^i) = e) \Rightarrow obs(\pi_1^{i+1}) = obs(\pi_2^{i+1})\Big) \quad \Rightarrow \\ &\Big(\forall i \in \mathbb{N}.A_{e_1}(\pi_1^i) = A_{e_2}(\pi_2^i)\Big)\end{aligned} \tag{6.6}$$

`Snapshot` alone clearly does not affect the confidentiality of the enclave. `Clone` on the other hand also does not affect confidentiality because it creates a logical duplicate of an enclave. Had the adversary been able to break the confidentiality of the child $e_c$, it should have been able to break the confidentiality of the parent $e_p$ because both should behave in the same way given the same sequence of input.

## 6.5.2 Cerberus Platform Invariants

We describe a few key additional platform inductive invariants that were required to prove the SRE property on TAP$_C$. Although the following list is not exhaustive[§], it provides a

[§]We refer the reader to the formal models in the Uclid5 code for the complete list of inductive invariants in full detail.

summary of the difference between the invariants in the base TAP model and the TAP$_C$ model and explains what precisely makes the other sharing models more difficult to verify. The invariants are typically over the two traces $\pi_1$ and $\pi_2$ in the properties previously mentioned. However, there are single-trace properties, and unless otherwise noted, it is assumed that single-trace properties defined over a single trace $\pi$ hold for both traces $\pi_1$ and $\pi_2$ in the properties.

**Memory Sharing Invariants**. We first begin with the invariants related to the memory-sharing model. As explained earlier, allowing the sharing of memory weakens the constraint that memory is strictly isolated. This means that the memory readable and executable by an enclave can either belong to itself or its root snapshot. This is true for the entrypoints of the enclave and the mapped virtual addresses. These are described as Eq. (6.7) and Eq. (6.8) respectively.

Eq. (6.7) states that all enclaves have an entrypoint that belongs to $e$ itself or its snapshot $\pi^i.\mathcal{M}^{RS}[e]$.

$$\begin{aligned}\forall e \in \mathcal{E}_{id}, \forall i \in \mathbb{N}.\Big(&valid(e) \quad \Rightarrow \\ &\big(\pi^i.o[\pi^i.\mathcal{M}^{EP}[e]] = e \quad \vee \\ &\pi^i.o[\pi^i.\mathcal{M}^{EP}[e]] = \pi^i.\mathcal{M}^{RS}[e]\big)\Big)\end{aligned} \tag{6.7}$$

Eq. (6.8) states that every enclave $e$ whose physical address $p \in PA$ corresponding to virtual address $v \in VA$ in the page table that is mapped $mapped_e(\pi[i].a_{PA}[v])$[¶] either belongs to $e$ itself or the root snapshot $\pi^i.\mathcal{M}^{RS}[e]$.

To illustrate the potential complexity of the capped and arbitrary memory-sharing models, the antecedent of this invariant would need to existentially quantify over all the possible snapshot enclaves that own the memory as opposed to the current two (the enclave itself or its root snapshot). This would introduce an alternating quantifier[176] in the formula, making reasoning with SMT solvers difficult.

$$\begin{aligned}&\forall e \in \mathcal{E}_{id}, v \in VA, \forall i \in \mathbb{N}. \\ &\quad\Big(\big(valid(e) \wedge active(e, \pi^i) \wedge mapped_e(\pi^i.a_{PA}[v])\big) \Rightarrow \\ &\quad\big(\pi^i.o[\pi^i.a_{PA}[v]] = e \quad \vee \\ &\quad\ \pi^i.o[\pi^i.a_{PA}[v]] = \pi^i.\mathcal{M}^{RS}[e]\big)\Big)\end{aligned} \tag{6.8}$$

Lastly, a memory that is marked free for an enclave $e$ is owned by that enclave itself, represented by Eq. (6.9).

$$\forall e \in \mathcal{E}_{id}, p \in PA, \forall i \in \mathbb{N}.\Big(\pi^i.\mathcal{M}^{PAF}[e][p] \Rightarrow \pi^i.o[p] = e\Big) \tag{6.9}$$

[¶]mapped is a function that returns whether a physical address is mapped in enclave $e$ and is equivalent to the *valid* function in the CCS'17 paper [204]

**Snapshot Invariants**. The next invariants relate to snapshot enclaves. First, the root snapshot of an enclave is never itself as follows:

$$\forall e \in \mathcal{E}_{id}, \forall i \in \mathbb{N}.(valid(e) \Rightarrow \pi^i.\mathcal{M}^{RS}[e] \neq e) \tag{6.10}$$

Snapshots also do not have root snapshots Eq. (6.11). This invariant reflects the property that the root snapshot to ancestor enclave relationship has a height of at most 1. This is stated as all enclaves that are snapshots have a root snapshot reference pointing to the invalid enclave ID $e_{inv}$.

$$\begin{aligned} &\forall e \in \mathcal{E}_{id}, \forall i \in \mathbb{N}. \\ &\Big(\big(valid(e) \wedge active(e, \pi^i) \wedge \pi^i.\mathcal{M}^{IS}[e]\big) \Rightarrow \\ &\pi^i.\mathcal{M}^{RS}[e] = e_{inv}\Big) \end{aligned} \tag{6.11}$$

Next, if an enclave has a root snapshot that is not invalid (i.e. $\pi^i.\mathcal{M}^{RS}[e] \neq e_{inv}$), then the root snapshot is a snapshot and the child count is positive. This is represented as Eq. (6.12).

$$\begin{aligned} &\forall e \in \mathcal{E}_{id}, \forall i \in \mathbb{N}. \\ &\Big((valid(\pi^i.\mathcal{M}^{RS}[e]) \wedge active(\pi^i.\mathcal{M}^{RS}[e], \pi^i)) \Rightarrow \\ &\qquad \big(\pi^i.\mathcal{M}^{IS}[\pi^i.\mathcal{M}^{RS}[e]] \quad \wedge \\ &\qquad \pi^i.\mathcal{M}^{CC}[\pi^i.\mathcal{M}^{RS}[e]] > 0\big)\Big) \end{aligned} \tag{6.12}$$

The last notable invariant says that the currently executing enclave cannot be a snapshot as described in Eq. (6.13).

$$\forall i \in \mathbb{N}.(\neg \pi^i.\mathcal{M}^{IS}[\pi^i.e_{curr}]) \tag{6.13}$$

We conclude this section by noting that coming up with the exhaustive list of inductive invariants for TAP$_C$ took a majority of the verification effort.

## 6.6 Implementation in RISC-V Keystone

To show the feasibility of our approach, we implement Cerberus on Keystone [127]. Keystone is an open-source framework for building enclave platforms on RISC-V processors. In Keystone, the platform operations $\mathcal{O}_{base}$ are implemented in high-privileged firmware called security monitor. We implemented additional `Snapshot` and `Clone` based on our specifications. All fields of the enclave metadata are stored within the security monitor memory. We extended the metadata with the variables corresponding to $\mathcal{M}^{IS}$, $\mathcal{M}^{CC}$, $\mathcal{M}^{RS}$, and $\mathcal{M}^{PAF}$. All implementations are available at `https://github.com/cerberus-ccs22/TAPC.git`.

The implementation complies with the assumptions of the model described in §6.4.2. First, Keystone enclave operations are atomic operations, and it updates the system state only when the operation succeeds. Second, we implement a deterministic memory allocation algorithm for copy-on-write, by leveraging Keystone's free memory module.

For memory isolation, Keystone uses a RISC-V feature called Physical Memory Protection (PMP) [220], which allows the platform to allocate a contiguous chunk of physical memory to each of the enclaves. When an enclave executes, the corresponding PMP region is activated by the security monitor. We implemented the weakened constraints (i.e., Equation 6.8) by activating the snapshot's memory region when the platform context switches into the enclave.

In the model, the platform would need to handle the copy-on-write. In Keystone, an enclave can run with supervisor privilege, which allows the enclave to manage its own page table. This was very useful when we prototype this work because the platform does not need to understand the virtual memory mapping of the enclave. Letting the enclave handle its own write faults does not hurt the security because the permissions on physical addresses are still enforced by the platform. One implementation challenge was that the enclave handler itself would always trigger a write fault because the handler requires some writable stack to start execution. We were able to implement a stack-less page table traverse, which allows the enclave to remap the page triggering the write fault without invoking any memory writes. The final copy-on-write handler is similar to the on-demand fork [235].

## 6.7 Evaluation

Our evaluation goals are to show the following:

- **Verification results:** Our incremental verification approach enables fast formal reasoning on enclave platform modifications.
- **Start-up latency:** The Cerberus interface can be used with process-creation system calls to reduce the start-up latency of enclaves
- **Computation overhead:** Our copy-on-write implementation does not incur significant computation overhead.
- **Programmability:** Cerberus provides a programmable interface, which can be easily used to improve the end-to-end latency of server enclave programs.

Throughout the performance evaluation, we used SiFive's FU540 [84] processor running at 1 GHz and an Azure DC1s_v3 VM instance with an Intel® Xeon® Platinum 8370C running at 2.4 GHz to run Keystone and SGX workloads respectively. Each experiment was averaged over 10 trials.

### 6.7.1 Verification Results

The TAP$_C$ model and proofs can be found at `https://github.com/cerberus-ccs22/TAPC.git`.

**Porting TAP from Boogie to Uclid5.** One other contribution of this work includes the port of the original TAP model from Boogie [16] to UCLID5 [193]. UCLID5 is a verification toolkit designed to model transition systems modularly, which provides an advantage over the previous implementation written in the software-focused verification IR Boogie. We also find that UCLID5 is advantageous over other state-of-the-art tools [15, 56, 129, 208] because of modularity and because it provides flexibility in modeling systems both operationally and axiomatically. This effort took 1.875 person-months working approximately 40 hours a week to finish[‖].

**Verifying TAP$_C$.** The modeling and verification took roughly three person-months to write the extensions to the TAP model and verify using a scalable approach. We note that this time is substantially less than it would have taken to rebuild the model from scratch without an existing abstraction.

Figure 6.7 shows the number of procedures #pn, the number of (uninterpreted) functions #fn, the number of annotations #an (which include pre- and post-conditions, loop invariants, and system invariants), and the number of lines of code #ln. The last column shows the verification time which includes the time it took UCLID5 to generate verification conditions and print them out in SMTLIB2.0 and verify them using Z3/CVC4[**]. The time discrepancy between the original proofs [204] and the ones in this effort can be explained by the way we generate all the verification conditions as SMTLIB on disk before verifying as a way to use other SMT solvers. We also use UCLID5 instead of Boogie [16]. We note that the number of lines for `Snapshot` and `Clone` is 1110, which means only 489 lines were used to extend the existing TAP operations and platform model.

| **Model/Proof** | **Size** #pr | #fn | #an | #ln | **Verif. Time (s)** |
|---|---|---|---|---|---|
| | | | **TAP Models** | | |
| `TAP` | 43 | 14 | 225 | 2100 | 140 |
| `Integrity` | 2 | 0 | 52 | 525 | 285 |
| `Mem. Conf` | 3 | 0 | 44 | 838 | 342 |
| | | | **TAP$_C$ Models** | | |
| `TAP` | 45 | 16 | 466 | 3689 | 1380 |
| `Integrity` | 2 | 0 | 109 | 937 | 934 |
| `Mem. Conf` | 3 | 0 | 119 | 1307 | 944 |

Figure 6.7: Model Statistics and Verif. Times

Despite the added complexity, each operation for each proof took only a few minutes to verify individually as shown in the last column of Figure 6.7. This demonstrates that our incremental verification methodology is practical and consequently reduces the overall time to verify additional operations at a high level.

[‖]We note that these statistics in this section are estimates and numbers may have slight inaccuracies.

[**]For one of the properties, we observed that Z3 would exceed our timeout limit of 30 minutes whereas CVC4 wouldn't.

As evident by the results, we emphasize that the single-sharing model is practical to formally encode and verify. We also confirm that introducing invariants with alternating quantifiers and existential quantifiers in our models degraded the verification time and would likely do the same for alternative models. These attempts heavily influenced our decisions and we strongly encourage the use of our sharing model.

### 6.7.2 Start-up Latency

To show the efficacy of Cerberus interface, we implement `fork` and `clone` system calls based on Cerberus. When the system calls are invoked in the enclave program, it calls `Snapshot` to create an immutable image and cooperates with the OS to clone two enclaves from the snapshot using `Clone`. We compare the latency of `fork` on two different platforms: SGX-based Graphene [210] (now Gramine Linux Foundation project [78]) and RISC-V Keystone [127] with Cerberus. Figure 6.8 shows the program that calls `fork` after allocating memory with `SIZE`.

```c
int main() {
  char* buf = malloc(SIZE);
  clock_t start = clock();
  // fork child
  if (!fork()) {clock_t end = clock();}
  // parent
  else { return; }
}
```

Figure 6.8: C code to measure `fork` latency

The baseline (Graphene-SGX) latency increases significantly as the allocation size increases (Figure 6.9). With a 400 MB buffer, it takes more than 6 seconds to complete. Also, each of the enclaves will take 400 MB of memory at all times, even when most of the content is identical until one of the enclaves writes. With Cerberus, the latency does not increase with respect to the allocation size. This is because we are not copying any of the parent's memory including the page table. It only took 23 milliseconds to fork on average, with a standard deviation of 16 microseconds.

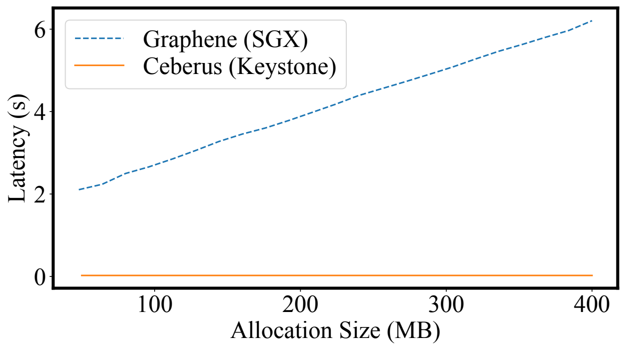


Figure 6.9: The latency of `fork` with respect to the size of the allocated memory.

### 6.7.3 Computation Overhead

We measure the computation overhead incurred by CoW invocation. In order to see the overhead for various memory sizes and access patterns, we use RV8 [181] benchmark. RV8 consists of 8 simple applications that perform single-threaded computation. We omit `bitint` as we were not able to run it on the latest Keystone, because of a known bug on their side. Since RV8 does not use the `fork` system call, we have modified RV8 such that each of them forks before the computation begins. Note that all of the application starts with allocating a large buffer, so we inserted `fork` after the allocation. Thus, copy-on-write memory accesses are triggered during the computation depending on the memory usage.

As you can see in Figure 6.10, the average computation overhead of copy-on-write memory over Keystone was only 3.9%. The worst overhead was 19.0% incurred in `qsort`, which uses the largest memory (about 190 MiBs). We argue that the benefit of cloning an enclave is small for such workloads that have a large buffer that is not shared across enclaves.

### 6.7.4 Programmability

To show the programmability of Cerberus interface, we showcase how server programs can leverage memory sharing to improve their end-to-end performance.

Although `Snapshot` or `Clone` are not directly related to `fork` or `clone`, their behavior maps well with `Snapshot` and `Clone`. For example, those system calls create a new process with exactly the same virtual memory, which can be mapped to `Clone` and optimized by `Snapshot`. Thus, we provided two co-authors with the modified `fork` and `clone` that use Cerberus interface and asked to make the server programs leverage memory sharing.

An author modified darkhttpd, a single-threaded web server, to fork processes to handle new HTTP requests inside the event loop. This allowed darkhttpd to serve multiple requests concurrently and continue listening for new requests. We measure the latency of an HTTP request using `wget` to fetch 0.5 MB of data. The resulting program incurs only a 2.1x slowdown over the native (non-enclave) execution, in contrast to a 33x slowdown in corresponding Intel SGX implementation (the exact same program ran with Graphene). The 2.1x overhead is mainly due to the slow I/O system calls, which is a well-known limitation of enclaves [127, 223].

Another author implements a simple read-only database server application using *Sqlite3*, which is a single-file SQL library that supports both in-memory and file databases. The

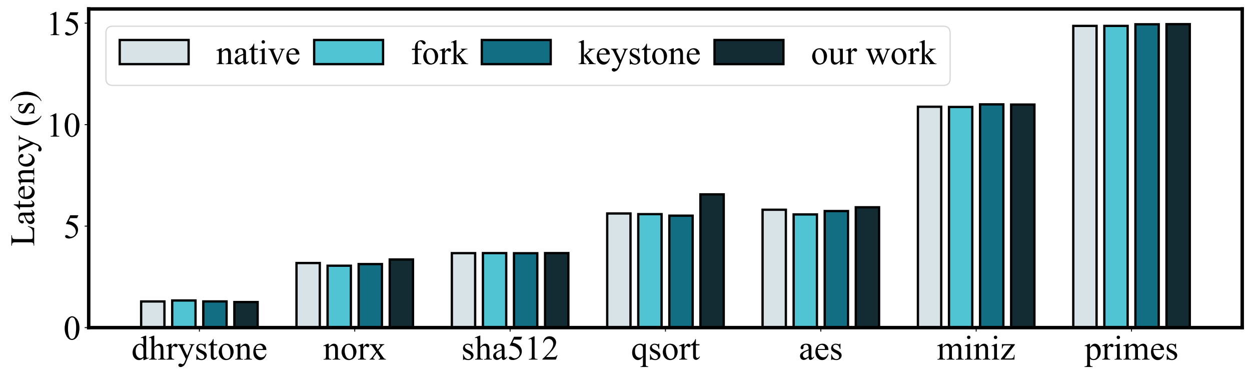


Figure 6.10: Computation Overhead on RV8. **native**: native execution of the original RV8, **fork**: native execution of the modified RV8 with `fork`, **keystone**: enclave execution of the original RV8, and **our work**: enclave execution of the modified RV8 with Cerberus.

resulting program serves each query with a fresh child created by `fork`. We measure the latency of 1,000 `SELECT` queries served by separate enclaves. The resulting program incurs a 36x slowdown over the native execution, compared to a 262x slowdown in corresponding Intel SGX implementation. SGX overhead is much worse than in Darkhttpd because there is more data to copy over (the entire in-memory database). The 36x slowdown is mainly due to limited concurrency in Keystone: since Keystone implements memory isolation with a limited number of PMP entries, it can support only up to 3-4 concurrent enclaves. This is not an inherent limitation of Cerberus.

Both authors did not have any difficulties in allowing enclaves to share a memory, because they were already familiar with the expected behavior of the system calls. However, they did not have any knowledge of the codebase of Darkhttpd nor Sqlite3 prior to the modification. Darkhttpd required modification of less than 30 out of 2,900 lines of code, which took less than 10 person-hours, and Sqlite3 consists of 103 lines of code, which took less than 20 person-hours. This shows that the Cerberus extension can be easily used to improve the end-to-end performance of server programs.

## 6.8 Discussion

**Low-equivalent states**. Our security model contains the notion of low states as in standard observational determinism type properties, even though they are not explicitly stated in section § 6.5. Instead, the low states are constrained to be equal in the antecedent of the implication of the properties. The traditional non-interference or observational determinism properties most closely resemble the confidentiality Eq. 6.6. In this property, the low states include the inputs $I_{e_1}(\sigma), I_{e_2}(\sigma)$ to the enclaves and platform operations controlled by the untrusted OS, all of the adversary controlled enclaves' state $E_e(\sigma)$, where $e$ is not the protected enclave (i.e., $e_1, e_2$), and the adversary state $A_{e_1}(\sigma), A_{e_2}(\sigma)$. The idea is to prove that under the same sequence of adversarial controlled inputs, the adversary cannot differentiate between the two traces which have the same enclave (i.e. $e_1$, $e_2$) with differing high data in memory. Similarly, the low states of the integrity property (Eq. 6.5) includes the untrusted inputs $I_{e_1}(\sigma), I_{e_2}(\sigma)$. However, instead of constraining the adversary inputs to be the same, we want to show that an enclave executes deterministically regardless of the state outside the enclave. As a result, the remaining low states include the protected enclaves $E_{e_1}(\sigma), E_{e_2}(\sigma)$. Lastly, secure measurement can be viewed as a form of integrity proof and contains the same low states as the integrity property.

**Performance comparison with previous work**. The evaluation section does not make direct performance comparisons with previous work such as PIE, which is based on x86. However, based on our calculations, Cerberus's overhead is on par with PIE. For example, PIE incurs about 200ms startup latency on serverless workloads [131] whereas Cerberus incurs 23ms on `clone` system calls. We analyze that Cerberus is faster mainly because it leverages Keystone's ability to quickly create an enclave with zero-filled memory without measurement, which SGX does not support. PIE's copy-on-write introduces 0.7-32ms overhead on serverless function invocations taking 144-1153ms (the paper did not provide relative overhead over native execution), which can be roughly translated into less than a few percent of overhead, which is similar to Cerberus.

**Verifying the implementation**. We do not verify the implementation of Cerberus in Keystone. Unsurprisingly, any discrepancy between the model and the implementation can make the implementation vulnerable. In particular, the enclave page table is abstracted as enclave metadata in TAP and $\text{TAP}_C$, where it is actually a part of memory $\Pi$. Cerberus in Keystone does not create any security holes because the page table management is trusted (the enclave manages it). However, this does not mean that we can apply the same argument to the other implementations. To formally verify the implementation, we can construct the model for Keystone implementation and do the refinement proof to show that the model refines the TAP model as described by Subramanyan *et al.* [204]. We leave this as future work.

**In-enclave isolation**. Instead of modifying the platform, a few approaches [3, 121, 146, 196] use *in-enclave* isolation to create multiple security domains within a single enclave. However, security guarantees of such solutions rely on the formal properties of not only the enclave platform, but also the additional techniques used for the isolation. For example, the security of software fault isolation (SFI) [218] based approaches [3, 196] depends on the correctness and robustness of the SFI techniques including the shared software implementation and the compiler, which should be formally reasoned together with the enclave platform. Thus, such approaches will result in a significant amount of verification efforts.

## 6.9 Summary

We showed how to formally reason about modifying the enclave platform to allow memory sharing. We introduce the single-sharing model, which can support secure and efficient memory sharing of enclaves. We also proposed two additional platform operations similar to existing process-creation system calls. In order to formally reason about the security properties of the modification, we defined a generic formal specification by incrementally extending an existing formal model. We showed that our incremental verification allowed us to quickly prove the security guarantees of the enclave platform. We also implemented our idea on Keystone open-source enclave platform and demonstrated that our approach can bring significant performance improvement to server enclaves.

# Chapter 7

# Conclusion

Every year, new hardware attacks are discovered, resulting in an ever-increasing attack surface for hardware platforms with no end in sight as there is no one-size-fits-all solution. Moreover, programs often have varying security criteria with vastly different attacker or platform models. Ideally, security should be guaranteed in all of these situations in an automated way that is agnostic to the details of the attack. Thus, it is beneficial to have formal security properties and parameterized models that are parameterizable and composable because it offers a standardized methodology for reasoning about security guarantees for a combination of attacker and platform models. In this thesis, we have shown how we accomplish this by formalizing hardware-attack-based properties and parameterized hardware platform models, and by extending the methodology to compositionally verify programs.

This concluding chapter summarizes the thesis contributions and discusses the limitations and suggested directions for future work.

## 7.1 Summary of Contributions

Chapter §4 presented a formalization of secure speculation to capture the newly discovered transient execution attacks that became one of the predominant classes of hardware attacks in the last several years. To capture hardware attacks in general (not due to speculation), the property was generalized as the trace property-observational determinism property. In addition, the methodology of combining an instruction-level program with speculative semantics defined by the Assembly Intermediate Representation was introduced as a way to model the behavior of a speculative microprocessor. Using this model, the chapter presented a model checking-based approach that combined hardware and software to verify secure speculation for a given program executing on a given platform against an adversary with the capability to observe the cache.

Chapter §5 acknowledged the limitations of the approach presented in Chapter §4 and focused on presenting methods of composition through a proof system and a modeling formalism. The proof system SymboTaint provides a way to compositionally reason about the information flow of a platform by localizing analyses to contiguous subsequences of a platform's run. Coupled with IFSMs, which implement SymboTaint, one can compositionally reason about an instruction sequence to avoid state explosion when using symbolic execution-based approaches. Moreover, IFSMs allow platform models to be composed, enabling parameterizable platform models. Lastly, the SAP model was introduced as a starting point for checking classes of hardware attacks and demonstrating the practicality of the approach.

Chapter §6 revisited the question of providing secure information flow but by using an alternative method that builds on top of a robust TEE platform model. A limitation of existing TEEs was the lack of ability to allow enclave programs to share memory. This limited adoption in addition to the lack of security guarantees on the platform model. This chapter discussed the need for memory sharing and presented proof of confidentiality and integrity for Cerberus, the first formally verified abstract TEE platform model that supports memory sharing.

To summarize these contributions, this thesis presents a formal approach to secure information flow by formalizing security properties for hardware attacks, modeling and verifying abstract hardware platform models, and improving compositional techniques for formally verifying information flow-based properties.

## 7.2 Future Work

This thesis introduced methods and directions for information flow-driven verification of vulnerabilities in hardware platforms. Unsurprisingly, they come with a number of caveats that open up a range of possible directions for future work. We sample a few potential directions to conclude the thesis.

### 7.2.1 Lifting Sound Models of Hardware Platforms

While abstractions are more practical to verify and can provide security for a family of platforms that refine it, they are often written manually by platform designers and formal verification experts. For larger systems, this task itself is already a lot of work. In addition to this, some proof of soundness is typically required for security. We discuss three following directions to address these challenges.

**Lifting sound models of hardware platforms**. Without the details of the implementation or a refinement proof, it's difficult to guarantee that proofs of secure information flow transfer from an abstract model to the implementation, which is . One standard method to address this gap is to show a refinement relation from the abstract model to the implementation (or less abstract model of the implementation) [70, 114, 204]. However, because of the necessity to model hardware, handwritten models are impractical, necessitating automated approaches to lifting implementations into more abstract models that preserve the soundness of desired secure information flow properties.

**Verifying out-of-order microprocessors**. One other long-standing problem and a stretch goal of a lifting-based approach is the verification of out-of-order microprocessors. To the best of our knowledge, existing approaches [55, 109, 122, 199] for out-of-order verification have not been applied to RTL-level implementation accurate models. There is also the question of how SymboTaint and IFSMs in Chapter §5 can be applied to verify secure information flow on out-of-order processors.

**Attacker Models**. Although the attacker model we are considering is general and parameterized, it is not without limitations. Specifically, we have observed in Chapter §5 that an attacker who is only capable of observing and tampering with the system state is weaker than one who can also transfer sensitive data from one state to another. When the attacker lacks this capability, it is necessary to model more specific functions available to the adversary. Therefore, we stress the importance of exploring different attacker models for various attacks in order to conduct a thorough analysis of the platform.

### 7.2.2 Building Upon SymboTaint

In Chapter §5, we introduced SymboTaint, an interpolant-based method for efficiently verifying information-flow-based properties. However, a number of aspects were not discussed that could be improved upon. We list these below.

**Generating Interpolants**. While finding interpolants for reachable states has been studied extensively in work such as McMillan's approach to finding Craig interpolants [144] and PDR/IC3 [31, 64], finding precise interpolants remains mostly an open problem for this class of applications. While in our specific examples, it was sufficient to assume that the set of reachable states $S$ in each interpolant $\{S, \Gamma\}$ is the entire set of states $Q$ (i.e., $S = Q$), this results in false positives. A potential direction to explore is which class of programs and interpolants not only guarantee the soundness but also the completeness of the approach.

**Optimizing Temporal Decomposition in SymboTaint**. On the other hand, the temporal partitioning of a platform execution in SymboTaint can also introduce false positives. Let us refer to the maximum partition of a run (of the platform) as the partitioning of the program operations *Prog* into $\mathsf{op}_{MAX} \doteq \{\{\mathsf{op}_1\}, \{\mathsf{op}_2\}, \ldots, \{\mathsf{op}_n\}\}$, where element of $\mathsf{op}_{MAX}$ is a separate run. This partition is the largest partition of a sequence of operations (i.e., there is no partition with more elements), where the platform execution $M(\mathsf{op}_{MAX})$ in SymboTaint results in each run in $\mathsf{op}_{MAX}$ to be symbolically modeled independently from one another as $M(r_i), r_i \in \mathsf{op}_{MAX}$. Alternatively, one can consider evaluating a smaller partition of the run that separates *Prog* into two runs, i.e, $\mathsf{op}_{TWO} \doteq \{\{\mathsf{op}_1, \ldots, \mathsf{op}_k\}, \{\mathsf{op}_{k+1}, \ldots, \mathsf{op}_n\}\}$, $k \in [n]$, resulting in the trace given by executing $M(\mathsf{op}_1 \ldots \mathsf{op}_k)\ M(\mathsf{op}_{k+1} \ldots \mathsf{op}_n)$, which can be more precise than $p_{MAX}$, but less precise than $\mathsf{op}_{MIN} \doteq \{Prog\} \doteq \{\{\mathsf{op}_1, \cdots, \mathsf{op}_n\}\}$. The challenge is to develop a sound and complete method to find a partition from the set of all partitions of the program *Prog*, say $p \in \mathit{Partitions}(\mathit{Prog})$, such that $p$ is easiest to verify while preserving equisatisfiability of the desired property. To formalize this problem, let $\mu(M, p, \theta)$ be a measure of "verification difficulty" based on the platform model, partitioning of *Prog* and other relevant parameters $\theta$ for the measure $\mu$. Intuitively, one can attempt to find the partitioning $p^*$ in 7.1.

$$p^* \doteq \mathit{argmin}_{p \in \mathit{Partitions}(\mathit{Prog})}\ \mu(M, p, \theta) \ \text{ s.t. } \ M(p) \models T \iff M(\{\mathit{Prog}\}) \models T \tag{7.1}$$

where $T$ is the non-interference property we desire to prove.

### 7.2.3 Trusted Execution Environments

Lastly, in Chapter §6, we proved secure remote computation for the single-memory sharing model, Cerberus, by formalizing an extension to the platform model and manually coming up with inductive invariants.

**Formalizing features of TEEs**. One natural next step is to extend Cerberus to support alternative sharing models, such as writable-memory sharing schemes, capped memory-sharing models [131], and uncapped memory-sharing models [231]. This need not be limited to sharing models, but other features supported by TEEs that are part of the TCB. Formalizing extensions to the API interface or specific implementations of trusted hardware such as Physical Memory Protection in RISC-V [221] would also advance the frontier of TEE security, where it has become almost standard to rely on informal arguments of security.

**Automatically generating inductive invariants**. Recall that the inductive invariants for Cerberus were found manually and took time in terms of person months. For more complicated systems (e.g., a model with less abstraction), the amount of effort required does not scale well in our experience. Thus, it would be valuable if one could automatically generate these inductive invariants. As introduced in §3.3.6, semantic-based program synthesis is one mechanism that can be used to generate these invariants.

# Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow Integrity”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS ’05. 2005, pp. 340–353.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “Predicting secret keys via branch prediction”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2007, pp. 225–242.

[3] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. “Chancel: efficient multi-client isolation under adversarial programs”. In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2021.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. “Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC”. In: *International Conference on Fast Software Encryption*. Springer. 2016, pp. 163–184.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying constant-time implementations”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 53–70.

[6] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. “Formal verification of side-channel countermeasures using self-composition”. In: *Science of Computer Programming* 78.7 (2013), pp. 796–812.

[7] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. “Syntax-guided synthesis”. In: *FMCAD*. IEEE, 2013, pp. 1–8.

[8] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O’Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. “Cogent: Verifying High-Assurance File System Implementations”. In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016.

[9] *ARM Instruction Reference*. `https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference`. 2022. URL: `https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference` (visited on 12/14/2022).

[10] *ARM TrustZone*. `https://www.arm.com/products/security-on-arm/trustzone`. 2013.

[11] Krste Asanović and David A Patterson. "Instruction Sets Should Be Free: The Case For RISC-V". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[12] Thomas H. Austin and Cormac Flanagan. "Efficient purely-dynamic information flow analysis". In: *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*. Ed. by Stephen Chong and David A. Naumann. ACM, 2009, pp. 113–124.

[13] *AWS SageMaker*. `https://aws.amazon.com/pm/sagemaker`.

[14] Jonathan Bachrach, Huy D. Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012* (2012), pp. 1212–1221.

[15] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*. Lecture Notes in Computer Science. Springer, Apr. 2022. URL: `http://www.cs.stanford.edu/~barrett/pubs/BBB+22.pdf`.

[16] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLIne, Bart Jacobs, and Rustan Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *FMCO 2005*. Springer Berlin Heidelberg, Nov. 2005. URL: `https://www.microsoft.com/en-us/research/publication/boogie-a-modular-reusable-verifier-for-object-oriented-programs/`.

[17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[18] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Second. IOS Press, 2021. Chap. 33, pp. 1267–1329.

[19] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. "System-level non-interference for constant-time cryptography". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1267–1279.

[20] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure Information Flow by Self-Composition". In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17)*. 2004, pp. 100–114.

[21] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure Information Flow by Self-Composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[22] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. "Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 328–343.

[23] Christoph Baumann, Mads Dam, Roberto Guanciale, and Hamed Nemati. "On Compositional Information Flow Aware Refinement". In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 2021, pp. 1–16.

[24] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 1999.

[25] *Binary Analysis Platform (BAP) Repository*. 2019. URL: `https://github.com/BinaryAnalysisPlatform/bap`.

[26] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. "Vale: Verifying high-performance cryptographic assembly code". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 917–934.

[27] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. "The Undecidability of First-Order Logic". In: *Computability and Logic*. 5th ed. Cambridge University Press, 2007, pp. 126–136.

[28] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. "Using lightweight formal methods to validate a key-value storage node in Amazon S3". In: *SOSP 2021*. 2021.

[29] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

[30] Matko Botincan, Matthew J. Parkinson, and Wolfram Schulte. "Separation Logic Verification of C Programs with an SMT Solver". In: *International Workshop on Systems Software Verification*. 2009.

[31] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'11. Austin, TX, USA: Springer-Verlag, 2011, pp. 70–87.

[32] Bryan A. Brady, Randal E. Bryant, Sanjit A. Seshia, and Bryan A. Brady. "Abstracting RTL Designs to the Term Level". In: UCB/EECS-2008-136. Oct. 2008. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-136.pdf`.

[33] Bryan A. Brady, Randal E. Bryant, Sanjit A. Seshia, and John W. O'Leary. "ATLAS: Automatic Term-Level Abstraction of RTL Designs". In: *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE).* July 2010, pp. 31–40.

[34] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. "BAP: A Binary Analysis Platform". In: *Proceedings of the 23rd International Conference on Computer Aided Verification.* CAV'11. Snowbird, UT, 2011, pp. 463–469.

[35] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In: *Computer-Aided Verification (CAV'02).* LNCS 2404. July 2002, pp. 78–92.

[36] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. James Hwang. "Symbolic Model Checking: $10^{20}$ States and Beyond". In: *Information and Computation* 98 (1992), pp. 142–170.

[37] Jerry R. Burch and David L. Dill. "Automatic verification of Pipelined Microprocessor Control". In: *International Conference on Computer Aided Verification.* 1994.

[38] John Cable. *Update on Spectre and Meltdown security updates for Windows devices.* `https://blogs.windows.com/windowsexperience/2018/03/01/update-on-spectre-and-meltdown-security-updates-for-windows-devices/`. 2018. URL: `https://blogs.windows.com/windowsexperience/2018/03/01/update-on-spectre-and-meltdown-security-updates-for-windows-devices/`.

[39] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *CoRR* abs/1811.05441 (2018). arXiv: `1811.05441`. URL: `http://arxiv.org/abs/1811.05441`.

[40] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266.

[41] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM. 2019.

[42] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. “Constant-time foundations for the new spectre era”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019).

[43] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. “FaCT: a DSL for timing-sensitive computation”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019).

[44] Kevin Cheang, Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. *Compositional Proofs of Information Flow Properties for Hardware-Software Platforms.* Tech. rep. UCB/EECS-2023-204. EECS Department, University of California, Berkeley, Aug. 2023. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-204.html`.

[45] Kevin Cheang, Cameron Rasmussen, Dayeol Lee, David W. Kohlbrenner, Krste Asanović, and Sanjit A. Seshia. *Verifying RISC-V Physical Memory Protection.* 2022. URL: `https://arxiv.org/abs/2211.02179`.

[46] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. “A Formal Approach to Secure Speculation”. In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019.* 2019, pp. 288–303.

[47] Zilin Chen, Liam O’Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. “The Cogent Case for Property-Based Testing”. In: *Proc. of Workshop on Programming Languages and Operating Systems (PLOS).* Shanghai, China, 2017.

[48] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. “Infinite-state invariant checking with IC3 and predicate abstraction”. In: *Formal Methods in System Design* 49 (2016), pp. 190–218.

[49] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model checking and abstraction”. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), pp. 1512–1542.

[50] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. “Model Checking and the State Explosion Problem”. In: *Workshop on Learning from Authoritative Security Experiment Results.* 2011.

[51] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. In: *2008 21st IEEE Computer Security Foundations Symposium.* 2008, pp. 51–65.

[52] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: `http://eprint.iacr.org/2016/086`.

[53] Victor Costan, Ilia Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium (USENIX Security 16).* USENIX Association, 2016, pp. 857–874.

[54] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252.

[55] Werner Damm and Amir Pnueli. "Verifying out-of-order executions". In: *Advances in Hardware Design and Verification: IFIP TC10 WG10. 5 International Conference on Correct Hardware and Verification Methods, 16–18 October 1997, Montreal, Canada.* Springer. 1997, pp. 23–47.

[56] Leonardo De Moura and Nikolaj Bjorner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[57] R. DeLine and K. R. M. Leino. *BoogiePL: A typed procedural language for checking object-oriented programs*. Tech. rep. MSR-TR-2005-70. Microsoft Research, 2005.

[58] Dorothy E. Denning. "A lattice model of secure information flow". In: *Commun. ACM* 19 (1976), pp. 236–243.

[59] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. "Protocol verification as a hardware design aid". In: *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors* (1992), pp. 522–525.

[60] Işil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. "Inductive invariant generation via abductive inference". In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications* (2013).

[61] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. "CacheAudit: A Tool for the Static Analysis of Cache Side Channels". In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 431–446.

[62] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. "CacheAudit: A tool for the static analysis of cache side channels". In: *ACM Transactions on Information and System Security (TISSEC)* 18.1 (2015), p. 4.

[63] Jules Drean, Miguel Gomez-Garcia, Thomas Bourgeat, and Srinivas Devadas. "Citadel: Side-Channel-Resistant Enclaves with Secure Shared Memory on a Speculative Out-of-Order Processor". In: *CoRR* abs/2306.14882 (2023). arXiv: `2306.14882`. URL: `https://doi.org/10.48550/arXiv.2306.14882`.

[64] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. "Efficient implementation of property directed reachability". In: *2011 Formal Methods in Computer-Aided Design (FMCAD)* (2011), pp. 125–134.

[65] Hassan Eldib, Chao Wang, and Patrick Schaumont. “Formal verification of software countermeasures against side-channel attacks”. In: *ACM Transactions on Software Engineering and Methodology* 24.2 (2014), p. 11.

[66] Hassan Eldib, Chao Wang, and Patrick Schaumont. “SMT-based verification of software countermeasures against side-channel attacks”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 2014, pp. 62–77.

[67] Steve Ellis, Ari Juels, and Sergey Nazarov. “Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks”. In: *Whitepaper v2.0* (2021). `https://research.chain.link/whitepaper-v2.pdf`.

[68] Xaver Fabian, Marco Patrignani, and Marco Guarnieri. “Automatic Detection of Speculative Execution Combinations”. In: *Proceedings of the 29th ACM Conference on Computer and Communications Security.* CCS 2022. ACM, 2022.

[69] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. “Scalable Memory Protection in the PENGLAI Enclave”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21).* USENIX Association, July 2021, pp. 275–294.

[70] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using verification to disentangle secure-enclave hardware from software”. In: *Proc. of Symposium on Operating Systems Principles (SOSP).* 2017.

[71] *Fission.io.* `https://fission.io/`.

[72] Pranav Gaddamadugu. “Formally Verifying Trusted Execution Environments with UCLID5”. MA thesis. EECS Department, University of California, Berkeley, Aug. 2021. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-200.html`.

[73] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wenmei Hwu. “Dynamic Memory Disambiguation Using the Memory Conflict Buffer”. In: *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS VI. San Jose, California, USA, 1994, pp. 183–193.

[74] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Symposium on the Theory of Computing.* 2009.

[75] Adwait Godbole, Yatin Manerkar, and Sanjit A. Seshia. “Automated Conversion of Axiomatic to Operational Models: Theoretical and Practical Results”. In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD).* Oct. 2022.

[76] Joseph A. Goguen and José Meseguer. “Security Policies and Security Models”. In: *IEEE Symposium on Security and Privacy.* 1982, pp. 11–20.

[77] Joseph A. Goguen and José Meseguer. “Unwinding and Inference Control”. In: *1984 IEEE Symposium on Security and Privacy* (1984), pp. 75–75.

[78] *Gramine*. `https://github.com/gramineproject/gramine`. 2021.

[79] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 955–972.

[80] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. 2016, pp. 368–379.

[81] Roberto Guanciale, Musard Balliu, and Mads Dam. “InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1853–1869.

[82] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. “Spectector: Principled Detection of Speculative Information Flows”. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1–19.

[83] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. “Hardware-Software Contracts for Secure Speculation”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1868–1883.

[84] *HiFive Unleashed*. `https://www.sifive.com/boards/hifive-unleashed`. 2020.

[85] Charles Antony Richard Hoare. “An axiomatic basis for computer programming”. In: *Commun. ACM* 12 (1969), pp. 576–580.

[86] Gerard J. Holzmann. “The SPIN Model Checker”. In: 2003. URL: `https://api.semanticscholar.org/CorpusID:53932627`.

[87] Jann Horn. *Read privileged memory with a side-channel*. 2018. URL: `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`.

[88] Ravi Hosabettu, Mandayam K. Srivas, and Ganesh Gopalakrishnan. “Decomposing the Proof of Correctness of pipelined Microprocessors”. In: *International Conference on Computer Aided Verification*. 1998.

[89] Guangyuan Hu, Zecheng He, and Ruby B. Lee. “SoK: Hardware Defenses Against Speculative Execution Attacks”. In: *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. 2021, pp. 108–120.

[90] *Huggingface*. `https://huggingface.co/`.

[91] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. "Chiron: Privacy-preserving Machine Learning as a Service". In: *CoRR* abs/1803.05961 (2018). arXiv: `1803.05961`. URL: `http://arxiv.org/abs/1803.05961`.

[92] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genç, and Jonathan Ragan-Kelley. "Exocompilation for productive programming of hardware accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022).

[93] Intel. *Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088*. 2018. URL: `https://software.intel.com/security-software-guidance/software-guidance/branch-target-injection`.

[94] Intel. *Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088*. 2018. URL: `https://software.intel.com/security-software-guidance/software-guidance/bounds-check-bypass`.

[95] Intel. *Deep Dive: Analyzing Potential Bounds Check Bypass Vulnerabilities*. 2018. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-analyzing-potential-bounds-check-bypass-vulnerabilities`.

[96] Intel. *Deep Dive: Managed Runtime Speculative Execution Side Channel Mitigations*. 2018. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations`.

[97] Intel. *Deep Dive: Mitigation Overview for Side Channel Exploits in Linux*. 2018. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-mitigation-overview-side-channel-exploits-linux`.

[98] Intel. *L1 Terminal Fault / CVE-2018-3615 , CVE-2018-3620,CVE-2018-3646 / INTEL-SA-00161*. 2018. URL: `https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault`.

[99] Intel. *Rogue System Register Read / CVE-2018-3640 / INTEL-SA-00115*. 2018. URL: `https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read`.

[100] Intel. *Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115*. 2018. URL: `https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass`.

[101] *Intel Trust Domain Extensions*. `https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf`. 2020.

[102] *Intel® 64 and IA-32 Architectures Software Developer Manual Volume 3*. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf`.

[103] *Intel® 64 and IA-32 Architectures Software Developer Manuals.* URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html` (visited on 12/14/2022).

[104] *Interactive guide to speculative execution attacks.* URL: `https://mdsattacks.com` (visited on 08/03/2023).

[105] G. Irazoqui, T. Eisenbarth, and B. Sunar. "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES". In: *IEEE Symposium on Security and Privacy.* May 2015, pp. 591–604.

[106] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. "Towards Formal Verification of State Continuity for Enclave Programs". In: *Proc. of USENIX Security Symposium.* 2021. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/jangid`.

[107] *Jasper Formal Verification Platform.* URL: `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html` (visited on 12/30/2022).

[108] Ranjit Jhala and Kenneth L. McMillan. "Microarchitecture Verification by Compositional Model Checking". In: *International Conference on Computer Aided Verification.* 2001.

[109] Robert B Jones, Jens U Skakkebaek, and David L Dill. "Reducing manual abstraction in formal verification of out-of-order execution". In: *Formal Methods in Computer-Aided Design: Second International Conference, FMCAD'98 Palo Alto, CA, USA, November 4–6, 1998 Proceedings 2.* Springer. 1998, pp. 2–17.

[110] Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks.* 2009, pp. 105–114.

[111] David Kaplan. *AMD SEV-ES.* `http://support.amd.com/TechDocs/ProtectingVMRegisterStatewithSEV-ES.pdf`. 2017.

[112] David Kaplan, Jeremy Powell, and Tom Woller. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`. 2016.

[113] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2018, pp. 974–987.

[114] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA, 2009, pp. 207–220. URL: `http://doi.acm.org/10.1145/1629575.1629596`.

[115] Paul Kocher. *Spectre Mitigations in Microsoft's C/C++ Compiler*. Feb. 2018. URL: `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html`.

[116] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *Proceedings of the IEEE Symposium on Security and Privacy* (2019), pp. 19–37.

[117] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018.

[118] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Lenny Truong, Gedeon Nyengele, Kecheng Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross G. Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James J. Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark W. Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. "AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers". In: *ACM Transactions on Embedded Computing Systems (TECS)* (2022).

[119] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. "Expressing Information Flow Properties". In: *Foundations and Trends in Privacy and Security* 3.1 (2022), pp. 1–102. URL: `http://dx.doi.org/10.1561/3300000008`.

[120] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. "Circuit-based Boolean reasoning". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)* (2001), pp. 232–237.

[121] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory safety for shielded execution". In: *Proc. of the Twelfth European Conference on Computer Systems (EuroSys)*. 2017.

[122] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. "Modeling and Verification of Out-of-Order Microprocessors in UCLID". In: *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*. Ed. by Mark D. Aagaard and John W. O'Leary.

Vol. 2517. Lecture Notes in Computer Science. Springer, 2002, pp. 142–159. URL: https://doi.org/10.1007/3-540-36126-X%5C_9.

[123] Michael Larabel. *Benchmarking The Work-In-Progress Spectre/STIBP Code On The Way For Linux 4.20*. 2018. URL: https://www.phoronix.com/scan.php?page=article%5C&item=linux-420wip-stibp%5C&num=1.

[124] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. "Sanctorum: A lightweight security monitor for secure enclaves". In: *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019.

[125] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. "Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1871–1885.

[126] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. "An Off-Chip Attack on Hardware Enclaves via the Memory Bus". In: *Proc. of USENIX Security Symposium*. 2020.

[127] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020.

[128] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *CoRR* abs/1611.06952 (2016). URL: http://arxiv.org/abs/1611.06952.

[129] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proc. of Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Dakar, Senegal, 2010.

[130] Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. "Verifying Linearizability of Intel® Software Guard Extensions". In: *Computer Aided Verification*. Cham: Springer International Publishing, 2015, pp. 144–160.

[131] Mingyu Li, Yubin Xia, and Haibo Chen. "Confidential Serverless Made Efficient with Plug-in Enclaves". In: *Proc. of International Symposium on Computer Architecture (ISCA)*. 2021.

[132] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. "Value Locality and Load Value Prediction". In: *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. 1996.

[133] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[134] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks Are Practical". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 605–622.

[135] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), pp. 635–646.

[136] Roger Lyndon. "An interpolation theorem in the predicate calculus." In: *Pacific Journal of Mathematics* 9 (1959), pp. 129–142.

[137] Giorgi Maisuradze and Christian Rossow. "Ret2Spec: Speculative Execution Using Return Stack Buffers". In: *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018.

[138] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. "RTLCheck: Verifying the Memory Consistency of RTL Designs". In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017), pp. 463–476.

[139] Scott McFarling. *Combining branch predictors*. Tech. rep. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[140] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. *Spectre is here to stay: An analysis of side-channels and speculative execution*. 2019. URL: `https://arxiv.org/abs/1902.05178`.

[141] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative Instructions and Software Model for Isolated Execution". In: *HASP*. 2013.

[142] John Mclean. "Proving Noninterference and Functional Correctness Using Traces". In: *Journal of Computer Security* 1 (1992), pp. 37–58.

[143] Kenneth L. McMillan. "Interpolation and Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 421–446.

[144] Kenneth L. McMillan. "Interpolation and SAT-Based Model Checking". In: *International Conference on Computer Aided Verification*. 2003.

[145] Kenneth L. McMillan. "Symbolic model checking". In: *International Conference on Computer Aided Verification*. 1993.

[146] Marcela S Melara, Michael J Freedman, and Mic Bowman. "EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments". In: *ArXiv:1907.13245* (2019).

[147] Microsoft. */Qspectre*. Oct. 2018. URL: `https://docs.microsoft.com/en-us/cpp/build/reference/qspectre?view=vs-2017`.

[148] Microsoft. *ADV180012 — Microsoft Guidance for Speculative Store Bypass*. 2018. URL: `https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV180012`.

[149] Microsoft. *Spectre mitigations in MSVC*. 2018. URL: `https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/`.

[150] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K. Brayton. "FRAIGs: A Unifying Representation for Logic Synthesis and Verification". In: 2005.

[151] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *CHES*. 2017.

[152] Federico Mora, Kevin Cheang, Elizabeth Polgreen, and Sanjit A. Seshia. *Synthesis in Uclid5*. 2020. arXiv: `2007.06760 [cs.PL]`.

[153] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. "Axiomatic Hardware-Software Contracts for Security". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 72–86.

[154] Toby Murray, Robert Sison, and Kai Engelhardt. "COVERN: A Logic for Compositional Verification of Information Flow Control". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 16–30.

[155] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 228–241.

[156] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Serval: Scaling Symbolic Evaluation for Automated Verification of Systems Code". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2019.

[157] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *Proc. of USENIX Security Symposium*. 2016.

[158] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. "You shall not bypass: Employing data dependencies to prevent bounds check bypass". In: *arXiv preprint arXiv:1805.08506* (2018).

[159] *Open Portable Trusted Execution Environment*. 2020. URL: `https://www.op-tee.org/`.

[160] *OpenFaaS*. `https://www.openfaas.com/`.

[161] *OpenTitan: Open Source Silicon Root of Trust*. URL: `https://opentitan.org/` (visited on 12/13/2022).

[162] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. "The Spy in the Sandbox - Practical Cache Attacks in Javascript". In: *CoRR* abs/1502.07373 (2015). URL: `http://arxiv.org/abs/1502.07373`.

[163] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. "Memoir: Practical State Continuity for Protected Modules". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2011.

[164] David A. Patterson and John L. Hennessy. "Computer Architecture: A Quantitative Approach". In: 1969.

[165] Colin Percival. *Cache missing for fun and profit*. 2005. URL: `https://www.daemonology.net/papers/htt.pdf`.

[166] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 565–581.

[167] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Automating Separation Logic Using SMT". In: *International Conference on Computer Aided Verification*. 2013.

[168] Amir Pnueli, Yoav Rodeh, Ofer Strichman, and Michael Siegel. "The Small Model Property: How Small Can It Be?" In: *Inf. Comput.* 178 (2002), pp. 279–293.

[169] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. "UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis". In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 538–551.

[170] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. "Satisfiability and Synthesis Modulo Oracles". In: *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*. Ed. by Bernd Finkbeiner and Thomas Wies. Vol. 13182. Lecture Notes in Computer Science. Springer, 2022, pp. 263–284. URL: `https://doi.org/10.1007/978-3-030-94583-1%5C_13`.

[171] Nelly Porter and Jason Garms. *Advancing confidential computing with Asylo and the Confidential Computing Challenge*. `https://cloud.google.com/blog/products/identity-security/advancing-confidential-computing-with-asylo-and-the-confidential-computing-challenge`. Feb. 2019.

[172] Christian Priebe, Kapil Vaswani, and Manuel Costa. "EnclaveDB - A Secure Database using SGX". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2018.

[173] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. "Verified low-level programming embedded in F". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), p. 17.

[174] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1852–1867.

[175] *Ray Serve*. `https://www.ray.io/ray-serve`.

[176] C. R. Reddy and D. W. Loveland. "Presburger Arithmetic with Bounded Quantifier Alternation". In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC '78. San Diego, California, USA: Association for Computing Machinery, 1978, pp. 320–325.

[177] John C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002), pp. 55–74.

[178] A. W. Roscoe. "CSP and Determinism in Security Modelling". In: *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*. 1995, pp. 114–127.

[179] John Rushby. "Proof of Separability: A Verification Technique for a Class of Security Kernels". In: *Proc. 5th International Symposium on Programming*. Vol. 137. Lecture Notes in Computer Science. Turin, Italy: Springer-Verlag, Apr. 1982, pp. 352–367.

[180] John M. Rushby. "Proof of separability: A verification technique for a class of a security kernels". In: *Proceedings of the International Symposium on Programming, 5th Colloquium, Torino, Italy*. 1982, pp. 352–367.

[181] *RV8 Benchmark*. `https://github.com/michaeljclark/rv8-bench`. 2017.

[182] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 238–252.

[183] Ravi S. Sandhu. "Lattice-based access control models". In: *Computer* 26 (1993), pp. 9–19.

[184] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. "Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification". In: *IEEE Access* 9 (2021), pp. 83067–83079.

[185] Jun Sawada and Warren A. Hunt. “Processor Verification with Precise Exeptions and Speculative Execution”. In: *International Conference on Computer Aided Verification.* 1998.

[186] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “RIDL: Rogue In-flight Data Load”. In: *S&P*. May 2019.

[187] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. “CacheOut: Leaking Data on Intel CPUs via Cache Evictions”. In: *2021 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 339–354.

[188] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *2010 IEEE Symposium on Security and Privacy* (2010), pp. 317–331.

[189] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* CCS ’19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 753–768.

[190] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. “NetSpectre: Read Arbitrary Memory over Network”. In: *ArXiv* abs/1807.10535 (2018).

[191] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *Proc. of Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* 2017.

[192] Sanjit A. Seshia. “Combining Induction, Deduction, and Structure for Verification and Synthesis”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051. URL: `http://dx.doi.org/10.1109/JPROC.2015.2471838`.

[193] Sanjit A. Seshia and Pramod Subramanyan. “Uclid5: Integrating Modeling, Verification, Synthesis and Learning”. In: *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE).* Oct. 2018.

[194] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. “Translation validation for a verified OS kernel”. In: *Proc. of ACM SIGPLAN Conference on Programming language design and implementation (PLDI).* 2013.

[195] J. P. Shen and M. Lipasti. *Fundamentals of Superscalar Processor Design.* McGraw-Hill, 2003.

[196] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020.

[197] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage". In: *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018, pp. 131–145.

[198] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. "Moat: Verifying Confidentiality of Enclave Programs". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[199] Jens Ulrik Skakkebæk, Robert B. Jones, and David L. Dill. "Formal Verification of Out-of-Order Execution Using Incremental Flushing". In: *CAV*. 1998.

[200] J. E. Smith and G. S. Sohi. "The Microarchitecture of Superscalar Processors". In: *Proc. IEEE* 83.12 (Dec. 1995), pp. 1609–1624.

[201] Marcelo Sousa and Isil Dillig. "Cartesian Hoare Logic for Verifying K-safety Properties". In: *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA, 2016, pp. 57–69.

[202] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels". In: *CoRR* abs/1806.07480 (2018). arXiv: `1806.07480`. URL: `http://arxiv.org/abs/1806.07480`.

[203] Ben Stuart. "Current state of mitigations for spectre within operating systems". In: *Proceedings of the 4th Wiesbaden Workshop on Advanced Microkernel Operating Systems*. 2018.

[204] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. "A Formal Foundation for Secure Remote Execution of Enclaves". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 2435–2450.

[205] Jakub Szefer. "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses". In: *Journal of Hardware and Systems Security* 3.3 (2019), pp. 219–234.

[206] Tachio Terauchi and Alexander Aiken. "Secure Information Flow as a Safety Problem". In: *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. Ed. by Chris Hankin and Igor Siveroni. Vol. 3672. Lecture Notes in Computer Science. Springer, 2005, pp. 352–367. URL: `https://doi.org/10.1007/11547662%5C_24`.

[207] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA, 2014, pp. 941–955.

[208] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette". In: *Proc. of ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming; Software*. Indianapolis, Indiana, USA, 2013.

[209] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. "CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 947–960.

[210] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *Proc. of USENIX Anual Technical Conference (ATC)*. 2017.

[211] Harvey Tuch and Gerwin Klein. "A Unified Memory Model for Pointers". In: *Logic Programming and Automated Reasoning*. 2005.

[212] Liam Tung. *Linus Torvalds: After big Linux performance hit, Spectre v2 patch needs curbs*. 2018. URL: `https://www.zdnet.com/article/linus-torvalds-after-big-linux-performance-hit-spectre-v2-patch-needs-curbs/`.

[213] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: `https://support.google.com/faqs/answer/7625886`.

[214] *UCLID5 Verification and Synthesis System*. 2022. URL: `http://github.com/uclid-org/uclid/`.

[215] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 27th USENIX Security Symposium*. See also technical report Foreshadow-NG. USENIX Association, Aug. 2018.

[216] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[217] Miroslav N. Velev and Randal E. Bryant. "Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction". In: *Proceedings 37th Design Automation Conference* (2000), pp. 112–117.

[218] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. "Efficient software-based fault isolation". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 1993.

[219] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[220] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. `https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf`. Dec. 2021.

[221] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9*. Tech. rep. UCB/EECS-2016-129. EECS Department, University of California, Berkeley, July 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html`.

[222] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`.

[223] Ofir Weisse, Valeria Bertacco, and Todd Austin. "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves". In: *ISCA*. 2017.

[224] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution". In: *Technical report* (2018). See also USENIX Security paper Foreshadow.

[225] Johannes Wikner and Kaveh Razavi. "RETBLEED: Arbitrary Speculative Code Execution with Return Instructions". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842.

[226] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[227] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. "Lazy Self-composition for Security Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II". In: July 2018, pp. 136–156.

[228] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014, pp. 719–732.

[229] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: a sandbox for portable, untrusted x86 native code". In: *Communications of the ACM* 53.1 (2010), pp. 91–99.

[230] Tse-Yu Yeh and Yale N. Patt. "Two-level Adaptive Training Branch Prediction". In: *Proc. of the 24th Annual International Symposium on Microarchitecture.* MICRO 24. 1991, pp. 51–61.

[231] Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. "Elasticlave: An Efficient Memory Model for Enclaves". In: *Proc. of USENIX Security Symposium.* 2022.

[232] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. "ShadowEth: Private Smart Contract on Public Blockchain". In: *Journal of Computer Science and Technology* 33 (May 2018), pp. 542–556.

[233] Steve Zdancewic and Andrew C Myers. "Observational Determinism for Concurrent Program Security". In: *Proc. of the 16th IEEE Computer Security Foundations Workshop.* IEEE. 2003, pp. 29–43.

[234] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. "Town Crier: An Authenticated Data Feed for Smart Contracts". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 270–282.

[235] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. "On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications". In: *Proc. of the Sixteenth European Conference on Computer Systems (EuroSys).* 2021.

[236] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. "The Move Prover". In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I.* Los Angeles, CA, USA: Springer-Verlag, 2020, pp. 137–150.