LANGUAGE–BASED TECHNIQUES FOR BUILDING TIMING CHANNEL SECURE HARDWARE–SOFTWARE SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Drew Zagieboylo

August 2023

LANGUAGE–BASED TECHNIQUES FOR BUILDING TIMING CHANNEL SECURE HARDWARE–SOFTWARE SYSTEMS

Drew Zagieboylo, Ph.D.

Cornell University 2023

We rely on a deep stack of abstractions to efficiently build software applications without having to completely understand the nuance of language runtimes, operating systems, and processor architectures. Each layer in the stack relies on the guarantees of the layer below, with all software relying on the functionality provided by the hardware on which it executes.

Similarly, when we build secure software, we define security in terms of high level application policies and rely on a stack of abstractions to enforce those policies. Therefore, all of software security relies on the guarantees provided by processor hardware. However, those guarantees offer less protection than we have traditionally assumed, and real processor implementations routinely exhibit vulnerabilities that undermine traditional assumptions about hardware behavior.

Modern processors incorporate a host of optimizations to execute software as quickly and efficiently as possible; unfortunately, these optimizations are at the root of some serious security weaknesses. In particular, researchers have recently discovered easily exploitable timing-channel vulnerabilities that arise due to processor speculation, like Spectre, Meltdown, and the many variants that have since been uncovered. Concerningly, these vulnerabilities are not the result of cutting-edge, untested optimizations; they are fundamental to the designs of almost all processors in the last 20 years.

The existence of these vulnerabilities highlights the need for a well-defined contract between software and hardware that does not allow the hardware to leak software's secrets arbitrarily, especially via timing channels. Furthermore, we need tools to enable the construction and verification of secure processors that adhere to these new contracts. As functional processor correctness is already a difficult verification problem, we likely need new approaches to prove processor security.

This dissertation addresses the above concerns by applying Information Flow Control (IFC) to both the hardware–software interface and to Hardware Description Languages (HDL) themselves. By using IFC as the de facto language of security, we can define a hardware–software contract capable of providing timing-channel security without exposing extraneous details about processor internals. Intuitively, using IFC as a tool to then build processors also enables proving that real processor implementations refine this IFC contract.

This dissertation also addresses the problem of constructing correct processors by introducing a high-level HDL that targets the design of efficient processor pipelines. By raising the abstraction of hardware design, we can more easily connect the implementation's semantics to the hardware–software contract. We can also reason statically about complex optimizations such as speculation by providing abstractions that generate correct circuitry by construction.

We hope that future processors and interfaces are designed with timing-channel security in mind, and that these new abstractions will percolate back up the software stack to make timing-channel security available and efficient for all applications.

BIOGRAPHICAL SKETCH

Drew Zagieboylo received his Bachelors degree in Computer Science from the University of California - Berkeley in 2014 and then worked as a software engineer for Electronic Arts until 2017. In addition to research, Drew has dedicated a significant portion of his time at Cornell teaching not only computer science but also rock climbing with the Cornell College of Outdoor Education. Drew received his Master's of Science in Computer Science in 2020 and was awarded his PhD in 2023.

For my parents who have always supported me and made me who I am today

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Andrew Myers, for his intellectual guidance and unfailing confidence in me; he has always believed in me and my abilities more than I ever have myself. I am also grateful for the opportunities he has created for me to learn and explore new ideas. When I arrived at Cornell I had little background in programming language theory; Andrew gave me the time, opportunity, and resources I needed to learn to become an expert in the area. Andrew has also been a great marketer, always promoting me, encouraging me, and enabling me to network and communicate my work to other researchers.

I am also thankful for the assistance of the rest of my special committee, Edward Suh and Adrian Sampson. Ed has often filled the role of a co–advisor for me and has been incredibly supportive and thoughtful. I am thankful for the additional perspective he has brought to all of our discussions, both academic and personal. I am especially thankful for his invitation to work on the Hyperflow project, which started me on the path of my thesis research.

I would also like to thank Adrian for his openness and availability, despite being quite busy with his own group. Since I spent my first semester at Cornell working with him, his welcoming nature made the transition to graduate school that much easier. I also appreciate the courses and seminars that I have had the chance to take with Adrian on programming languages, compilers, and computer architecture.

Anne Bracy has been a crucial part of my Cornell experience, as I had the luck to serve as a TA with her in my first year at Cornell *and* the privilege to co-teach CS 3410 with her in my final year. She has been an excellent teaching mentor and given me so many opportunities to learn and grow. Undoubtedly, I would have had a much more difficult and less fun time becoming an educator without her guidance and friendliness.

The Applied Programming Languages group has been a great source of camaraderie and support throughout my PhD, without which I certainly would have had an unfulfilling experience. Specifically I want to thank Tom Magrino, Ethan Cecchetti, Isaac Sheff, Mae Milano, Rolph Recto, Coşku Acay, Charles Sherk, Haobin Ni, Siqiu Yao, and Silei Ren. It has always been reinvigorating and often hilarious to attend our weekly group meetings or have spontaneous discussions in the Systems Lab.

I also want to thank a few other people who have made the PhD experience less lonesome and more meaningful: Coşku Acay for our many heated debates, long nights gaming, and shared dinners; Rachit Nigam for his friendship and our heartfelt discussions; Katie Van Koevering and John Paul Ryan for numerous hours playing D&D and their infectious wonderful excitement; Sebastian Birthelmer and Jessica Moy for their continuous support and unconditional friendship; Andrew McLaughlin, Brian Doscher, Nolan Miller, and the Cornell College of Outdoor Education for giving me a home in the Ithaca climbing community.

Lastly, I want to thank my partner, Adrienne, for inspiring me in all aspects of my life. It is so easy to lose yourself in PhD research and she has always kept me grounded in the things we love and in belief in myself. Thank you so much for everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software security is a critical issue in the ubiquitous computer systems that we trust to communicate, store our data, do business, and generally rely on for almost every aspect of our lives. One of the central directives of software security is *confidentiality*: preventing unauthorized parties from observing secret data. Confidentiality, especially of critical information like passwords and cryptographic keys, is necessary to enforce of many application security policies.

Unfortunately, confidentiality has proven difficult to formally guarantee (or even practically enforce) thanks in significant part to timing side-channels. side-channels are unintended means of observing or communicating information which bypass security mechanisms. As the name would suggest, timing side-channels can leak information through the time an operation takes to complete. While timing channels are a well-studied problem, recent discoveries have revealed that optimizations fundamental to modern hardware design have also ingrained subtle and yet exploitable timing vulnerabilities into most modern processors. These vulnerabilities are known as *transient execution attacks* (or speculative execution attacks) since they exploit the speculative nature of high performance processors and the presence of transiently executed instructions. Spectre [103] and Meltdown [72] were the first examples of transient execution attacks to be discovered, but since then researchers have unearthed dozens of variants and other speculative side-channels [126].

All hardware-based timing channels (including transient execution attacks) are especially concerning since they allow attackers to bypass higher level security mechanisms; it is impossible to build and execute secure software on vulnerable hardware. Furthermore, as these vulnerabilities are baked into the silicon, it is more or less impossible to patch affected processors; the efforts to apply microcode patches have been of limited success and error prone [75]. Instead, operating systems (OS) and compiler-level defenses [67, 24] have been broadly applied to mitigate Spectre and its variants. However, these defenses are coarse-grained, accompanied by large performance overheads, brittle to future potential attacks, and even still fail to enable secure software sandboxing [79].

To build software systems which we can trust, we need stronger guarantees about the integrity and confidentiality properties of the processors that software runs on. Additionally, we need to be able to trust that real processor implementations do indeed provide their claimed security assurances, in spite of the complex optimizations that they incorporate. To this end, we cannot simply rely on one-off defenses and the status quo for processor testing; we need provably secure designs and tools that can automatically check or generate their implementations.

This dissertation addresses the challenges facing the construction of hardware–software systems with strong, timing-channel resilient security properties:

- How can we define hardware–software interfaces that enable the development of timing-channel resilient software? Can we support programs with different security and performance requirements with a single Instruction Set Architecture (ISA)?
- Is it practical to implement and verify hardware that provides the necessary guarantees? How do we develop tools to design secure hardware while still enabling key performance optimizations like speculation?

Next, this dissertation presents further background on the problem of hardware timing channels and provides a roadmap for how to answer the aforementioned questions.

1.1 Transient Execution Attacks

Transient execution attacks rely on the fact that modern processors predict which instructions to execute next instead of waiting for potentially long latency operations. These processors delay finalizing, or committing, instructions until they are sure that it was not the result of an incorrect prediction. In this way, speculation has no visible effect on the *architectural* state: the registers and memory which are readable and writable via software. Nevertheless, mispredictions still end up leaving evidence of those incorrect instructions inside the processor's *microarchitectural* state: the internal processor structures used to implement the ISA and optimize performance. These changes to the microarchitecture affect the time at which instruction commitment takes place. Transient execution attacks first cause the processor to speculatively access confidential information and then use well-known timing channels to then extract it from the microarchitectural state.

The canonical example exploiting Spectre vulnerabilities allows software to read memory outside of a software sandbox by effectively ignoring a bounds check on an array.

```
if (x < a.len) {
  b = a[x]; // speculatively out of bounds
  c = y[b]; // can leak b via cache timing channel
}
```

This "bounds check bypass" is the poster child for transient execution attacks. Most memory accesses are in bounds and thus the processor is likely to predict that the branch should be taken. Given this prediction, the memory access on line 2 will be speculatively executed even when it is outside the bounds of `a`. That value (`b`) can then be leaked via well-known cache timing channels. When line 3 executes, the address computed using `y` and `b` will be pulled into the cache, and the attacker can probe the `y` array later to determine the value of `b`.

Even this simple attack allows malicious JavaScript to escape a web browser's sandbox, and the litany of variants [79] make total defense impossible within a single process. Unfortunately, recent research [16] has shown that building completely timing-channel-free caches would only begin to mitigate the issue; the exploitability of subtle timing channels within instruction scheduling and microarchitectural resource contention hint at the vast depths of unexplored potential speculative vulnerabilities.

In addition to effectively reading any memory location that they can address [103], attackers can also use speculative execution to subvert the integrity guarantees of secure enclaves [116] and to undermine the techniques we use to program secure cryptographic libraries [25]. These weaknesses make it impossible to secure software without restrictive and pessimistic assumptions about processors' speculative behavior. Even worse, the software mitigations employed on one processor might not work on another or on future chips, despite their pessimism! We need more precise semantics with respect to speculative behavior and timing channels than traditional ISAs provide. Future hardware–software interfaces need to establish clear and usable limits on what data may or may not influence timing both in general and under speculative execution.

1.2 Hardware–Software Contracts For Security

The root of trust for all systems is, in the end, the hardware on which the software executes. For an operating system to provide process isolation, it relies on the memory management unit to enforce page table permissions. Privilege rings and system calls enable manipulation of trusted state via tightly controlled entry points and protect the OS from processes. However, these abstractions fail to encapsulate timing channels and the threats that they pose.

The timing behavior of processors is left intentionally abstract. The (quite reasonable) logic behind this choice is that it enables development of new optimizations in the processor itself without changing the software interface. Nevertheless, in practice, software developers routinely break this abstraction barrier to improve performance, for instance to take best advantage of the processor's caches. Similarly, attackers leverage undocumented behavior of branch predictors and caches to carry out transient execution attacks.

As time has gone on, processor vendors have realized the utility of breaking the traditional ISA abstraction barriers with instructions that manipulate what has historically been considered microarchitectural state. For example, the `clflush` instruction clears the cache hierarchy state. Traditionally one considers the cache part of the microarchitecture; different implementations of the same ISA may have differently structured caches. One the one hand, without primitives like `clflush` there would be no way for software to isolate two security domains from side-channels in current processors. On the other, the semantics of these instructions do not actually provide any guarantees about timing-channel freedom. They are *state-centric* and imperative; they only indicate *how* the microarchitecture should be changed and do not qualify *what* the microarchitecture may influence.

Taking `clflush` as an example, we can see that its functionality is useful and to some degree necessary for security, but it is wholly insufficient. There are no guarantees that the latency of the `clflush` instruction itself does not create a timing channel, nor does it prevent other memory-access based channels. Ironically such primitives can sometimes even be used by attackers to create more reliable timing channels! We argue in this dissertation that ISAs must provide *timing-centric* guarantees that enable software to control or at least comprehend when and what information might be leaked via timing channels.

1.3 Information Flow Control

This dissertation adopts Information Flow Control (IFC) [38, 139] to build abstractions with end to end security guarantees. IFC is a technique with a wide range of applications as it tracks how data within the system influence each other. In the realm of security, IFC has been well-established as a means of enforcing both confidentiality and integrity policies. IFC can either be statically checked [39] as a form of formal verification, or dynamically enforced with a run-time monitor [105].

The gold standard confidentiality and integrity property provided by IFC systems is *noninterference*. Intuitively, noninterference says that you can freely alter some subset of your system without changing the behavior that is visible to some observer. For instance, confidentiality can be defined by declaring that secret inputs will not influence public outputs. In this way, noninterference effectively says that your system behaves the same as one where the data you wish to protect is explicitly partitioned from the rest of the system. Since real systems often violate noninterference, many other IFC conditions have been proposed that weaken noninterference but maintain principled definitions of security [84, 26].

We utilize IFC to define hardware–software interfaces with extensional security guarantees. In this way, we can still maintain an abstraction barrier that allows many different hardware microarchitectures while still enforcing useful and strong guarantees about how secret or untrustworthy data may affect execution. Unlike most prior work, we propose IFC models where real time is exposed to the attacker. This choice puts an end to timing side-channels; once time is part of the security contract, it can no longer be exploited. This dissertation will show how to define such strong contracts while both maintaining a flexible programming model for software, and also allowing critical hardware optimizations like speculation.

1.4 Contributions

This dissertation describes three broad contributions to the problems laid out so far. The subsequent chapters each address the following contributions, respectively:

- Chapter 2 explores how to design a hardware–software interface that provides strong, timing-sensitive security conditions. We propose that security should be a first-order concern for ISA design, rather than tacked on ad hoc just to support OS features.

The contract we propose models the processor as an IFC monitor; the software is responsible for specifying the security policies on architectural state, and the hardware is responsible for enforcing those policies. Importantly, these policies allow software to dictate which architectural state is allowed to influence the timing of execution. This chapter expands upon the formal security guarantees of this ISA, how it enables software to trade off security and performance smoothly, and the guidance it provides to hardware designers to build secure processors.

- The research in Chapter 3 is motivated by the question: "How can we statically reason about the behavior of speculative processors?" In trying to prove that synthesizable descriptions of circuits actually fulfill the hardware–software contract proposed in Chapter 2, we found that reasoning about speculative execution quickly became intractable. The root cause of this difficulty was that the register transfer level (RTL) abstraction adopted by most hardware description languages (HDL) is very low level. RTL languages do not make the high-level structure of processors explicit, and thus our tools could not leverage any of the invariants established by these structures.

  We propose a high-level HDL specifically designed to describe speculative processor pipelines which we call PDL (Pipeline Description Language). Unlike existing HDLs, PDL provides *one instruction at a time* semantics: a processor's behavior in PDL is equivalent to one that executes instructions sequentially, in isolation. However, PDL enables the creation of efficient, parallel execution of instructions via abstractions for common microarchitectural optimizations including: pipelining computation across clock cycles; efficient resolution of data dependencies between instructions; and speculative instruction execution.

We show that PDL enables swiftly describing a series of different microarchitectures that implement the same ISA and make use of interesting performance optimizations. In future work, we hope to show how to use PDL's high-level abstractions to easily verify security properties of the specified designs.

- Chapter 4 is motivated by a similar question to the previous chapter: "How can we verify information flow properties of speculative circuits?" Although we designed PDL to build secure speculative processors, some *components* of processors will still need to be built in lower level languages like SystemVerilog [1]; at the very, least PDL relies on Verilog libraries to implement its own abstractions.

  We incorporate *erasure labels* into an existing IFC RTL language, and then show how they can be used to statically track the influence of speculative execution. We show that erasure labels allow us to statically check that a speculative processor obeys a strong notion of security called Speculative Noninterference. This condition ensures that programmers only need to reason about the architectural behavior of their program to understand what information might leak through timing channels; they *do not* need to understand what they processor may or may not do speculatively.

  Lastly, we implement number of different processor modules central to high performance out-of-order processors in our language. These modules often exhibit transient execution vulnerabilities; our implementations are guaranteed to be free of such weaknesses and required minimal effort to convince our type checker that they were secure.

CHAPTER 2

AN INFORMATION FLOW ISA FOR CONTROLLING TIMING CHANNELS

2.1 Introduction

While timing channels have been well known to the security community for decades, recent hardware-based exploits attest that these vulnerabilities remain unsolved problems. For example, the Spectre, Meltdown, and Foreshadow attacks allow unprivileged processes to learn secrets by timing memory accesses [103, 72, 116]. The sophisticated security mechanisms provided by these modern processors—privilege rings, memory management units, and software guard extensions [80]—are completely undermined by uncontrolled timing behaviors. Current processors are not timing-safe.

The hardware-security community has investigated how to eliminate timing channels from circuit implementations, but these are not panaceas. Hardware description languages (HDLs) such as SecVerilog [139] and Caisson [71] provide timing-sensitive noninterference. They ensure that the time at which "public" state is updated does not depend on any "secret" state. While they do provide useful primitives for implementing secure processors, these languages are not sufficient for executing timing-safe software in a real-world setting. They can preclude necessary operations (such as modifying security labels at run time) and limit software's ability to specify security policies by baking those policies into the hardware. In practice, software needs the ability to make application-level policy decisions while still benefiting from the timing-sensitive guarantees of security-focused HDLs. On the other hand, more complex instantiations of secure processors lack proofs that their ISAs enforce a meaningful security condition. The Hyperflow processor [47], for instance, allows bounded software modification of the "context label", but no ISA-level security condition gives guidance on how safe this is.

Software attempts to eliminate timing channels have had some success but ultimately are not comprehensive, instead targeting empirically known sources of timing variation. For example, compilers for cryptographic computation[5, 14, 122] help to mitigate side-channels but are fundamentally incomplete, since they only model well known sources of timing variation such as branching and caching. To fully remove timing channels, a new interface is needed to constrain how hardware state influences timing and which software instructions might leak information [137, 49].

The missing link between these hardware and software approaches is an Instruction Set Architecture (ISA) with an explicit abstraction for the influence of the machine state on timing. With such an ISA, strong timing-sensitive security conditions could be proved about software, relying on the guarantees made by hardware.

As a straw man, a software–hardware contract might ensure that all instructions with secret operands execute in constant time. In fact, existing techniques for securely implementing cryptography implicitly assume such a contract. However, constant time inevitably means worst-case time, in general, so such a contract has daunting implications for the performance of memory operations. We argue that this kind of contract is unnecessarily restrictive. It is not necessary that such instructions take constant time; it is only necessary that the time taken does not leak information.

This chapter presents an ISA design that can be the interface connecting high-level timing-sensitive software abstractions to low-level timing-safe processor implementations. Our ISA is based on information flow control (IFC), which means our software–hardware contract is a set of IFC properties, rather than a prescriptive set of implementation behaviors such as forcing certain instructions to take constant time. Because the interface is based on IFC, it is possible to formally prove that only permitted information affects timing.

Our ISA design includes features to avoid being overly restrictive, as IFC systems often are [95]. To this end, it includes downgrading operations that allow software to endorse untrusted inputs and to declassify secret data. We also allow software to specify its own timing security policy, which permits trading off timing-channel protection for performance. Both of these features are limited so that they cannot be abused by attackers to undermine the security guarantees of well-behaved programs. We additionally include security primitives that are required to implement a practical operating system. These instructions are analogous to traditional *system calls*, but they are designed to prevent unexpected information leakage.

The ISA in this chapter tackles these goals with novel constructs and stronger formal security assurance:

- The ISA dynamically enforces timing-sensitive nonmalleable information flow [26], while also preventing implicit flows created by checking mutable labels.
- The ISA allows software to control the level of timing-channel protection. The ISA can be used to eliminate timing channels, mitigate timing channels with bounded information leak using predictive mitigation [137],

or enforce nonmalleable information flow control without timing-channel protection.

- The ISA also includes novel instructions for implementing privilege changes to emulate the functionality of system calls while maintaining nonmalleability.
- The ISA is accompanied by formal, proved security guarantees for programs implemented with it.
- We also formally specify security conditions with which hardware implementations must comply to ensure security of the ISA.

The chapter proceeds as follows. Section 2.2 presents background on security labels and our attacker model. Section 2.3 sketches our approach to controlling timing channels. Sections 2.4 and 2.5 formalize the ISA and discuss its novel features in detail. In Section 2.6, we discuss the security conditions assumed of the hardware and the practical challenges in realizing those policies with modern HDLs. Section 2.7 presents the security results for this ISA and brief sketches of their proofs. Section 2.8 uses example code to demonstrate use of the ISA. In Section 2.9 we discuss related work and we discuss future work in Section 2.10.

2.2 Background

Our ISA both extends the RISC-V ISA[1] [121] with new instructions and modifies the semantics of existing instructions. RISC-V has instructions for computing

[1]Our approach is not specific to RISC-V and could be adapted for use in other instruction sets.

$$\begin{aligned}
&(i,c)^{\rightarrow} \triangleq c \\
&(i,c)^{\leftarrow} \triangleq i \\
&l_1 \sqsubseteq l_2 \overset{\triangle}{\Leftrightarrow} (l_1^{\leftarrow} \sqsubseteq l_2^{\leftarrow})(l_2^{\rightarrow} \sqsubseteq l_1^{\rightarrow}) \\
&l_1 \sqcup l_2 \triangleq ((l_2^{\rightarrow} \sqcup l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcap l_2^{\leftarrow})) \\
&l_1 \sqcap l_2 \triangleq ((l_2^{\rightarrow} \sqcap l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcup l_2^{\leftarrow})) \\
&\bowtie(i,c) \triangleq (c,i)
\end{aligned}$$

Figure 2.1: Security lattice operators

on data, moving data to and from memory, and for changing program control flow. *Architectural* state refers to any storage location that is explicitly accessible or modifiable by software, including the 32 general-purpose registers, the program counter and all memory locations. Our extension modifies all architectural state to be associated with a security label. All other hardware state is considered *microarchitectural* and affects only the performance of software but not its functional behavior.

The complete RISC-V ISA has many Control Status Registers (CSRs) which are considered architectural, but for brevity we omit most of them from our formalization. These CSRs should in principle also each have their own security labels.

2.2.1 Security Labels

As in most IFC systems, our security labels form a lattice that supports a "flows to" relation $\sqsubseteq$, a lattice join $\sqcup$ and a lattice meet $\sqcap$. We use the phrase "more restrictive" to refer to labels higher in the lattice ordering (e.g., $a \sqsubseteq b$ means "b is at least as restrictive as a"). Figure 2.1 defines useful and mostly standard notation for label reference and manipulation. The label lattice is a product of two other lattices, one for integrity (trustworthiness of data) and one for confidentiality (secrecy of data), so a lattice element is a pair (i, c). For generality, we represent the two component lattices abstractly, but we restrict them to be dual lattices over the same carrier set. That is, the ordering $\sqsubseteq$ is reversed for the integrity and confidentiality components of the label lattice. The *reflection operator* $\between$, used for controlled downgrading, swaps the two components of a lattice element.

An illustrative instantiation of this lattice is for the component lattice elements to represent principals. For instance, component b could represent both *Bob's* integrity (data written by Bob) and *Bob's* confidentiality (data readable by Bob), where *Bob* is a user of the system. Bob's data can flow to anywhere that has a label at least as confidential and no more trusted than b. Suppose there is a principal $\top$ that is least in the integrity ordering (meaning that it is trusted by everyone) and greatest in the confidentiality ordering; conversely, $\perp$ is highest in the integrity ordering (meaning that it is untrusted) and least in confidentiality. Then data labeled $(\top, b)$ flows to the label $(b, \top)$ because in integrity we have $\top \sqsubseteq b$ and in confidentiality, $b \sqsubseteq \top$.

2.2.2 Downgrading

Downgrading is the act of lowering the label of data in the lattice, violating the normal direction of information flow expressed by the lattice ordering. While downgrading greatly improves expressibility, it is important to constrain it, so that an attacker cannot leverage the downgrading mechanism to extract more secrets or modify more trusted state than the application developer intended. Our ISA enforces nonmalleability, a form of constrained downgrading, defined by Cecchetti et al. [26]. Nonmalleability guarantees both *robust declassification* and its dual *transparent endorsement*, which respectively constrain the downgrading of confidentiality and integrity.

We define *compromised* labels to represent exactly the set of labels that can never be safely downgraded under nonmalleability.

Definition 2.1 (Compromised Labels)**.** *A label is compromised if it is not as trusted as it is secret:*

$$l \not\sqsubseteq \text{⧖}(l)$$

Intuitively, compromised data contains secret information but has been modified by an attacker or other low-integrity source. Allowing such data to be downgraded opens up the possibility of "confused deputy" style attacks, where trusted code that executes downgrades can be tricked into downgrading arbitrary data.

2.2.3 Attackers

We represent attackers by the maximal integrity i_A with which they can act and a minimal confidentiality c_A that they cannot observe. This is equivalent to typical attacker definitions which use a *maximal confidentiality* c_M the attacker can observe. Since we assume a finite lattice, we can translate c_M to c_A as follows:

$$L_s = \{l \mid l \not\sqsubseteq c_M\}$$

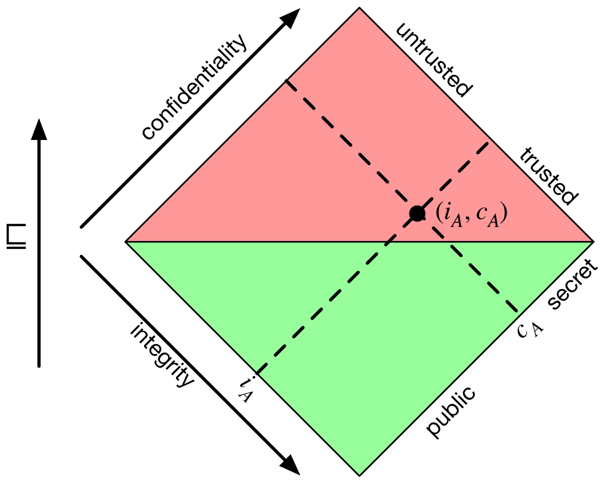


Figure 2.2: A 2-D slice of the combined confidentiality and integrity lattice. The red section represents all compromised labels. The dotted lines represent valid boundaries specifying a particular attacker model and dividing the lattice into quadrants. The intersection of these lines must be a *compromised* label, but need not be the same in each component lattice.

$$c_A \equiv \bigvee_{l_s \in L_s} l_s$$

c_A represents the disjunction of all labels which c_M is not allowed to read, and therefore defines the minimal confidentiality that they cannot observe.

It is convenient to summarize the attacker as a single label $A = (i_A, c_A)$. As depicted in Figure 2.2, the components c_A and i_A define upward-closed sets of secret and untrusted labels:

$$\mathcal{S} = \{l \mid c_A \sqsubseteq l\}$$

$$\mathcal{U} = \{l \mid i_A \sqsubseteq l\}$$

The sets of public ($\mathcal{P}$) and trusted ($\mathcal{T}$) labels are simply any labels not in $\mathcal{S}$ or $\mathcal{U}$, respectively. Attackers can only read public data and can only write to untrusted data.

Fair Attacks. Similar to prior work on robust declassification [84], our security guarantees hold against *fair attacks*, where high-secrecy and high-integrity information are only protected from attackers that do not already know those secrets or are not already highly trusted. In this work, fair attacks are defined as those where A represents a compromised label:

Definition 2.2 (Fair Attacker). *Attacker $A = (i_A, c_A)$ is a* fair attacker *if and only if A is a* compromised label*.*

Since a given attacker may be partly trusted with respect to integrity and confidentiality, the label A is not a fixed, known label. Rather, we consider the system to be secure if it is secure against all possible fair attackers A.

Our earlier *Bob* example can illustrate why this definition eliminates unfair attackers. In a security lattice including the orderings $(\top, \bot) \sqsubseteq (b, b) \sqsubseteq (\bot, \top)$, consider the attacker with *Bob's integrity* who is only allowed to read fully public data: $A = (b, b)$.[2] A is *not* a fair attacker: it is as trusted as *Bob* (and can therefore impersonate him) but is not supposed to learn any of *Bob's* secrets. Essentially, this A would model *Bob* attacking himself. Our security condition does not prevent *Bob* from mistakenly releasing his own data to the public; it prevents untrusted attackers from doing so and from manipulating *Bob* into doing so for them.

Other Assumptions. We assume a strong attacker that may observe the wall-clock time at which writes to public locations occur, and not just the ordering of writes. This observational power corresponds to a colocated attacker-controlled process that can race on memory accesses and has access to wall-clock time.

[2]Note that this label is *not compromised* since $(b, b) \sqsubseteq (b, b)$

```
# s0: secret int, a0: public int[], a1: public int
add s1, a0, s0         # s1 = &(a0[s0])
lw s2, 0(s1)           # s2 = *s1
lw a1, 0(a0)           # a1 = a0[0]
```

Figure 2.3: Meltdown-style timing channel via microarchitectural state

Defending against such a strong attacker is preferable since it makes the security assurance correspondingly stronger.

Since our ISA implements a dynamic IFC system, attackers can observe the labels of data through the success or failure of run-time checks [15]. For example, if secret (S) is used (either directly or implicitly through branching) to label another piece of data (D) as secret, then an attacker may learn information about S when their attempt to read D fails. The ISA does not include instructions for explicitly reading labels and therefore we assume attackers cannot directly read label values.

2.3 Controlling Timing Channels

Here we present high-level examples of where timing channels arise and how we approach mitigating them. Figure 2.3 contains RISC-V code with a simple microarchitectural timing channel: a secret-dependent load causing cache interference. In this example, `s0` is a secret value; `a0` and `a1` are public information. In modern processors, `lw` ("load word") is not a constant-time operation; its duration depends primarily on the address being accessed and other microarchitectural state (notably the cache). In this case, the address depends on `s0`, a secret offset into array `a0`. Loading the data at address `s1` also causes some

```
# l0,l1,l2: public int
# h1,h2:     secret-trusted int
# secret:    secret-trusted boolean
l0 = l1
if (secret):
  h1 = l1;
else:
  h1 = l2;
l0 = 1
```

Figure 2.4: Untrusted inputs causing secrets to leak via timing

region of the `a0` array to be placed in cache. If this region happens to be close to the beginning of the array, the second `lw` experiences a cache hit and executes quickly. In this way, an attacker who can observe how long it takes to load public information learns some secret information. This vulnerability reflects the core information transfer mechanism of the Meltdown attack [72].

In our ISA, software specifies a timing label, an upper bound on what information may influence instruction completion timing. If the program in Figure 2.3 executed with a secret timing label, then it would have the same unsatisfactory timing guarantees as current software. However, if the timing label were set to public, then only public information could influence how long any instruction took and the latency of the second `lw` will not reveal any information about `s0`. Obviously, software running at a low timing label may not benefit from all possible performance optimizations, but it does not necessarily require hardware to take worst-case time.

Figure 2.4 represents a different kind of timing channel, where an attacker can determine information about secrets by observing how long secret-dependent operations take. In this example, the attacker primes the cache by loading a public value, `l1`. Then, by observing when `l0` is updated, they can infer whether or not the memory read operation in between was a cache hit or miss. A hit implies that the `true` branch was taken, since `l1` was already cached.

The problem here is related to the interaction of low-integrity state with high-confidentiality computation; a cache that has been tainted with attacker-influenced state should not be allowed to influence the duration of secret operations. We incorporate this idea into our `upcall` instruction, which allows software to execute in a secret context for a predetermined amount of time. Critically, low-integrity attackers cannot `upcall` their way into learning secrets nor can they influence how trusted code execute their `upcalls`. By considering the relationship between *integrity* and *confidentiality*, we can allow programs similar to Figure 2.4 to execute safely, while disallowing variants that might leak information through timing.

2.4 Formalizing the ISA

2.4.1 Definitions and Model

In this section we present an abridged semantics for our ISA. First, we introduce the model for our semantics and some notational definitions. We represent our ISA as a small-step operational semantics on configurations.

Definition 2.3 (Configurations)**.** *A processor* ***configuration*** *represents the current state of the processor, encompassing both architecturally visible state and microarchitec-*

Table 2.1: Modified Semantics for Standard RISC-V Instructions

Insn Type	**Restrictions**	**Behavior**
COMPUTE	$pc_l \sqcup L(r_{s1}) \sqcup L(r_{s2}) \sqsubseteq L(r_d)$	$M' = M[r_d \mapsto R_{s1} \otimes R_{s2}]$
LOAD	$pc_l \sqcup L(r_{s1}) \sqcup L(M(R_{s1})) \sqsubseteq L(r_d)$	$M' = M[r_d \mapsto M(R_{s1})]$
STORE	$pc_l \sqcup L(r_{s1}) \sqcup L(r_d) \sqsubseteq L(M(R_{s1}))$	$M' = M[M(R_{s1}) \mapsto R_d]$
BRANCH	$L(r_{s1}) \sqcup L(r_{s2}) \sqsubseteq pc_l$	$pc' = (R_{s1} \otimes R_{s2})?imm : pc + 4$
JUMP	$L(r_{s1}) \sqsubseteq pc_l$	$pc' = R_{s1}$
ALL_PC	$L(M(pc_v)) \sqsubseteq pc_l \wedge pc_l \sqsubseteq \text{⧖}(pc_l)$	applies to all instructions
ALL_T	$t_l \sqsubseteq \text{⧖}(t_l) \wedge pc_l \sqsubseteq t_l$	applies to all instructions

tural state.

SW registers/memory	$M : Int \rightarrow Int$
SW label mappings	$L : Int \rightarrow Lbl$
opaque HW state	$\mu : Name \rightarrow Lbl$
program counter and label	$pc : PC = Int \times Lbl$
cycle counter and label	$t : T = Int \times Lbl$
call stack	$CST : List(PCT)$
processor configuration	$C : \langle CST, M, L, \mu, pc, t \rangle$

For simplicity, we represent both registers and DRAM as a single mapping M, in which registers are located at special addresses. Addresses are drawn from *Int*, a set of finite-size integers.[3] *Name* is a set of variable names, which can refer to locations but are not directly representable as values. *Lbl* is the set of labels representable in our lattice. For brevity, throughout this chapter we use

[3]The size of this range (for example, 32 or 64 bits) is architecture-specific.

the names of the elements of C_i, indexed with the same index i, as a shorthand for the corresponding component of C_i. For any index i, $C_i = CS_i, M_i, L_i, {}_i, pc_i, t_i$. For example, μ_1 represents the μ component of C_1. Additionally, we use pc_v to refer to the value of the pc and pc_l to refer to its label. The same convention is used for t.

In order to reason about the security label of a given piece of state in the processor, we define various conventions for looking up label values and converting integers to labels.

Definition 2.4 (Label lookup)**.** *Both architectural state and microarchitectural state are tagged with security labels. These functions describe how to determine the value of a location's label, where $i \in Int$, and $n \in Name$.*

Interpret i as a Lbl value	$\gamma(i)$
Label of location i	$L(i)$
Label of n	$\Gamma(C)(n)$

Γ is a function parameterized on processor state. This function is defined statically for a given implementation of the hardware at design time. This parameterization allows the label of any location to depend on software-specified values and/or other run-time microarchitectural state.

2.4.2 Operational Semantics

We present this ISA as a small-step operational semantics, factored into two semantics: a partial semantics specified by software instructions and an opaque hardware semantics that describes the behavior of microarchitectural state. Figure 2.5 shows the complete operational semantics for a CPU and how, in any

$$\boxed{GR \vdash \langle CST, M, L, \mu, pc, t\rangle \longrightarrow \langle CST', M', L', \mu', pc', t'\rangle}$$

$$\textsc{Execute}\ \frac{\begin{array}{c} GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST', M', L', pc', t'_l\rangle \\ GR \vdash \langle CST, M, L, \mu, pc, t\rangle \longrightarrow_{\mu} \langle \mu', t'_v\rangle \end{array}}{GR \vdash \langle CST, M, L, \mu, pc, t\rangle \longrightarrow \langle CST', M', L', \mu', pc', t'\rangle}$$

$$\textsc{Stall}\ \frac{\langle CST, M, L, \mu, pc, t\rangle \longrightarrow_{\mu} \langle \mu', t'_v\rangle}{GR \vdash \langle CST, M, L, \mu, pc, t\rangle \longrightarrow \langle CST, M, L, \mu', pc, (t'_v, t_l)\rangle}$$

Figure 2.5: Complete CPU operational semantics. These rules defer to semantics which describe how architectural state is modified ($\longrightarrow_{\mathcal{A}}$) and which describe how microarchitectural state is modified ($\longrightarrow_{\mu}$).

given time step, the CPU can update architectural state (by taking a $\longrightarrow_{\mathcal{A}}$ transition) or "stall" (from the perspective of software) by updating only microarchitectural state. While we provide the explicit semantics for $\longrightarrow_{\mathcal{A}}$ (see Figures 2.7 and 2.8), the semantics for $\longrightarrow_{\mu}$ are intentionally left unspecified because they are implementation-dependent. The architectural semantics ($\longrightarrow_{\mathcal{A}}$) do not depend upon the current state of μ since μ should not, by definition, influence the behavior of software (beyond timing). Instead, we define a set of properties that the transition function $\longrightarrow_{\mu}$ must satisfy. It is these properties that allows the ISA to offer security guarantees that current architectures lack.

Table 2.1 provides an abridged definition of instruction restrictions (also referred to as "label checks") and behavior for pre-existing RISC-V instructions. For abbreviation purposes, the notation r_x represents the index of a register specified by an instruction. To refer to the contents of the register, we write R_x, a shorthand for $M(r_x)$, the contents of the special memory location which holds that register. The symbol $\otimes$ represents some arithmetic or relational operator appropriate to the instruction in question.

In general, the restrictions on instructions prevent state with high-security labels from influencing state with low security labels. If the restrictions for a given rule cannot be met, the instruction becomes a "no-op" that increments pc_v but has no other effects. No-ops avoid leaking information through the enforcement of label checks. However, for certain errors, it is safe to jump to a special program counter, `errorpc`, while retaining the current pc_l and t_l. One such error is violation of the ALL_PC rule, which can safely cause the program to jump to `errorpc` without breaking noninterference. The full list of these errors is specified in Appendix A. At this point, any error-handling program may execute (for example, to signal termination), as long as it obeys the restrictions on normal execution. To a public observer, a program that produces an error with a secret *pc* label therefore appears equivalent to a correctly operating program.

Appendix A also lists specific rules for which label-checking operations can raise explicit errors and which require squashing via no-op. We include in our proofs that error handling does not violate our security conditions.

The COMPUTE, LOAD/STORE and BRANCH restrictions are straightforward; they ensure that instruction operands and the *pc* must flow to the destination register. The BRANCH restrictions prevent implicit flows.

The ALL_PC restriction ensures that the instruction being executed is at least as trusted and public as the *pc* itself. This constraint prevents a trusted or public program from reading instructions from secret or untrusted memory. Additionally, ALL_PC maintains the invariant that a program may execute only if it has an uncompromised *pc*. We note in §2.5 that keeping the *pc* uncompromised is required to prevent call gates from breaking nonmalleability.

```
if (s):
  upgrade(ts, UNTRUSTED)
else:
  skip
ts2 := ts
```

Figure 2.6: Leaking secrets via an integrity upgrade. Execution is successful exactly when `s` is false.

The ALL_T restriction ensures that the timing label is uncompromised and is *at least* as restrictive as the *pc* label. We summarize these restrictions as a validity condition:

$$\mathit{ISVALID}(pc_l, t_l) \triangleq (pc_l \sqsubseteq t_l) \land (pc_l \sqsubseteq \text{⧖}(pc_l)) \land (t_l \sqsubseteq \text{⧖}(t_l))$$

Intuitively, it would be difficult to implement any reasonable hardware that did not guarantee this condition. In any case where the *pc* label was more restrictive, the duration of the instruction would have to be independent of the instruction performed! This is obviously impractical for real systems, and the restriction allows us to mostly reason about pc_l when proving security conditions (see Appendix A).

2.4.3 Label Mutation

Figures 2.7 and 2.8 give the operational semantics for instructions that modify label state or that raise or lower privilege.[4] Label-mutation instructions modify the labels of memory locations. It is well known that *flow-sensitive* monitors, including this ISA[5], can leak information by modifying labels if mutation is not

[4]The R_{sn} notation refers to RISC-V style register addresses; instruction-size limitations require that the real encoding differ slightly from this notation, but it is semantically equivalent.

[5]Although this ISA is flow-sensitive, it does not have floating labels [22], and therefore labels must be explicitly changed by software instructions.

$$\boxed{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M', L', pc', t_l\rangle}$$

$$\frac{\begin{array}{c} l = L(r_d) \\ l' = \gamma(R_{s1}) \qquad RELBL(pc_l, l, l') \qquad L(r_{s1}) \sqsubseteq pc_l \qquad L' = L[r_d \mapsto l'] \end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L', (pc_v + 4, pc_l), t_l\rangle}\ \text{DWNLBL}$$

$$\frac{\begin{array}{c} l = L(r_d) \\ l' = \gamma(R_{s1}) \qquad UPLBL(pc_l, l, l') \qquad L(r_{s1}) \sqsubseteq pc_l \qquad L' = L[r_d \mapsto l'] \end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L', (pc_v + 4, pc_l), t_l\rangle}\ \text{UPLBL}$$

$$\frac{\begin{array}{c} pc'_l = \gamma(R_{s1}) \qquad t'_l = \gamma(R_{s2}) \qquad pc_l \sqsubseteq pc'_l \\ t_l \sqsubseteq t'_l \qquad ISVALID(pc'_l, t'_l) \qquad L(r_{s1}) \sqcup L(r_{s2}) \sqsubseteq pc_l \qquad \emptyset \neq CST[head] \end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (pc_v + 4, pc'_l), t'_l\rangle}\ \text{RAISELBL}$$

$$\frac{\neg INUPCALL}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (pc_v + 4, pc_l), t_l\rangle}\ \text{OTHER\_ERROR}$$

Figure 2.7: Operational semantics for label-mutating instructions given a call-gate registry *GR*.

appropriately limited [15, 22]. Since our approach involves no extra static information about the executing software, we implement the *no-sensitive-upgrade* (NSU) policy [9]. The NSU policy dynamically prevents leaks by requiring that the pc_l can flow to both the original label and the final label of the data.

However, this restriction does not eliminate all information leakage caused by label mutation. Consider the example in Figure 2.6. In this case, the label change is inside a secret context, which requires that the *pc* is secret and trusted. Register `ts` is secret and trusted and the upgrade makes it secret and *untrusted*. The label pc_l flows to both the original and final labels of `ts`, so the aforementioned rule is satisfied. Nevertheless, the final assignment (which occurs in a public context) to `ts2` will succeed in the case where `s` is false and fail otherwise since `ts` now represents untrustworthy information.

$$\frac{\begin{array}{c}\neg INUPCALL \qquad pc'_l = \gamma(R_{s1}) \\ t'_l = \gamma(R_{s2}) \qquad ISVALID(pc'_l, t'_l) \qquad L(r_{s1}) \sqcup L(r_{s2}) \sqcup L(r_{s3}) \sqcup L(r_d) \sqsubseteq pc_l \\ pc_l \sqcup t_l \sqsubseteq pc'_l \sqsubseteq t'_l \qquad endpc = R_{s3} \qquad endt = R_d + t_v \\ CST'[head] = ((endpc, pc_l), (endt, t_l)) \qquad CST'[tail] = CST\end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST', M, L, (pc_v + 4, pc'_l), t'_l\rangle}\ \textsc{upcall}$$

$$\frac{INUPCALL \qquad ((endpc, pc'_l), (endt, t'_l)) = CST[head] \qquad t_v \neq endt}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, pc, t_l\rangle}\ \textsc{upret-nop}$$

$$\frac{\begin{array}{c}INUPCALL \qquad ((endpc, pc'_l), (endt, t'_l)) = CST[head] \\ CST' = CST[tail] \qquad t_v = endt\end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST', M, L, (endpc, pc'_l), t'_l\rangle}\ \textsc{upret-done}$$

$$\frac{\begin{array}{c}\emptyset = CST[head] \qquad endpc = pc_v + 4 \\ CST'[head] = ((endpc, pc_l), (null, t_l)) \qquad CST'[tail] = CST \\ (pc', t'_l) = GR(R_{s1}) \qquad ISVALID(pc'_l, t'_l) \qquad L(r_{s1}) \sqsubseteq pc_l \qquad pc'_l \sqcup t'_l \sqsubset pc_l\end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST', M, L, pc', t'_l\rangle}\ \textsc{dwncall}$$

$$\frac{\begin{array}{c}((pc'_v, pc'_l), (null, t'_l)) = CST[head] \\ pc_l \sqcup t_l \sqsubset pc'_l \sqcap t'_l \qquad CST' = CST[tail]\end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST', M, L, (pc'_v, pc'_l), t'_l\rangle}\ \textsc{dwnret}$$

$$\frac{INUPCALL}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, pc, t_l\rangle}\ \textsc{upret\_error}$$

Figure 2.8: Operational semantics for call gate instructions given a call-gate registry *GR*.

Additionally, since label arguments themselves are labeled memory locations, we require that the label of those arguments flows to pc_l. For example, the instruction `dwnlbl x3, x6` means: "Downgrade the label of register `x3` to the label represented by the value stored in register `x6`". If the label of `x6` itself were secret, using it to change the label of `x3` in a public context could allow an observer to learn about the content of `x6`. If the label of a location whose content is used as a label does not flow to pc_l, then the instruction becomes a no-op to prevent this kind of leakage.

We introduce additional restrictions on both upgrade and downgrade rules to prevent similar kinds of information leakage; these rules differ from each other in order to be more permissive.

Upgrading. The predicate $\mathit{UPLBL}(pc_l, l, l')$ expresses the NSU check for upgrading label l to label l' in the context pc_l:

$$\mathit{UPLBL}(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqsubseteq l') \wedge (l' \sqsubseteq \between(pc_l))$$

The intuition here is that we need an upper bound for the final label to prevent it from moving to a new quadrant in the lattice. *UPLBL* deviates from the original NSU definition by adding the constraint $l' \sqsubseteq \between(pc_l)$. This prevents programs from creating untrustworthy information in secret contexts and vice versa. For the program in Figure 2.6, the `uplbl` instruction fails the *UPLBL* test, preventing the offending label modification. Unfortunately, this *still* leaks the value of `s` since the program only fails when `s` is true. The key insight for handling this case is that the failure happens while the *pc* is still in a high context, so measures can be taken to prevent a low context from observing the failure. We discuss this leakage in further detail below (§2.4.4).

```
public_val = 0
while (secret_1 < secret_2):
  # do some slow computation
  secret_1++
public_val = 1
```

Figure 2.9: Secrets may be learned from the timing of the write to `public_val`.

Downgrading. There are two different cases to consider when downgrading label l to l': $l' \sqsubseteq l$ and $l' \not\sqsubseteq l$. For the first case, the predicate $\mathit{DWNLBL}(pc_l, l, l')$ expresses the existing nonmalleable information flow restrictions when downgrading label l to label l' in the context pc_l.

$$\mathit{DWNLBL}(pc_l, l, l') \triangleq (pc_l \sqsubseteq l') \wedge (l' \sqsubseteq l) \wedge (l \sqsubseteq \text{⧖}(l))$$

The other case is the general form of downgrading, which we model as first executing a downgrade from l to $l \sqcap l'$, followed by an upgrade to l'. As one might expect, this essentially combines the restrictions from those other cases:

$$\mathit{RELBL}(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqcap l') \wedge (l \sqsubseteq \text{⧖}(l)) \wedge (l' \sqsubseteq \text{⧖}(pc_l))$$

This check implies the original nonmalleability restrictions,[6] which means it is no more permissive. An alternative for modeling general downgrades would be to simulate first an upgrade to the join and then a downgrade. That requirement implies the one we've just described and is therefore also safe. However, it may be overly restrictive. It is unclear if the two are equally permissive or if downgrade-then-upgrade permits more safe programs for our lattice definition. This question lies outside the scope of this chapter.

[6]In our setting, their requirement would roughly translate to the conditions: $l \sqsubseteq l' \sqcup \text{⧖}(pc_l \sqcup l)$ and $pc_l \sqsubseteq l'$.

2.4.4 Raising context labels

The `upcall`/`upret` instruction pair introduces primitives for controlling timing channels while branching on secret or untrusted values. The `upcall` instruction allows a process to enter a more restricted context with a higher pc_l and t_l, while pushing the current pc_l and t_l to a call stack. In the new context, the program cannot write to low outputs, but its execution timing can be influenced by high hardware state. However, returning from this context reveals timing information about the duration of the subprogram. This problem can be seen in the higher-level program shown in Figure 2.9. The low adversary is allowed to observe the time of completion for the `while` block, since it can observe the timing of the writes to `public_val`. However, the duration of this block depends upon secret values. This example shows a more general version of the label-checking termination channel from Figure 2.6.

To control timing channels, `upcall` instructions are given an absolute end time and an ending program counter as arguments. Once the end time is reached, the processor steps to the end pc_v. The instruction arguments are saved onto a hardware call stack along with the caller's pc_l and t_l. Intuitively, this semantics preserves noninterference because the subprogram cannot modify memory locations or labels in a way that changes low observations. Since the completion of the upcall is determined purely from information of at most the level pc_l, no termination channel influences subsequent program steps.

In general, this approach is impractical because it requires programmers or compilers to know cycle-accurate durations of program segments. However, it can easily be used to execute untrusted functions. The `upcall` instruction can create a low-integrity sandbox that runs until the provided timeout expires.

Using upcalls for timing mitigation. To support a more flexible programming model, we also expose a generic interface for handling returns from high contexts via an exception. When the timer completes, if the current instruction is not an `upret`, the configuration steps to a known exception handler pc_v[7]. Furthermore, when a label check fails inside of an upcall, the program simply stalls (i.e., steps to a new configuration where no architectural state has changed). Whichever of these conditions causes the exception is recorded in a status register (implemented as a CSR), with the high label of the upcall. In Figure 2.8, we use the *INUPCALL* check to specify whether or not a configuration is inside of an upcall by inspecting the head of the call stack. If *INUPCALL* is true, then the error can be handled normally, otherwise it should be squashed and the program should stall.

$$\begin{aligned} &\mathit{INUPCALL} \triangleq \\ &\quad (((\mathit{endpc}, pc'_l), (\mathit{endt}, t'_l)) = \mathit{CST}[\mathit{head}]) \\ &\quad \wedge (pc'_l \sqsubseteq pc_l \wedge t'_l \sqsubseteq t_l) \end{aligned}$$

With this primitive, the timing mitigation algorithms described in prior work [137, 8] can be implemented, enforcing bounded leakage on information from the high context. We note that this information release is still nonmalleable; both robust declassification and transparent endorsement are maintained under these mitigation mechanisms. Importantly, our restrictions prevent attackers from exploiting mitigation to exfiltrate arbitrary data.

Checking whether or not a high context subprogram failed due to violating the label check restrictions also represents a nonmalleable information release.

[7]Termination behavior can be configured on a per-program basis; it is only required that the configuration is completed using only information that is low relative to the program's original pc_l.

The data in the status register can be declassified or endorsed to reveal whether or not a label check caused the subprogram to fail. Revealing this information violates the termination sensitivity of the subprogram noninterference. Although the subprogram cannot modify any low state, information is transferred via termination.

Further upcall restrictions. `upcall` and `dwncall` instructions may not be executed inside an upcall. Intuitively, a `dwncall` (which lowers pc_l) would allow a process to produce public outputs while still inside the upcall, leaking information about its timing and progress. As mentioned, the arguments to the `upcall` instruction must also themselves be labeled so that they flow to the current pc_l. Without this requirement, secret or untrusted information could still influence the duration of the subprogram.

The nesting restriction could be relaxed to allow for multiple `upcall` instructions so that the context could be raised repeatedly. However, we do not include it in this formalism since it would complicate the requirements for hardware (call stacks would no longer have finite depth). In reality, nesting would be useful for implementing the process of control transfer from user space to operating system privileges and from there to the hypervisor level.

Permanently raising context labels. In addition to the `upcall` instruction, the pc_l and t_l can be raised by simply writing to them (they are implemented as CSRs). In order to preserve noninterference, the labels can only be raised in this way. Once raised, a program can only lower its context labels by executing a `dwncall` instruction. This limits the possible leakages caused by the program to outputs produced by the set of trusted functions which it is allowed to call.

2.4.5 Lowering context labels

The `dwncall`/`dwnret` instructions allow programs to call into more-public and more-trusted contexts via *call gates*. Call gates are essentially labeled functions that have been pre-registered by a public–trusted entity. The call-gate registry is effectively a read-only function lookup table.[8] A call gate registration contains a *pc* and t_l; using a `dwncall` instruction sets the current *pc* and t_l to the gate's values while pushing the prior values onto a call stack. These instructions provide hardware support for the privilege escalation features described in prior work on security and information flow. In particular, they closely resemble the primitives required to implement *gates* from the Multics and HiStar operating systems [99, 136]. In those systems, gates were used respectively to call known functions with higher privileges than the caller, and to implement synchronous RPC.

2.4.6 Exceptions and Asynchrony

We do not include exception configuration or handling in our ISA formalism or formal security proof. In this section, we describe how one could incorporate these features into our ISA without compromising its security conditions. All exceptions have a triggering condition and an *exception program counter* (`epc`) that points to the interrupt service routine (ISR)[9].

Trigger conditions can be specific to an ISA-extension or architecture and

[8]Using rules similar to the `uplbl` instruction, call gate entries can also be made more secret or less trusted without violating noninterference.

[9] This is not the same as the RISC-V `epc` CSR, we are paraphrasing the exception handling mechanism for clarity.

are often defined by the hardware. The `epc` is programmed by software and stored in a CSR. There are additional exception-masking CSRs which software can use to suppress the trigger conditions. In general, in order for an exception to fire, the security label of all trigger conditions (including masks) must flow to the current pc_l; otherwise, an attacker process may learn that an exception fired and deduce some secret related to its cause. For arithmetic exceptions such as integer overflow or divide-by-zero, this implies that the instruction operands flow to the current pc_l; if they don't, the exception must be suppressed. The label of the *pc* while the ISR is actually handling the exception must also be lower bounded by all trigger inputs and the label of the `epc` register itself. In this way, if an exception trigger condition is secret, its handler must be executing in a secret context and cannot produce public outputs.

We believe the primary complications involved in integrating exception handling into such an ISA are as follows. First, it is not always clear how to label exception triggers. For example, should an incoming network packet signal be labeled public or could the timing of packet arrival give an attacker information about co-resident processes? Likely, this choice should be programmable by software depending on the threat model. Secondly, depending upon how hardware state is labeled, asynchronous exceptions (such as timers and incoming network packets) may be frequently dropped or delayed. In order to account for this, the processor and ISA may need to be modified to support batched handling of exceptions along predetermined schedules within the CPU itself. Additionally, it may be difficult to limit the number of actual hardware signals that contribute to exception trigger conditions in real implementations. For example, Van Bulck et al. [117] found that Intel SGX implementations allowed the currently executing instruction to complete before handling certain exceptions.

Waiting for instruction completion means that most control signals in the CPU would influence the exception trigger conditions. It is not always possible to immediately transfer control to the ISR without waiting for some state to clear in the CPU, and thus it may be challenging to implement practical exceptions that execute in contexts that have low confidentiality or high integrity.

2.5 ISA Design Discussion

Here we highlight some salient points of our design and compare and contrast with other language-based IFC systems.

Compromised contexts and data undermine nonmalleability The original nonmalleability chapter [26] identified restrictions on downgrading that are equivalent to our observation that compromised labels cannot be downgraded to public or trusted status. We additionally notice that executing in a compromised context can unsafely leak information through timing. Specifically, this can violate the *non-occlusion* principle of declassification described by Sabelfeld and Sands [96]. Consider the scenario where `upcall` operations implement predictive mitigation, and therefore enforce nonmalleability (rather than noninterference). Allowing a process to raise its pc_l and/or t_l to a compromised level is unsound because it implicitly allows that process to declassify arbitrary data. With our restrictions, observing the duration of this subprogram leaks only the caller's secrets and is therefore robust; otherwise *any* information could be implicitly declassified via this channel.

Software can control how much information it leaks through timing channels Our ISA provides strong guarantees with respect to timing. As long as a program keeps its timing label low and executes fully low-deterministic upcalls, it leaks no information through its timing behavior. However, programs are not strictly bound by these restrictions. By explicitly exposing the pc_l, t_l and `upcall` timing to software, we grant programs the ability to weaken these restrictions gracefully to suit their needs. This provides important flexibility for situations where our threat model is overly strong or when application-specific data may only require probabilistic guarantees about timing consistency.

Limitations of Our ISA While our ISA has strong security guarantees and important security primitives, there is much room for future research. First of all, our timing label mechanism does provide a bound on which information may be implicitly leaked through timing channels. However, this coarse-grained approach could potentially leak *any* information below the timing label. This behavior is unlike the `dwnlbl` instruction, which explicitly denotes the memory location to be downgraded. Our ISA also does not incorporate explicit timing into any instructions other than `upcall`. While this lack of explicitness is beneficial for remaining implementation-agnostic, it does not give guidance on how to implement secure and efficient hardware. Yu et al. [128] describe an ISA which focuses on this performance aspect, by exposing more microarchitectural information in their ISA. Future secure ISAs and ISA extensions must be designed with both of these goals in mind, potentially leading to new semantics or completely novel timing-aware instructions.

Finally, our work only targets the single core subset of the RISC-V ISA and does not provide guidance on how to address multicore communication and interference. This realm of interconnected computing devices communicating via shared memory and coherence networks introduces many more opportunities for timing interference and side-channel communication. Investigating this problem requires a significant further effort in analyzing the semantics of existing memory models, microarchitectural coherency guarantees and how to efficiently incorporate IFC labels into these protocols.

2.6 Hardware Semantics and Properties

As mentioned earlier, an actual hardware implementation of this ISA will be a circuit that not only implements the software-visible semantics but also refines the full CPU semantics. We now discuss properties of a hardware implementation that are sufficient to guarantee the ISA-level security conditions. Additionally, we discuss the implications of these properties on hardware implementations and comment on what techniques may be utilized to verifiably construct hardware with said properties.

Property 2.1 (Deterministic Execution)**.** *For any configuration C, and for $i \in \{1, 2\}$*

$$C \longrightarrow_{\mu} \langle \mu_i, t_{vi} \rangle \implies ((\mu_1 = \mu_2) \land (t_{v1} = t_{v2}))$$

$$\land$$

$$C \longrightarrow C_i \implies C_1 = C_2$$

The operational semantics for the transition function on microarchitectural states must be deterministic. Furthermore, we assume that the full semantics which determines when to stall the processor is also deterministic.

We believe that this property can also be relaxed to allow for sources of nondeterminism (such as changes in clock frequencies, random number generators, etc.) as long as this nondeterminism is truly generated by noise or other public/trusted factors. Defining exactly what factors are public/trusted is a complex decision related to particular threat models and is out of scope for this chapter.

Property 2.2 (Single-Step Machine Noninterference)**.** *Given a set of low labels in the security lattice, $\mathcal{L}$,*

$$\begin{aligned}
&\forall C, i \in \{1, 2\}. \\
&\quad (C_1 =_{\mathcal{L}} C_2) \land (C_i \longrightarrow C_i') \\
&\qquad \implies ((\mu_1' =_{\mathcal{L}} \mu_2') \land (t_{v1}' =_{\mathcal{L}} t_{v2}')).
\end{aligned}$$

The hardware implementation must enforce a timing-sensitive noninterference condition for microarchitectural state for all transitions. With this definition, the label of t effectively bounds which hardware state may affect the timing of operations (including the decision to stall or not stall computation). The above property also implies that $\longrightarrow_\mu$ enforces timing-sensitive noninterference on μ and t. Note that this noninterference condition only applies for microarchitectural state, not architectural state. The architectural state may be downgraded using the downgrade instructions in our ISA.

The above definition of timing-sensitive machine noninterference is actually overly strong and we can substitute a slightly weaker property. t is interpreted as a global clock; however, this requirement enforces that hardware end instructions at exactly the same *real time* whenever $t_l' \in \mathcal{L}$. For most cases this isn't a problem, since $t_l \in \mathcal{L}$ and therefore both configurations start executing the instruction at the same time. It is not unreasonable for hardware to therefore ensure that they end at the same time by using only low-labeled state to influence their duration.

However, some instructions can lower t_l thereby creating a scenario where $t_l \in \mathcal{H}$ and $t'_l \in \mathcal{L}$. In our ISA, `dwncall` can create this scenario and would theoretically require that two executions always enter the call gate at the same time, even when they previously had high timing labels. Since $t_{v1} \neq t_{v2}$ there is no way for a CPU to ensure $t'_{v1} = t'_{v2}$. Luckily, real-time equivalence is not really the guarantee we need. We just need the *duration* of the instructions to be equal in both configurations, if $t' \in \mathcal{L}$. For all of the instructions in our ISA, this results in exactly the same security guarantees that we have claimed in §2.7. Below is the amended Single-Step Machine Noninterference property:

$$\forall C, i \in \{1, 2\}.$$

$$(C_1 =_{\mathcal{L}} C_2) \wedge (C_i \longrightarrow C'_i) \implies (\mu'_1 =_{\mathcal{L}} \mu'_2) \wedge (t'_l \in \mathcal{L} \implies t'_{v1} - t_{v1} = t'_{v2} - t_{v2}))$$

Property 2.3 (Computability of Label Lookups)**.**

$$\exists \Gamma,\ \forall C,\ n \in \mathrm{dom}(\mu), \Gamma(C)(n) \textit{ is computable}$$

Property 2.3 has so far been an implicit assumption. The function Γ is parameterized on all of the configuration state; it represents a function that must be computed at run time and therefore must be implemented in the microarchitecture. In combination with Property 2.2, this implies that the process of looking up microarchitectural labels does not violate noninterference [141]. It also implies that, after a configuration step $C \longrightarrow C'$, Γ determines low equivalence by evaluating labels of μ using C', not the original configuration C (we formalize low equivalence further in §2.7).

Intuitively, the above properties suggest that there is no hardware-level information flow which violates timing-sensitive noninterference except for flows that are explicitly induced by software instructions. For instance, declassifying a secret memory location, *loc*, with a `dwnlbl` instruction can only declassify microarchitectural state that specifically represents *loc's* data. We discuss the ISA-level security properties that we can derive from these guarantees in §2.7.

2.6.1 Implications for Hardware Implementations

Property 2.1 can be easily satisfied, for the most part, as processors are typically implemented as deterministic digital circuits. While some features require a notion of nondeterminism (such as random number generators or external sensor inputs), these can be modeled as the I/O to a deterministic digital circuit. In the design, one must label and build deterministic circuitry used to process these values (e.g., a buffer containing input packets from the network) but the non-determinism of the outside system has no direct impact on the security of the processor itself. As discussed in §2.4.6, this may lead to different low-level behaviors and performance characteristics in real implementations.

Furthermore, even features with unpredictable behavior can be modeled deterministically as long as their inputs are deterministic. For example, DVFS [54] modulates clock frequency during execution and can change the wall-clock time of code execution. However, if those modulation decisions are made via a digital circuit and their inputs are deterministic, we can model DVFS as software-visible architectural state and guarantee that its use does not violate our security conditions.

Property 2.2 requires a processor to be designed to remove timing channels through its microarchitecture. A recent publication [47] shows that such a tagged processor with strong control for microarchitectural timing channels and potentially reasonable overheads is feasible. Yu et al. [128] have also shown recently that it is feasible to build a modern CPU with speculation, out-of-order execution and other microarchitectural optimizations while enforcing probabilistic-noninterference [11]. These results provide evidence that it is possible to build efficient secure hardware, with the appropriate ISA abstractions.

Property 2.3 suggests that processor microarchitecture needs to be designed in a way that allows the security label of microarchitectural state to be determined. This property can be achieved by either statically labeling hardware modules at design time or by adding hardware tags to track runtime labels. Recursively, these tags are also microarchitectural state and their labels must also be computable. Therefore, real implementations will use both of these techniques (static vs. dynamic labels) since Γ is only computable if it eventually reaches a fixed point.

Our ISA provides hardware designers with the flexibility to choose how to realize timing-sensitive noninterference. For example, in order to remove cache timing channels, a processor designer may statically partition a cache; dedicate a cache to one security level and flush it when the security level is lowered; bypass the cache; or even introduce scratchpad memory with a fixed latency; any implementation will suffice so long as its timing behavior is noninterfering.

2.6.2 Enforcing Timing-Sensitive Noninterference in Hardware

For strong security assurance, we ideally want to formally enforce the properties needed for a secure hardware implementation. There exist several efforts to develop security-annotated Hardware Description Languages (HDL) that can provide timing-sensitive noninterference guarantees, similar to the one we specify here [139, 44, 10]. Previous studies show that these security-annotated HDLs can be used to express realistic security policies and implement complex circuits that satisfy them [46, 47, 70, 71].

The primary challenge with proving Property 2.2 by using secure HDLs is that these languages do not have separate notions of "architectural" and "microarchitectural" state; the entire circuit is represented as a single state machine. Phrased another way, hardware and software are concerned with different definitions of observability; in the hardware description, all state is considered observable, even though software can only directly observe architectural state. This disconnect makes proving a hardware implementation correct challenging for a few specific reasons.

First, it is impossible to prove that an implementation that supports ISA-level downgrading provides microarchitectural noninterference. Any implementation of our ISA must contain downgrades at the HDL level, which correspond to those required to implement downgrading instructions. However, the noninterference guarantees provided by these HDLs are completely obviated by including downgrades; they cannot ensure that the information being downgraded is limited only to architectural state.

A second issue with proving hardware implementations secure is the difference in label equivalence models. We assume that an attacker cannot read the value of a secret label, but can observe the fact that the label is secret. In the hardware, any location which stores a label value must itself be labeled. Given the attacker model above, it is unclear how to write down the label of this location. If we label it as public, then the HDL will allow us to define hardware that leaks the values of secret labels to attackers. If we label it as secret, then the HDL will conservatively disallow some safe label checking operations.

We believe that these problems may be solved by applying prior techniques for verifying CPU correctness (such as Pipecheck and RTLCheck [77, 78]). Moreover, these approaches could be augmented with formal verification tools specifically designed for IFC. For instance, Nickel [102] is a framework for proving noninterference that uses application specific definitions of observational equivalence. Investigating how to utilize these approaches to prove microarchitectural noninterference while supporting software-level downgrading and notions of observability is an interesting research question that has yet to be fully demonstrated.

2.7 ISA Security Properties

This section describes some of the security properties of this ISA and their performance and usability tradeoffs.

Low Equivalence. We start by formalizing the low equivalence of configurations, relative to a set of low labels, $\mathcal{L}$. This models the ability of an observer

$$\begin{aligned}
pc_1 =_{\mathcal{L}} pc_2 &\iff ((pc_{l1} \wedge pc_{l2}) \notin \mathcal{L}) \vee (pc_1 = pc_2)\\
t_1 =_{\mathcal{L}} t_2 &\iff ((t_{l1} \wedge t_{l2}) \notin \mathcal{L}) \vee (t_1 = t_2)\\
L_1 =_{\mathcal{L}} L_2 &\iff (L_1 \approx L_2) \wedge (\forall j \in \text{dom}(L).\quad L(j) \in \mathcal{L} \implies L_1(j) = L_2(j))\\
M_1 =_{\mathcal{L}} M_2 &\iff (L_1 \approx L_2) \wedge (\forall j \in \text{dom}(M).\quad L(j) \in \mathcal{L} \implies M_1(j) = M_2(j))\\
\mu_1 =_{\mathcal{L}} \mu_2 &\iff \Gamma(C_1) \approx \Gamma(C_2) \wedge\\
&\qquad \forall n \in \text{dom}(\mu).\quad \Gamma(C)(n) \in \mathcal{L} \implies \mu_1(n) = \mu_2(n)\\
CST_1 =_{\mathcal{L}} CST_2 &\iff CST_1 \approxeq_{\mathcal{L}} CST_2\\
C_1 =_{\mathcal{L}} C_2 &\iff (pc_1 =_{\mathcal{L}} pc_2) \wedge (t_1 =_{\mathcal{L}} t_2) \wedge (M_1 =_{\mathcal{L}} M_2)\\
&\qquad \wedge (\mu_1 =_{\mathcal{L}} \mu_2) \wedge (CST_1 =_{\mathcal{L}} CST_2)
\end{aligned}$$

Figure 2.10: Low Equivalence of Configuration Components, relative to "low" labels, $\mathcal{L}$.

who can only differentiate between low states; two low-equivalent configurations appear identical to a "low observer". First, we define an equivalence operator on label mappings to formalize our notion that attackers cannot observe exact label values.

Definition 2.5 (Label Lookup Domain Equivalence)**.** For an attacker inducing label sets $\mathcal{P}$, $\mathcal{S}$, $\mathcal{U}$, and $\mathcal{T}$

$$L_1 \approx L_2 \iff \forall n \in \text{dom}(L).$$

$$(L_1(n) \in \mathcal{P} \iff L_2(n) \in \mathcal{P}) \wedge$$

$$(L_1(n) \in \mathcal{T} \iff L_2(n) \in \mathcal{T})$$

We define the $\approx$ relation on the labels of microarchitecture similarly.

Figure 2.10 shows the definition of low equivalence for all configuration components. We assume that L, M, μ and Γ are total functions so that domain equality is implicit. The requirements of low equivalence explicitly require that "label lookups" for both architectural and microarchitectural state return equivalent but not equal values for high labels. Call stack low equivalence requires that all entries with low pc_l are in the same position in the stack and are themselves low-equivalent. By construction, all low entries must be at the head of the stack[10] so it is sufficient to check that the low prefixes of each call stack are equivalent.

Definition 2.6 (Call Stack Prefix Low Equivalence)**.**

$$
\begin{aligned}
& CST_1 \cong_{\mathcal{L}} CST_2 \iff \\
& \qquad (1) \quad CST_1 = \emptyset \land \forall (pc^i, t^i) \in CST_2, pc^i \in \mathcal{H} \\
& \qquad \quad \ \textit{or} \\
& \qquad (2) \quad CST_2 = \emptyset \land \forall (pc^i, t^i) \in CST_1, pc^i \in \mathcal{H} \\
& \qquad \quad \ \textit{or} \\
& \qquad (3) \quad CST_1[head] = (pc_1, t_1) =_{\mathcal{L}} (pc_2, t_2) = CST_2[head] \\
& \qquad \qquad \land CST_1[tail] \cong_{\mathcal{L}} CST_2[tail]
\end{aligned}
$$

Security Guarantees. All of the theorems in this section have full proofs, which can be found in Appendix A.2 First, we show that executing programs that do not contain downgrade or call-gate instructions preserve noninterference.

We use the term *valid* configurations to refer to configurations that were initialized with reasonable values. Specifically, the configurations satisfy the ALL_PC and ALL_T requirements and the initial call stacks are empty.

Theorem 2.1 (Noninterference Modulo Downgrading and Call Gates)**.**

[10]This is enforced by preventing `dwncall`s while inside of an `upcall`.

For any two valid configurations, C_1 and C_2 and any low set of labels, $\mathcal{L}$, where no instruction is a `dwnlbl`*,* `upcall`*,* `upret-done`*,* `dwncall` *or* `dwnret`*:*

$$(C_i \longrightarrow^* C_i^{'}) \wedge (C_1 =_{\mathcal{L}} C_2) \implies C_1^{'} =_{\mathcal{L}} C_2^{'}$$

where $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$.

The proof is a straightforward structural induction on the operational semantics of the processor. By assuming Property 2.2, essentially all of the work in this proof requires proving noninterference of the $\longrightarrow_{\mathcal{A}}$ semantics.

We next extend Theorem 2.1 to prove noninterference even when using `upcall` instructions.

Theorem 2.2 (Noninterference Modulo Downgrading)**.**

For any two valid configurations, C_1 and C_2, and any low set of labels, $\mathcal{L}$, where no instruction is a `dwnlbl`*,* `dwncall` *or* `dwnret`*.*

$$(C_i \longrightarrow^* C_i') \wedge (C_1 =_{\mathcal{L}} C_2) \implies C_1' =_{\mathcal{L}} C_2'$$

In the scenario covered by Theorem 2.1, once the pc_l was high, it could never be lowered again. That makes the noninterference proof trivial but also limits functionality. To prove Theorem 2.2, we show that all operational steps taken while an `upcall` is on the call stack can be modeled as a single operational step to low-equivalent configurations. We can show this since the end configuration of the upcall is predetermined by low-equivalent state and high pcs are noninterfering (i.e., programs executing with a high pc cannot modify any low visible state).

Note that while this theorem is termination-sensitive , it is not timing-sensitive. In the case where $t_l \not\sqsubseteq pc_l$, attackers may make observations about high state based on the timing of writes to low state. We present a corollary that provides timing sensitivity.

Corollary 2.1 (Timing-Sensitive Noninterference Modulo Downgrading)**.**

If ($pc_l \in \mathcal{L} \implies t_l \in \mathcal{L}$) for all intermediate configurations and `upcall` *regions have fixed durations, then Theorem 2.2 provides timing sensitivity.*

This corollary ensures that at any time that low writes are possible, the attacker observes them occurring at the same time in any execution. Furthermore, the duration of high call gates will be determined by low information.

As defined in §2.2, nonmalleability is essentially defined as maintaining both robust declassification and transparent endorsement. Even with no syntactic restrictions (unlike the prior theorems) our ISA enforces nonmalleability.

Theorem 2.3 (Nonmalleable Information Flow)**.** *For attacker-induced high label sets $\mathcal{S}$ and $\mathcal{U}$ and their respective complements, $\mathcal{P}$ and $\mathcal{T}$ and valid configurations, $\forall \{s, u\} \in \{1, 2\}, C_{su}$*

$$((C_{su} \longrightarrow C'_{su})\ (C_{1u} =_{\mathcal{P}} C_{2u})\ (C_{s1} =_{\mathcal{T}} C_{s2}))$$

$$\implies$$

$$((C'_{11} =_{\mathcal{P}} C'_{21} \implies C'_{12} =_{\mathcal{P}} C'_{22})$$

$$\wedge$$

$$(C'_{11} =_{\mathcal{T}} C'_{12} \implies C'_{21} =_{\mathcal{T}} C'_{22}))$$

Assuming Theorem 2.2, we only need to reason about instructions which violate information flow: `dwncall` and `uplbl`. The key restrictions which provide nonmalleability are those that prevent the pc_l or t_l from becoming compromised and the restriction that compromised data is never downgraded.

```
# PCLBL = TLBL = (TRUSTED, PUBLIC)
# L(key) = L(s0) = (TRUSTED, SECRET)
# L(in0) = (TRUSTED, PUBLIC)
upcall est, ST, ST, enc_end
----------------------------
# PCLBL = (T,S), TLBL = (T,S)
andi in0, in0, MASK
xor s0, key, in0
lw s0, 0(s0)             # (a)
andi s0, in0, mask
lw s0, 0(s0)             # (b)
declreg s0, PUBLIC
upret
----------------------------
enc_end:
```

Figure 2.11: Mitigated AES.

2.8 Program Examples

We now describe examples of how to use our ISA features in practical scenarios.

AES is a well known encryption algorithm which does not require the program to branch on any secrets [34]. Instead, AES uses a public lookup table indexed by computation involving both the secret key and public input. This behavior of executing secret-dependent memory accesses makes it susceptible to a number of timing-channel attacks [17, 59, 104, 52, 89], some of which are similar to the vulnerability in Figure 2.3.

Figure 2.11 is a toy version if this AES-style lookup table access in our ISA. Without mitigation techniques, the execution of the second load *(b)* could be faster if it accesses the same cache line from *(a)*. Similarly, another program may also infer the value of the secret through cache contention.

One existing software-based mitigation technique for preventing this cache timing channel is to preload the entire lookup table ahead of time [21]. Preloading allows a cache implementation to fill its entries with useful data based only on public addresses. However, this approach is not guaranteed to be secure on normal hardware; if a cache were too small to contain the entire table (or evicted entries for any other reason), it is possible that some lookups would trigger misses, thereby leaking information with an unexpectedly *slow* duration for certain keys. Other efforts to eliminate these problems with AES still rely on the assumption that certain instructions are constant-time [64].

Our ISA enables software to control microarchitectural timing channels in a principled manner. On hardware implementing our ISA, the secret-dependent loads in Figure 2.11 cannot affect public microarchitectural state and therefore cannot leak secret information through memory contention. Additionally, the strategy of preloading the cache can still improve performance on some implementations. One potential CPU implementation might maintain private and public cache partitions. During the preload phase, public and trusted code fills up the public cache partition with some or all of the AES table. During the encryption phase, secret code can read those entries but cannot modify them, instead making updates only to the private cache partition. This implementation would allow for a more secure and efficient AES execution. Nevertheless, the duration of the entire execution could leak some information about the secret key; this example also shows how software can use an `upcall` instruction to obscure that duration by providing an explicit end time (via the `est` argument in the example's upcall).

```
# PCLBL = TLBL = (PUBLIC, UNTRUSTED)
# L(guess) = (PUBLIC, UNTRUSTED)
# L(pass) = (SECRET, TRUSTED)
dwncall check_pass
===============================
# PCLBL = TLBL = (PUBLIC, TRUSTED)
check_pass:
  endoreg guess, TRUSTED
  upcall est, ST, ST, end_check
-------------------------------
# PCLBL = TLBL = (SECRET, TRUSTED)
  beq guess, pass, success
  li res 0
  upret
success:
  li res 1
  upret
-------------------------------
end_check:
  declreg res, PUBLIC
  dwnret
```

Figure 2.12: Password checking in the proposed ISA.

2.8.1 Password Checker

In this example, we show how to implement a nonmalleable password checker which can be called by untrusted users with the `dwncall` instruction. The code for this checker is shown in Figure 2.12. This program starts in a public and untrusted context, which would be typical for an unauthenticated user. The untrusted user generates their guess and puts it into the register called `guess`. Then they use the `dwncall` instruction to call the `check_pass` function and gain high integrity. This is analogous to executing a tradition system call where the user program can execute trusted code with operating system privilege.

Once the `check_pass` function has started, it must endorse the user's guess, since a trusted *pc* cannot branch on low-integrity data. In order to compare the secret password value with the guess, the program executes an `upcall` instruction to enter a timing-mitigated region. Inside that region, the program computes either a 1 or 0 based on whether or not the guess was right or wrong, and then returns. Finally, at the end of the `check_pass` function, the result is declassified to public and the call gate exits back to the untrusted context.

If an untrusted user were to execute the `check_pass` function like a normal function call, their attempts to endorse their own guess and `upcall` into a secret and trusted state would both fail. This example illustrates the nonmalleability guarantees and how trusted system code can be resident in the system but only accessible via call gates.

2.9 Related Work

Software Information Flow Control. Software-based IFC has been applied in many settings with the goal of eliminating timing channels [119, 135, 105, 63, 137, 2, 40, 15]. Kashyap et al. [63] discuss various software strategies for enforcing timing-sensitive noninterference. In particular, they focus on using lattice scheduling to ensure that the ordering of visible events does not leak secret information. Parsec [135] is a language for concurrent programming which, given a race-freedom analysis, ensures observational determinism, a noninterference condition for concurrent programs. Bedford et al. [15] have also shown how a hybrid IFC system can provide progress-sensitive noninterference, a weaker condition than timing sensitivity; it does not leak information based on which sets of outputs a program successfully produces. Secure multi-execution, where a program is executed multiple times at varying security levels, has also been used to prove timing-sensitive noninterference [40]. LIO [105] is a Haskell-based language extension for mitigating both external and internal channels through the use of monadic computation and IFC. Of the aforementioned systems, only LIO handles external timing channels. Like our ISA, LIO provides a dynamic semantics for enforcing noninterference but lacks features such as downgrading and integrity tracking.[11] Additionally, it is a high-level language which requires a software runtime for its security, making it unsuitable as an ISA description.

This chapter describes a *dynamic label*[12] model where all data is labeled, including data used to represent labels. This is reminiscent of systems such as JFlow/Jif [85, 83] and the dynamic security labels formalism [141]. These languages rely on static annotations and dynamic label-checking operations to guarantee noninterference while still permitting labels to be used in types. Since our labels have a hardware representation, label-checking operations can be implemented efficiently in hardware. Our platform appears well suited to accelerating such languages, allowing them to execute outside of a software runtime. We considered explicitly modeling precise "labels of labels" as software-accessible state, but it was unclear what further security this provided, and it significantly complicated the run-time checks. Other dynamic IFC languages, such as LIO [105], treat the label of label values as visible to the current context but do not allow for their precise manipulation. This choice mirrors our rule that the label of label arguments must flow to pc_l.

[11]Follow-up work (e.g., [23, 22]) addresses some of these features.

[12]The word dynamic is unfortunately overloaded. Here, it refers to labels whose values are explicitly visible or comparable at run time. Dynamic IFC systems are those that enforce security via run-time checks. Our ISA both has dynamic labels *and* is a dynamic IFC system.

Hardware-level information flow control. IFC techniques have also been used to build timing-safe hardware. While not focused on timing, Suh et al. [110] showed that processors could implement efficient information flow tracking. Caisson and Sapper [71, 70] provided a nested state machine abstraction for circuit design and proved that hardware built using those tools enforced timing-sensitive noninterference. More expressive HDLs that provide similar security guarantees have also been developed using dependent types [45, 44]. The Hyperflow processor [47] is a fully-featured implementation of a RISC-V CPU developed using these techniques.

Secure ISAs. While many of the above HW IFC systems presented CPUs and ISAs, they were focused on security guarantees about the circuits. None of them have proved security results for programs executing on top of their example abstractions. Ge et al. [49] have defined a set of properties they argue post-Spectre ISAs (called aISAs) must enforce to provide efficient, timing-sensitive security. These properties primarily focus on prescribing how an operating system can interact with the hardware to provide timing security. They refer to concrete mechanisms such as hardware partitioning and time multiplexing rather than the security properties that these mechanisms should aim to enforce. Our ISA provides more fundamental guarantees than those suggested in their work, but real implementations of our ISA would likely exhibit many of the properties they list.

Yu et al. [128] have built an ISA extension for "oblivious computing" and have proved probabilistic noninterference results. They have also built and measured the performance of a speculative, out-of-order processor using this ISA and demonstrated its performance improvements over more conservative techniques. Their ISA treats security as an optional component which software may opt-in to by labeling instruction operands as public or secret. This is promising evidence of the practicality of efficient microarchitectures for secure ISAs.

The work of Zhang et al. [137] on language-based timing mitigation defines a software–hardware contract based on "write labels" and "read labels" that almost directly parallel our pc_l and t_l. However, that contract requires well-typed programs that correctly specify write and read labels; the hardware itself is not assumed to enforce any restrictions on how these labels change over time. Furthermore, our ISA considers both confidentiality and integrity while enforcing nonmalleable downgrading. We do not require a fully trusted entity to perform timing mitigation: any `upcall` caller can implement their own mitigation algorithm in their own context.

OS-level information flow control. Asbestos [42] and HiStar [136] are two well known IFC operating systems. They do not assure timing safety. However, HiStar's notion of *gates* informed our call gate mechanism, but the restrictions on gates and the security guarantees differ from ours. NickelOS [102] has been recently developed using *intransitive noninterference*, which allows more flexible security policies than traditional IFC. However, NickelOS is not timing-sensitive and focuses on information flow exposed through OS APIs.

2.10 Conclusion and Future Work

In this chapter, we have proposed an ISA that defines a contract between software and hardware that defines how information may or may not affect the timing of instructions. Importantly, it provides timing safety without prescribing a fixed microarchitecture. As a byproduct, our proofs delineate conditions that hardware should satisfy, thus providing guidance to hardware designers.

We foresee many avenues for further research in the domain of timing secure ISAs. Modeling more ISA features such as exceptions, memory models, and other concurrency mechanisms can provide evidence toward the practicality of this approach to ISA design. Furthermore, it will help expose more potential side-channels that exist throughout the complex environment of multicore processors.

Given this foundation, we can develop new instructions or instruction semantics that expose different timing characteristics, such as fixed-latency scratchpad memory [12] or other "oblivious" computation [128]. Experimenting with these new ideas in the context of a nonmalleable ISA can also ensure that the security guarantees hold end to end.

The largest open question is how to formally verify that hardware implementations satisfy the properties defined in §2.6, allowing us to connect security guarantees of high-level languages and verified operating systems to the actual behavior of the underlying hardware. There are many opportunities to improve tools for relating the behavior of processor implementations to software-visible specifications and security policies, which we begin to address in the following chapters.

CHAPTER 3

A HIGH-LEVEL LANGUAGE FOR PROCESSOR DESIGN

The ISA defined in the previous chapter left some open questions: how can we be sure that the hardware we build actually respsects the timing guarantees necesary for security? How do we know the design implements all of the secure instructions faithfully?

This chapter begins to investigate these questions by addressing the general problem of automatically relating low-level processor implementations to their software-visible behavior.

3.1 Introduction

To achieve high performance, processors parallelize the execution of sequential instruction streams through pipelines, achieving high throughput via microarchitectural optimizations such as bypassing, speculation, and out-of-order execution. Processor designs are inherently complex since they must respect the sequential semantics of the instruction set architecture (ISA) despite aggressively executing operations in parallel. Processors are usually designed using hardware description languages (HDL) that operate at the register transfer level (RTL), providing low-level control but at the cost of highly parallel semantics that make reasoning difficult. This combination of complexity and RTL abstraction makes it difficult to achieve high confidence in the correctness of processor implementations. In practice, RTL processors are usually validated via simulation or bounded model checking: techniques that have seen practical success but cannot expose all bugs in large designs [91, 62].

We propose a new approach, a Pipeline Description Language (PDL) that raises the level of abstraction to specifically target the construction of processor pipelines. PDL allows designers to easily specify the intended functionality of a processor, while still giving them fine-grained control over its microarchitecture and performance. Designers can demarcate stage boundaries, ensuring each stage executes in a single clock cycle. PDL introduces *hazard locks*, which abstract different implementations of stalling and bypass logic to prevent data hazards. Additionally, PDL offers a speculation API that enables pipelines to flexibly initiate and resolve branch prediction. Lastly, PDL supports a limited form of out-of-order execution.

Despite this microarchitectural control, PDL provides an easy-to-understand *one-instruction-at-a-time* semantics. The realized behaviors of pipelines are consistent with an execution that runs each instruction completely in sequence. This strong assurance allows designers and static analysis tools to easily reason about the behavior of a design with respect to a sequential specification, facilitating design space exploration.

As such, PDL does not directly support architectures with relaxed consistency guarantees, such as the memory models of multicore architectures. Nor can PDL express all pipelined architectures, such as superscalar or 2D systolic arrays. Lastly, PDL does not provide strong guarantees about the *timing* of updates to architectural state, and thus cannot reason precisely about timing channels. Supporting more relaxed definitions of correctness, microarchitectural expressivity, and precise reasoning about timing are interesting potential future extensions to PDL.

In this work we present the following:

- An overview of the PDL language and its microarchitectural abstractions for pipeline structure, data hazard resolution, and speculation.
- An informal presentation of PDL's semantics, correctness assurance, and advantages over RTL.
- A description of the PDL compiler implementation.
- Evidence of PDL's expressivity, practicality, and utility in design-space exploration. To do so we evaluate the performance of several RISC-V cores, implemented in PDL with differing microarchitectures.

3.2 Pipeline Description Language

In an RTL implementation, the designer must explicitly instantiate registers to store each pipeline stage's inputs and must manually coordinate the communication between stages. PDL, in contrast, only requires the user to specify the core functionality as an imperative-style program; then they can employ a few key microarchitectural primitives to control the pipeline's structure and performance. The PDL compiler automatically generates the registers and control logic necessary to split the pipeline into multiple, concurrently executing stages. The PDL compiler also makes it safe and easy to alter the pipeline structure or to move functionality across stages, without worrying about introducing bugs.

```
pipe cpu(pc)[rf, imem, dmem] {
  acquire(imem[pc], R); //IFETCH STG
  insn <- imem[pc];
  release(imem[pc]);
  --- //DECODE STG
  op = insn{6:0};
  //decode logic for rs1,rs2,etc.
  acquire(rf[rs1], R); acquire(rf[rs2], R);
  rf1 = rf[rs1]; rf2 = rf[rs2];
  release(rf[rs1]); release(rf[rs2]);
  if (writerd) reserve(rf[rd], W);
  --- //EXEC STG
  alu_out = alu(alu_op, alu_arg1, alu_arg2);
  offset = calc_offset(op, pc, imm, alu_out);
  //start next instruction
  call cpu(pc + offset);
  --- //MEM STG
  acquire(dmem[alu_out]);
  if (isStore(op)) { dmem[alu_out] <- data; }
  if (isLoad(op)) { rddata <- dmem[alu_out]; }
  else { rddata = alu_out; }
  release(dmem[alu_out]);
  --- //WB STG
  if (writerd) {
    block(rf[rd]);
    rf[rd] <- rddata;
    release(rf[rd]);
  }
}
```

Figure 3.1: Abbreviated PDL code for a 5-stage RISC pipeline.

3.2.1 Language Design

Figure 3.1 demonstrates some features of PDL by presenting an abbreviated RISC processor. The code in this example is more similar to an imperative program than typical RTL code, and can mostly be understood via the straightforward imperative interpretation. Syntax for combinational logic in PDL is mostly standard, with support for sized integers and typical operators such as bit selection and concatenation. Variables are declared and assigned exactly once, like Verilog [111] wires. Array access notation denotes a request to a *memory*, which is any stateful, addressed data structure, including registers; it need not be implemented as SRAM/DRAM. Memories may be declared as providing either combinational or synchronous read access[1]. The read at line 3 is made with the arrow (<–) notation because ”imem” has a synchronous interface: its data cannot be used until the next pipeline stage. Modern processors often contain pipelined subcomponents: PDL thus supports a `call` statement that allows one pipeline to make a synchronous request to another, e.g.:

```
int<32> divres <- call multi_cycle_div(arg1,arg2);
```

The above example sends a request to a pipelined divider, `multi_cycle_div`. The subsequent stage in the primary pipeline will wait for the divider's response before executing.

Pipeline Structure. Stage separators (–––)[2] control the structure of a pipeline, breaking up combinational logic across multiple clock cycles.

Although each instruction flows through pipeline stages in sequence, stages actually execute in parallel and can process multiple instructions at a time. For the most part, separators can be placed wherever the designer wishes, to tune the critical path of the realized design without affecting the functionality; PDL rejects any design that could violate one-instruction-at-a-time semantics.

Out-of-Order Stages. PDL is not limited to fully in-order pipeline descriptions; placing stage separators inside conditional branches describes a pipeline as a directed acyclic graph. While instructions travel through the *ordered*

[1] Synchronous means that requests and responses are coordinated via a clock-edge aligned protocol, such as a ready-valid interface.

[2] This notation is inspired by Dahlia's [86] ordered composition operator.

Table 3.1: The hazard-lock interface with summarized requirements for using each operation.

Operation	Description	Requirements
`reserve(m[a], R/W)`	Defines the intended order of memory ops	Lock has not been acquired and is executed during an in-order stage.
`block(m[a])`	Stall the current thread until it can execute the associated op	Lock has been "reserved".
`acquire(m[a], R/W)`	Syntactic sugar for `reserve; block`	Same as reserve.
`read/write(m[a])`	Execute the op, potentially forwarding data	Lock is "acquired" via block().
`release(m[a])`	Release lock resources associated with op and commit	Read or write has executed, and release is executed during an in-order stage.
`checkpoint(m)`	Create a checkpoint of lock state	Automatically inserted with final reservation.
`rollback(c)`	Reset lock state to checkpoint c	Automatically inserted with verify statements.

stages of the pipeline in the same order they were started, this is not true for *unordered* stages. Consider the following CPU design, which utilizes the aforementioned **`call`** statement to execute division in a separate pipeline, but still allows memory access operations to execute in parallel:

```
//DISPATCH
if (isDiv) {
  --- //DIV
  int<32> res <- call multi_cycle_div(arg1,arg2);
} else {
  int<32> addr = arg1 + off;
  --- //DMEM
  int<32> res <- dmem[addr];
}
--- //WB
rf[rd] <- res;
```

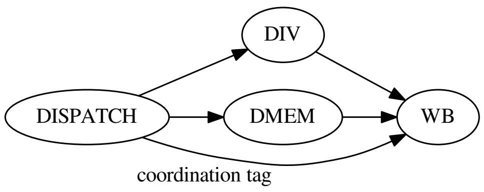


Figure 3.2: Stage graph of a pipeline with unordered DIV and DMEM stages. Each stage can execute different instructions in parallel: an in-order issue, out-of-order execute pipeline. The coordination tag is used to re-establish the original execution order in the WB stage.

Figure 3.2 visualizes the pipeline generated by this code snippet. The DIV and DMEM stages may execute in parallel, despite being unordered. PDL ensures that the code following the branch (the WB stage) *does* execute in order. PDL generates coordination signals that record the original execution order. The DISPATCH stage enqueues a tag indicating which branch an instruction took; the WB stage uses this queue to determine from which stage to receive its next inputs, and stalls until that stage has completed execution.

Pipeline Threads. In PDL, a pipeline body describes how to sequentially process a *single* instruction. To initiate execution of the next instruction, a *recursive* **`call`** is used, as at line 16. The one-instruction-at-a-time semantics allows the designer to think of these recursive calls as tail calls that occur at the end of the pipeline body. Semantically, a pipeline is a loop that processes one instruction per iteration.

The placement of the recursive **`call`** does not affect the semantics of the generated circuit, but it does have an impact on performance: it introduces concurrency. At this point in the pipeline, the pipeline begins processing the subsequent (called) instruction, *in parallel* with the rest of the current instruction. We borrow the term *thread* from concurrent software; each instruction is executed by a single thread that travels through the pipeline independently, and potentially in parallel with other threads. *Thread order* refers to the order in which threads are initiated; it is equivalent to program order in processors.

3.2.2 Preventing Data Hazards with Hazard Locks

A key to one-instruction-at-a-time semantics is ensuring that pipelines are free of data hazards. Data hazards occur when read and write operations on memories do not respect thread order, and are typically prevented by explicit stall logic or by bypassing values from writes to reads. For instance, in a standard 5-stage processor pipeline, stall and bypass logic are needed to prevent the following RISC-V [121] instruction sequence from creating a read-after-write hazard:

```
lw a0, 0(sp) //load data from stack into a0
addi a0, a0, 1 //increment a0 register
```

Absent this logic, the read from register ”a0” in the second instruction would occur *before* the load into ”a0”, so the final value of ”a0” would effectively ignore the load instruction.

Hazard Locks. PDL introduces a novel *hazard lock* abstraction to prevent data hazards. Hazard locks encapsulate data hazard prevention in a separate hardware module, whose usage in the pipeline can be checked by the compiler. This design contrasts strongly with traditional RTL development, where bypass and stall logic is explicitly described by the designer and manually integrated into the entire pipeline. RTL hazard resolution logic is also *non-modular* and brittle; it cannot be re-used across designs, and must often be changed if the pipeline structure is modified. On the other hand, hazard locks are a general abstraction that can express a variety of different microarchitectural designs, from simple stall logic to the renaming used in complex out-of-order pipelines.

As with traditional software locks, a thread must *acquire* a hazard lock before accessing the associated memory location:

```
acquire(rf[rs1], R); //acquire READ lock for rs1
int<32> x = rf[rs1]; //OK: lock acquired
int<32> y = rf[rs2]; //ERROR: acquire missing
```

Similarly, hazard locks must eventually be *released*. To support implementations with a variety of performance characteristics, PDL allows acquisition to be split into two phases: *reservation*, and *blocking* until the reservation is fulfilled.

```
reserve(rf[rd], W); //reserve WRITE lock for rd
---             //(READ locks can be reserved too)
```

Table 3.2: Speculation operations supported by PDL.

Operation	**Description**
`s <- spec call pipe(pred)`	Spawn a speculative thread using value `pred`.
`update(s, npred)`	Update the prediction for speculation `s` using `npred`.
`verify(s, real) { pred(...) }`	Mark speculative thread `s` as correctly or mispredicted by comparing the value of `real` to the original prediction; optionally, update external predictor module `pred`.
`spec_check()`	Kill the current thread if it has been mispredicted.
`spec_barrier()`	Stall until this thread's status is known. If mispredicted, then kill this thread.

```
block(rf[rd]); //STALL this stage until OK to exec
```

The **`acquire`** operation is actually just syntactic sugar for the sequence **`reserve`** followed by **`block`** in the same stage.

A key insight is that, even in highly speculative, out-of-order processors, there is an in-order execution point where the CPU establishes and records sequential data dependencies in some data structure. We abstract this record-keeping point as lock *reservation*; it must execute in thread order, but still allows execution to proceed freely. *Blocking* represents the point in the pipeline when a stage may be forced to stall lest it observe a stale value or incorrectly overwrite state. *Writing* data makes it available for bypassing, and *releasing* the lock represents the actual, in-order commit point.

Table 3.1 lists a summary of the hazard lock interface and how the PDL compiler restricts its use. For brevity, we often refer to hazard locks as "locks" in the remainder of the chapter.

3.2.3 Refining the Hazard Lock Abstraction

While the lock abstraction allows PDL to reason about whether a design is free of data hazards, different lock implementations have different performance characteristics. PDL is bundled with a small library of lock implementations reflecting different microarchitectural designs; designers can also implement and use their own unique locks in RTL. Here we present the implementations we have developed.

Queue Lock. The simplest lock implementation is a First-In-First-Out (FIFO) queue of reservation requests for a given memory location. *Reserve* enqueues a request. *Block* stalls until the associated reservation is at the head of the queue. *Read* and *write* access memory normally. *Release* dequeues the reservation. The implementation refines the specification required by PDL, but assumes we have a separate queue for each memory location: an obvious impracticality for large memory. To efficiently implement queue locks, we provide a fully associative array of queues. In this way, any location can be associated with any queue and is disassociated once the queue is completely empty (and is therefore reusable by another location). The size of the associative array and the depth of the queues are design parameters that may influence performance; for instance, attempting to reserve an unused location when all queues are in use could cause pipeline stalls. This lock represents a simple but low-performance design: it has stall logic but no bypassing paths between conflicting writes and reads.

Bypass Queue. To support in-order cores with bypassing, we implemented a lock which commits writes to the memory in reservation order, but allows write values to be bypassed to reads by storing them in a temporary buffer. We implement this lock as a queue of write addresses, values, and valid bits. *Reserve write* enqueues the address and sets the associated valid bit to 0. *Block write* is a no-op, and writes update the data and valid bits of the associated queue entry. *Release* then commits the write to the actual memory.

Reserve read checks for conflicting writes and updates a register with the entry number of the given write. *Block read* stalls until the conflicting write has executed (if there is one), and reading either forwards data from the write or reads directly from the memory. *Release read* frees internal state for future read reservations. This implementation also buffers read data so that access to the memory occurs in the same cycle as reservation, and includes combinational bypass paths so that writes are observable to reads in the same cycle. With this implementation, we can fully bypass a standard 5-stage in-order core.

Renaming Register File. We also implemented the lock interface with a renaming register file of the kind used in modern out-of-order processors. A renaming register file maintains a table that maps architectural register addresses (a.k.a. names) to physical names, and stores data in a traditional register file indexed by physical names. Lock *reservation* translates to physical name allocation for writes, and physical name lookup for reads. A vector of per-register valid bits tracks their status: they are set to 0 on allocation, and 1 once data is written. *Block* operations are no-ops for writes and check the appropriate valid bit for reads. *Release* operations are no-ops for reads, but for writes they add the old name mapping to a free list for future allocation. Like the Bypass Queue, this implementation can fully bypass a 5-stage pipeline, although it is also general enough to be a good fit for a Tomasulo-style out-of-order machine [114].

3.2.4 Speculation

Speculation is critical for processor performance, and PDL enables a large class of speculation through *speculative call* statements. As with locks, PDL offers a modular abstraction for speculative operations, summarized in Table 3.2. Designers can initiate a speculative thread, mark it as (mis)predicted, update a prediction, and kill speculative threads. In PDL, all speculation is made explicit, even speculation that is often overlooked in processors: the typical ”pc + 4” prediction that instructions usually execute sequentially. The following snippet implements this speculation:

```
spec_check(); //Kill this thread if misspeculated
s <- spec call cpu(pc + 4); //Spawn a new thread
```

The **spec call** speculatively spawns a new thread with the argument `pc + 4` and produces a handle, `s`, used to later reference this speculation. We refer to the thread making the speculative call as the *parent* thread, and the thread created by the speculative call as the *child* thread. In a pipeline that uses speculation, every thread has the potential to be both a parent *and* a child. For that reason, we use the operation **spec_check** to kill the current thread if it is misspeculated. Note that this check does not prevent “nested” speculation (i.e., speculation initiated by an already speculative thread); this check just ensures that *already misspeculated* threads do not continue to speculate.

Eventually, the parent thread must *verify* whether its prediction was correct:

```
s <- spec call cpu(pc + 4);
...             //later in the pipeline
spec_barrier(); //blocking version of spec_check()
verify(s, npc); //check that npc == pc + 4
```

The parent thread first ensures that it itself is non-speculative with a blocking version of the speculation check. Then it marks reference `s` with a single bit defining its correctness; PDL automatically propagates the original prediction and inserts a comparison with the given value. In this instance the **`verify`** operation marks `s` as correct if `npc == pc + 4`. If the prediction was *wrong*, the child thread will be killed once it executes a **`spec_check`** or **`spec_barrier`** operation (often in the same cycle). In this case, **`verify`** also causes the parent to spawn a new, non-speculative, thread with the correct value.

PDL also supports an **`update`** operation that can be used to compose both termination and speculation, by spawning a new thread if the new (presumably more accurate) prediction does not match the original and marking the old child thread for termination.

PDL allows predictors to be implemented as modules in RTL safely: predicted values cannot affect functional correctness! Predictor accuracy has significant impact on processor performance, so the ability to integrate custom predictors without compromising PDL's correctness assurance is critical for efficiency. The following example shows how to use an external branch history table (BHT) for branch prediction:

```
s <- spec call cpu(pc + (bht.req(pc) ? imm : 4));
...
verify(s, npc) { bht.upd(pc, brTaken) }
```

The module `bht` must be declared earlier in the PDL program as externally implemented with a `req` interface that produces a boolean. A true value indicates the branch should be taken; false means it should not. The predictor `bht` also provides an `upd` interface that receives a pc and a "was taken" bit to update its own internal state. Whenever **`verify`** executes, the branch history table is also updated.

Implementation. Like locks, speculation in PDL places few restrictions on the structure and timing of the pipeline. To achieve this, PDL stores speculation state in a table, which threads can use to update and read speculative status. Spawning a thread allocates a new identifier. **`verify`** and **`update`** statements mark a child thread's entry as correct or incorrect. Entries are freed whenever a child thread learns its speculative status through **`spec_check`** and **`spec_barrier`** statements. Importantly, whenever a **`verify`** or **`update`** marks an entry as mispredicted, it also marks *all newer entries* as mispredicted too. In this way, all threads will eventually be notified of their status, even if their parent is killed before it calls **`verify`**.

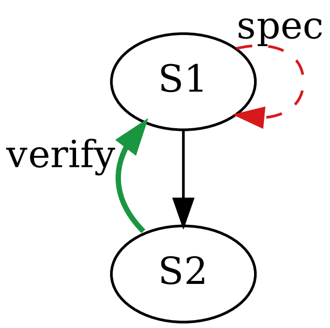
This table is a straightforward circular buffer, which is also synthesized with combinational bypass paths between the status updates and speculation checks. These paths are necessary when pipelines both speculate and resolve every cycle (the typical case). However, representing this structure as a registered table allows it to function even in loosely timed scenarios, where threads may not check their speculative status in *every* pipeline stage. This implementation requires no strict assumptions about the timing of speculative checks and verification operations, and frees PDL from the in-order requirements of other tools that generate pipeline speculation [88]. This design can contribute to the overhead of PDL when building very simple processors whose control logic can be manually optimized by reasoning about global invariants. Nevertheless, it generalizes to more complex processors which do not broadcast speculation results to every stage.

```
pipe ex1(in)[m]: {
  spec_barrier();
  s <- spec call ex1(in + 1);
  reserve(m[in], R);
  acquire(m[in], W);
  m[in] <- in; release(m[in], W
     );
  ---
  block(m[in], R);
  a1 = m[in]; release(m[in], R)
     ;
  verify(s, a1);
}
```

(a) Original PDL Code

```
pipe ex1(in)[m]: {
  a1 = m[in];
  m[in] = in;
  call ex1(a1);
  //Erased ---
  //Erased Spec Operations
  //Erased Lock Operations
  //Delayed Write Operation
  //Replaced verify() with
     tail call
}
```

(b) Code representing sequential semantics



(c) Stage Graph Representing the Pipelined Circuit

Figure 3.3: Interpretations of a Sample PDL Pipeline

3.2.5 Supporting Speculative Reservation

For any given lock implementation, allowing speculative lock reservation could lead to bugs; a thread holding a lock may be killed, leading to an inconsistent lock state. To increase the expressivity of PDL, we extend the lock abstraction with **`checkpoint`** and **`rollback`** primitives that can be used to safely undo speculative lock operations. A **`checkpoint`** causes the lock to logically snapshot its current internal state and returns a handle referencing this snapshot; **`rollback`** indicates that the snapshot is no longer needed, and/or that the lock should revert its internal state to the given snapshot.

Unlike the other lock operations, these need not be exposed to the designer; these operations *must* be executed exactly at certain points in the pipeline, and the compiler can automatically insert them. Specifically, a checkpoint must be taken atomically as each thread completes its reservations; thereby making a checkpoint *between* the reservations of a parent thread and its (speculative) child. Rollback invocations must coincide with **verify** and **update** operations; whenever a parent terminates its child thread, it should revert all memories to the state that captures the parent's reservations, but not subsequent speculative ones. To illustrate this, we annotate the following snippet where the compiler would invoke **checkpoint** and **rollback** operations:

```
s <- spec call cpu(pc + 4); //...
if (writerd) { reserve(rf[rd],W); }
//c <- checkpoint(rf);
//take checkpoint after last reservation
...
verify(s, npc); //rollback(rf, c);
//undo speculative ops on rf if misprediction
//release lock state associated with checkpoint
```

Checkpoint Implementations. We extend both the BypassQueue and the Renaming Register File with standard rollback mechanisms. The former requires little additional state; the head of the write queue (i.e., most recently reserved write) itself serves as a checkpoint. *Rollback* simply requires invalidating all entries newer than that point and moving back the write queue head. For the rename file, we replicate the mapping table and free list; *rollback* resets the main mapping table and free list to the indicated replica.

3.3 Sequential Pipeline Behavior

A central draw of PDL is that it allows designers and static analysis tools to describe and reason about pipelines as *sequential programs* that process instructions one at a time. Any PDL program accepted by the compiler can be automatically translated—via a straightforward erasure process—to a sequential program that serves as a specification of correctness, which the pipelined circuit generated by PDL refines.

RTL languages have no such canonical sequentialization; many hardware designs do not implement a sequential specification and thus may exploit the unfettered parallelism of RTL. However, PDL's one-instruction-at-a-time semantics greatly simplify reasoning and can likely alleviate the scalability problems of traditional RTL testing and verification. For instance, to achieve soundness, bounded model checking of RTL pipelines requires considering instruction sequences long enough to saturate the pipeline [91]; applying this technique to PDL programs would only require analyzing sequences of length 1. Sequential software proof techniques, such as Hoare logic [55], which are not easily adapted to RTL languages, can also be applied to sequential PDL programs.

Assumptions. PDL's correctness relies on the correctness of the compiler itself, and the RTL implementations of the lock API. Namely, locks must ensure that reads and writes must be stalled (via `block`) if they would produce observations inconsistent with the reservation order. We plan to formalize this verification requirement, as well as the correctness of PDL's overall design, in future work.

Locks can be verified using existing hardware verification techniques [19, 62]. However, unlike verifying bypassing networks in RTL processors, locks enable modular verification: they can be verified in isolation, since PDL checks that they are used correctly by the main pipeline. Modularity also implies that locks may be reused across processor designs, amortizing the effort of correctness proofs. Importantly, a verifier (human or tool) only needs to reason about the software-visible architectural state, and *does not* need to supply any global invariants about a pipeline's *microarchitectural* state, a notoriously difficult verification task [28, 140].

3.3.1 Extracting a Sequential Specification

A PDL pipeline can be understood as a sequential program through a straightforward translation procedure. This program effectively defines the behavior of the given PDL pipeline as a sequence of updates to, and observations of, architectural state. As an example, Figure 3.3 includes a simple pipeline, its sequential interpretation, and a graph describing the circuit structure. The steps for this translation are straightforward:

- Erase stage separators, speculation checks, initiation and invalidation, and lock operations.
- Replace `verify` statements with `call` statements
- Delay memory write and recursive `call` statements to the end of the pipeline.

Erasing microarchitecture-controlling primitives is intuitive; by design they should have no impact on the intended functional behavior of the pipeline. Verifying speculation is the exception, as it *does* imply functionality; however, without any speculative events it reduces to unconditionally spawning a child thread (i.e., a recursive **`call`**).

We also apply reordering transformations on memory writes and recursive **`call`**s. Memory writes are delayed until after all reads, and recursive **`call`** statements are moved to the end of the program to become tail calls. In the realized pipeline, the placement of these statements have performance impact (and often placing them earlier is better); functionally, their location in the program should have no impact. For **`call`** statements, this property is obvious; since **`call`** initiates the next instruction, its behavior should be sequenced after all of the operations for the current thread. For memory writes, we made a simplifying design decision to declare that their effects are not visible to the current thread. This decision simplifies locks so that they do not need to consider dependencies between reads and writes from the same thread, as no thread may read its own writes. Conveniently, this transformation also produces programs that align with typical ISA semantics; when a location is both read and written by an instruction, the read appears to occur *before* the write.

The obvious operational interpretation of these sequential specifications (such as Figure 3.3(b)), yields a definition of correctness for the generated circuit; the effects on memories referenced by the pipeline appear to happen one iteration at a time, in sequence. Thus, each iteration corresponds to a single instruction that may read and write shared memory, and lastly determines which instruction to execute next.

3.3.2 Informal Correctness of PDL

We briefly justify why the PDL compiler only generates pipelines whose concurrent execution is consistent with the behavior of their sequential interpretation.

Preventing Data Hazards. In PDL, locks do the "heavy lifting" to prevent data hazards. As explained in §3.2.3, locks implement stall, bypass, and commit logic for each memory that the pipeline accesses, and expose this logic through the lock interface in Table 3.1. PDL confirms that this interface is used appropriately, and rejects pipelines in which data hazards could still occur. Specifically, lock implementations need to assume that:

- *Reservations* are made in the intended program order.
- Stages check that *block* returns true before accessing memory.
- *Write locks* are released (committed) in program order.

The PDL compiler rejects any pipeline description in which these three requirements may not be satisfied. To enforce them, it checks that each lock is used in the intended sequence (i.e., *reserve*; *block*; *access*; *release*), and that the *reserve* and *release* operations are guaranteed to execute in thread order. The former is easily checked with a path-sensitive analysis (see §3.4.3). The latter requirement necessitates reasoning about the possible parallelism in the compiled design. PDL does not reason about concrete timing of stage execution. Instead, by examining the structure of the stage graph, it proves that all threads must traverse, in thread order, the stages that contain *reserve* and *release write* statements for a given memory. For example, if all *reserve* statements for a given memory only occur in a single stage, in-order reservation is trivially satisfied. §3.4 discusses how the PDL compiler implements these checks in more detail.

Speculation Correctness. We also argue that speculation does not influence the observations of threads in PDL pipelines. PDL guarantees this by validating the following conditions:

- All speculative calls are verified or killed accurately.
- Misspeculated threads are rolled back before committing writes

PDL establishes a key invariant that simplifies correctness reasoning around speculation: all speculative calls are resolved *in thread order*. PDL enforces this by restricting `verify` statements to non-speculative threads. If speculation were resolved out of order, then a *verified* thread might still be speculative and PDL would need a more complicated speculation tracking and resolution mechanism. Instead, PDL guarantees that each `verify` statement fully determines the speculative status of its child thread: either it was correctly predicted, or it *and its children* were all misspeculated. Through another path-sensitive analysis, the compiler ensures that all speculation is eventually resolved by the parent.

Intuitively, PDL checks that reservations to write locks (i.e., those that can change the observations of other threads) are rolled back before they can influence non-speculative threads, if they were misspeculated. Since speculation verification happens in order, the rollback event associated with that verification resets all speculatively updated locks to a point *right after the parent's reservations*. Effectively, in addition to terminating all speculative threads, verification signals for locks to undo all speculative modifications to their state. We also require that write locks are not released by speculative threads; this prevents writes from becoming permanent before misspeculation is discovered.

3.4 Rule Checking

This section expands on the details of the PDL compiler's program checking process which defends the intuition outlined in §3.3. Figure 3.4 visualizes the end-to-end checking and compilation process.

3.4.1 Lock Checking

In addition to standard type checking rules, PDL has a unique set of restrictions for locks to ensure that the realized, parallel design accesses locks correctly.

- Locks must be reserved *in thread order*.
- Each thread must use locks in the appropriate sequence: reserve; block; read/write; release.
- Write locks must be released non-speculatively *in thread order*.

The first and third restrictions are checked by proving that all of the reserve and release operations occur during in-order stages of the pipeline. We establish in-orderness by constructing the stage graph for the pipeline; if a stage is ordered with respect to *all* other stages, then it will be executed by threads *in order*. We slightly relax this restriction, allowing these operations for a given

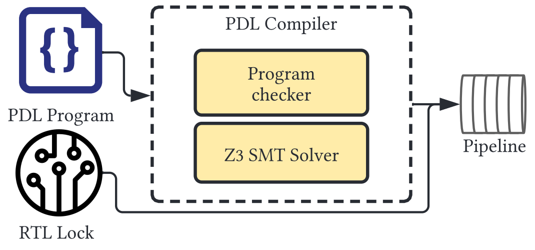


Figure 3.4: Each PDL program is checked for well-formedness and correct lock usage, relying on Z3 for its path-dependent analyses. If successful, it produces a BSV pipeline. Locks are implemented in RTL and not checked for correctness.

memory to occur inside *at most one branch* of an out-of-order region. For instance, in Figure 3.2 it is safe if all reservations for access to data memory occur in the `DMEM` stage. Although it is unordered with respect to the `DIV` stage, `DIV` does not make any reservations and thus races to reserve locks cannot occur.

Reservations must also happen *atomically*, meaning that thread i makes all of its reservations for a given memory before thread $i + 1$ makes any. The PDL compiler ensures atomicity by annotating the start and end of a *lock region*: the set of stages in which the reservations for a given memory occur [3].

```
reserve(m[a], R); //Start Lock Region m
---
reserve(m[b], W); //End Lock Region m
```

The compiler inserts control logic to ensure that only a single thread may execute inside a lock region at a time. However, in practice the region is usually only a single stage and no extra logic is needed or used. The main use of multi-stage reservation is for indirect references:

[3] The designer may also manually place these annotations and compiler will check that they actually wrap all of the reservation operations for the given memory.

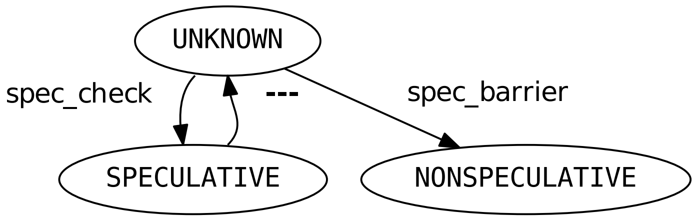


Figure 3.5: A state machine representation of the typestate used to check speculative status of pipeline threads.

```
acquire(m[a], R); //Start Lock Region m
b <- m[a];
---
acquire(m[b], W); //End Lock Region m
```

This pipeline cannot reserve all aliasable memory locations in a single stage, because it must read from `m` before knowing all the addresses to reserve. While this pattern can arise in certain pipelined circuits, it is uncommon for processors. Importantly, *atomicity* is only required for all of the reservations of a given memory; reservations for two different memories may occur in different stages without synchronization penalty, since those reservations cannot possibly alias each other.

3.4.2 Speculation Checking

PDL limits the use of speculation to ensure that the final design is equivalent to a non-speculative version. First, we check that all speculative calls are eventually verified across any program path; this uses the same machinery as checking that lock operations are called in the correct sequence and is described in more detail in §3.4.3. Second, PDL prevents speculative effects from being observable by non-speculative threads via restricting the set of operations that speculative threads may execute. We adapt typestate [108] to determine the speculative status of threads in any given stage. Threads can transition between three states: `Unknown`, `Speculative`, and `Nonspeculative`[4], beginning in state `Unknown` and using the speculation primitives to transition to other states. Figure 3.5 illustrates the relationship between typestates and these primitives.

The non-blocking check **`spec_check`** transitions to `Speculative`, only establishing that the thread is *not definitely misspeculated*. After a stage separator, if `Speculative`, the typestate is reset to `Unknown` since its status may have been resolved by the time the thread executes the next stage. The only way to establish that a thread is `Nonspeculative` is to use a blocking check **`spec_barrier`** . `Unknown` threads may not make speculative calls or reserve locks, as these operations (if made by an already misspeculated thread) could cause races on starting new threads, and inconsistent lock state, respectively. Neither `Unknown` nor `Speculative` threads may **`verify`** their own speculation or release write locks, lest they permanently update lock state and write data that some non-speculative thread may read before they are rolled back.

3.4.3 Path-Sensitive Checking

PDL cannot rely on purely syntactic type checking to prove that locks transition through the correct sequence of states. Since the placement of operations and stage separators inside conditional branches can influence the structure of

[4]We do not need a `Misspeculated` state since misspeculated threads will automatically be terminated and will not execute code following the speculation check operations.

the pipeline, it is important to allow flexible placement of lock operations. For example, a purely syntactic type checker could not prove the following code snippet reserves the lock before blocking on it:

```
if (writerd) { reserve(rf[rd], W); }
---
if (writerd) {
  block(rf[rd]); rf[rd] <- wdata; release(rf[rd]);
}
```

To permit such programs, we generate constraints to prove that locks are in the necessary state when they are used. The compiler runs an abstract interpretation over the program that approximates branch conditions using variable equality and boolean logic. To allow the compiler to precisely check lock usage, designers need only to simplify branch conditions into booleans or comparisons between variables. We then employ the Z3 SMT solver [36] to verify that the constraints are satisfied. The same code snippet follows, annotated with the information derived by our compiler and checked by Z3:

```
if (writerd) { //LockState: free
  reserve(rf[rd], W);
}               //Lockstate: writerd => reserved
                //           ^ !writerd => free
---
if (writerd) {
  block(rf[rd]); rf[rd] <- wdata; release(rf[rd]);
}              //Lockstate:  writerd => released
               //           ^ !writerd => free
```

PDL also uses this technique to confirm that speculative calls are resolved and that pipeline **`call`** statements are well-formed: each thread *either* makes a single recursive call, *or* outputs a value. In other words, each thread may either spawn a single child thread or terminate.

3.5 Implementation

Our prototype implementation of the PDL compiler is written in 10K lines of Scala; the lock implementations are written in 1.7K lines of Bluespec System Verilog (BSV) [87] and 1K lines of Verilog[5]. Given a PDL source program, the PDL compiler produces a BSV module that implements the specified design. From this module, the open-source BSV simulator and compiler can be used to run simulations or produce synthesizable Verilog. We chose BSV as a target language since it provides a natural translation from PDL stages to BSV rules. While this choice simplified code generation, it is not a fundamental requirement; it is certainly possible to implement a different back end for the PDL compiler targeting Verilog or another similar HDL. We implement locks in a combination of BSV and Verilog; the Queue Lock is implemented in BSV and the others are written in Verilog. The language choice at this level is for convenience; in principle, locks can be implemented in any RTL language.

3.5.1 Code Generation

In BSV, each rule is guaranteed to execute atomically in a single clock cycle; additionally, one can provide conditions that prevent BSV rules from executing. Given these conditional rules, BSV will automatically generate the control

[5] The PDL compiler is open-source and can be found at: `https://github.com/apl-cornell/PDL`.

Table 3.3: Performance in Cycles-Per-Instruction of multiple processor configurations on a selection of integer benchmark kernels. All processors implement the RV32I ISA, except for the PDL 5Stg RV32IM configuration.

Processor	coremark	aes	gemm	mm-block	ellpack	kmp	nw	queue	radix	GeoMean
Sodor	1.441	1.201	1.530	1.525	1.380	1.496	1.355	1.332	1.282	1.37
PDL 5Stg	1.436	1.230	1.529	1.525	1.380	1.496	1.376	1.332	1.282	1.39
PDL 3Stg	1.205	1.101	1.265	1.262	1.190	1.247	1.188	1.118	1.108	1.18
PDL 5Stg BHT	1.367	1.154	1.413	1.414	1.269	1.255	1.306	1.231	1.202	1.28
PDL RV32IM	1.384	1.230	1.421	1.226	1.280	1.496	1.376	1.332	1.282	1.32

logic necessary to execute as many each cycle as possible. The PDL compiler's strategy is to represent each pipeline stage as a single BSV rule, and to supply conditions to *stall* or *kill* a stage's execution when necessary, according to the appropriate PDL primitives.

We split the original PDL program into a DAG of pipeline stages as described in §3.2.1. Live variable analysis is used to annotate the edges between the stages with all variables needed by a later stage. Each stage translates to a single BSV rule, guaranteeing that all of its state-modifying operations occur in the same cycle. Each edge translates to a FIFO which stores the data communicated between stages. The FIFO is an abstraction over pipeline registers; the current compiler uses the default BSV FIFO implementation (which employs 2 registers), but it could be replaced with a single-register implementation. The only exception to this generation mechanism are the coordination edges generated to control out-of-order regions of the graph; these send only a single value, used by the downstream stage to determine which other input FIFOs to read from.

The combinational logic associated with each stage can be generated in a straightforward fashion and placed outside of the rules. The rule bodies contain all of the state-modifying or inter-stage operations: FIFO en/dequeues, lock operations, memory writes (and reads for synchronous memories), and updating speculation status. Lastly, we generate PDL's stall conditions to prevent BSV from scheduling rules erroneously. BSV automatically ensures rules do not execute when there is no valid data or there is back pressure from a later stage (i.e., the input FIFO is empty or the downstream one is full). Thus we only need to add stall conditions for **`spec_check`**, **`spec_barrier`**, and **`block`** commands and for stages that receive data from a variable-latency operation (e.g., responses from synchronous memories or other PDL pipelines).

BSV automatically generates a schedule that executes as many rules as possible within a single clock cycle; this corresponds to the control logic for stage activation. PDL does not guarantee that this schedule is optimal. PDL does automatically include two scheduling directives necessary for high performance. One indicates to BSV that it is safe to execute all stages that send data to the beginning of the pipeline (i.e., recursive **`call`** and **`verify`** statements) and appropriately muxes the correct values based on misprediction logic. The second ensures that BSV will make speculation results combinationally available to earlier stages (i.e., the misprediction signal is propagated immediately to early stages that contain **`spec_check`** or **`spec_barrier`** statements). With these directives, PDL can generate speculative pipelines that execute one instruction per cycle.

3.6 Evaluation

To demonstrate PDL's expressivity and efficiency, we present a number of different implementations of the RISC-V32 ISA, and compare their performance and area to a baseline implemented in Chisel [10]. To measure software-visible performance (cycles per instruction—CPI), we use RTL simulations for the designs that simulate cache hits for every memory access (single-cycle responses). To measure processor area, we target a 100 MHz clock frequency using 45nm FreePDK [107] technology and execute synthesis and place-and-route. In our measurements we consider only the processor cores and exclude caches and any other parts of the memory hierarchy since PDL was not used to generate that part of the microarchitecture.

3.6.1 Performance

We compare the PDL-designed processors with the Sodor processor in its fully bypassed configuration. Sodor is implemented in Chisel and represents a standard 5 stage RV32I processor [92]. First, we show that our processors can implement a similar architecture. The PDL 5 Stage processor divides functionality across stages in the same way as Sodor and uses the Bypass Queue lock (see §3.2.3) to bypass write data. The PDL processor also implements the same speculation logic as Sodor, always predicting that branches are not taken and suffering a 2-cycle stall on taken branches and jumps. Both processors also experience the same stalls for data hazards thanks to their bypassed designs; they stall for 1 cycle, but only on load–use dependencies[6].

[6] A load instruction whose value is used by the subsequent instruction.

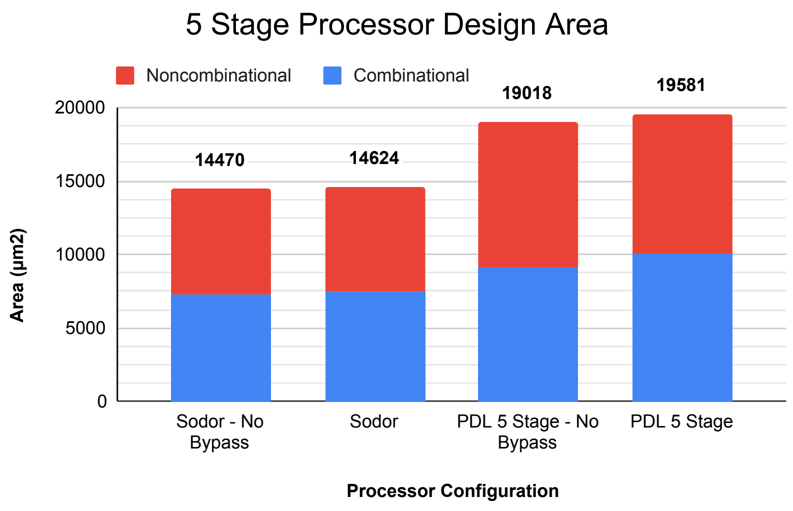


Figure 3.6: Design cell area for 5 Stage processors both with and without bypassing logic. Sodor is a baseline implemented in Chisel. Results achieved using 45nm technology targeting a 100 MHz clock frequency.

The first two rows of Table 3.3 present the CPI of the designs when executing a number of small benchmarks. The first benchmark, coremark, is from the eembc [33] embedded benchmark suite. The remainder of the benchmarks are the selection of integer kernels from the MachSuite [90] that we could successfully execute on the Sodor processor. There is a small variance between Sodor and PDL 5Stg, especially in the `aes` benchmark; this is the result of a minor difference in the benchmark binaries, which was required to be compatible with the test benches. We manually confirmed that exactly the same stalls occurred in both pipelines and that this CPI difference does not signal a difference in processor performance.

Figure 3.6 shows the synthesis results for the Sodor and PDL 5 stage processors. We also include the areas for non-bypassed versions of both processors to measure the overhead caused by including bypassing logic. The PDL processor requires more area than the Sodor processor, and bypassing induces a larger percentage overhead than in Sodor (2.96% increase vs. 1.06%). Some of this area is due to less efficient stall logic and pipeline register representation. Specifically, the FIFO implementations consume significant combinational and non-combinational area. These overheads are all artifacts of the immaturity of the PDL compiler, and are not fundamental to PDL's language design. Bypassing, on the other hand, is more expensive in PDL than in the hand-written version, because PDL assumes nothing about the timing or coordination of stages and pays for this generality. In particular, the Bypass Queue requires a dynamic priority calculation to determine which write is the most recent, and stores some information redundant with data in pipeline registers.

Nevertheless, we are not concerned with this overhead for two reasons. First, these cores are small and thus cache areas would likely dominate costs in a complete chip. We used CACTI [124] to estimate the area overhead when using even tiny (4KB, 2-way associative) L1 data and instruction caches. For this configuration, PDL induces only a 5% overhead when considering the total area of core and L1 caches together; this provides an upper bound on chip area overhead, as real systems often use significantly larger caches. Second, in designs that support any out-of-order execution (even mostly in-order CPUs such as Ariane [133]), the complexity of PDL locks is required by the implementation. The bypass logic in Sodor is simple because the designer can statically know which stages might contain the bypass data *and* in which order to prioritize them. As soon as out-of-order behavior is introduced, a dynamic mechanism (such as those implemented in the PDL Bypass Queue and Renaming Register File) becomes required to correctly forward write data.

3.6.2 Expressivity

Lastly, we highlight the expressivity of, and design exploration enabled by, PDL. In addition to the 5 stage RV32I processor, we present:

- A 3-stage RV32I core
- A 5-stage RV32I core with a branch history table
- An RV32IM core with parallel, pipelined multiplication and division units

To demonstrate their microarchitectural differences, Table 3.3 also contains performances results for these three processors. Deriving these other designs from the original required far less effort than in a conventional design process. Reducing the number of stages from 5 to 3 required eliminating two stage separators, and modifying read locks to be reserved and acquired in the same cycle; we used a slightly simplified version of the Bypass Queue to support this efficiently. Adding a custom branch predictor required almost no change to the PDL design since the same speculation primitives can be linked with an external, RTL-implemented predictor. This did involve changing some logic to `update` predictions in the second pipeline stage once we determined an instruction was a branch. We needed to modify only about 20 lines of code from the original 5-stage design to implement each of these microarchitectures.

The RV32IM processor required noticeably more effort—it introduced new functionality and made the pipeline structure not fully linear. Similar to the design of the Ariane [133] processor, the execute stages of this processor are split based on functional unit (multiply, divide, ALU/memory), can execute in parallel, and write back data out of order (when using the Bypass Queue or

```
pipe cache(addr, dataIn, isWr)[entry, main]: int {
  idx = getIdx(addr);
  acquire(entry[idx], R);
  cline = entry[idx]; release(entry[idx]);
  hit = //cline is valid and matches addr
  if (!hit || isWr) { reserve(entry[idx], W); }
  if (hit || isWr) {
    dout = //cline data for rd, else default val
    output(dout); //enqueue response
  }
  maddr = alignAddr(addr);
  if(!hit) { newline <- main[maddr]; }
  ---
  if (!hit || isWr) { //update cache entry
    newCline = //construct from newline and data
    block(entry[idx]);
    entry[idx] <- newCline; release(entry[idx]);
  }
  //queue response for cache miss
  if (!hit && !isWr) {
    output shiftOut(newline, getOffset(addr));
}}
```

Figure 3.7: Abbreviated PDL code for a direct mapped, write-allocate, write-through cache.

Renaming Register File locks). The multi-cycle divider and multiplier are both also implemented as PDL pipelines. The former computes 1 bit of an integer division per cycle and supports only 1 concurrent operation; the latter takes two stages to implement integer multiplication and is fully pipelined, supporting 2 concurrent operations. Together, these were written in 32 lines of PDL code. A non-linear pipeline allows all execution units to run in parallel without increasing pipeline depth, providing a slight CPI benefit over a 5-stage in-order CPU, and this structure reflects the microarchitecture necessary for high performance in deeper and/or wider pipelines. Modifying the decode logic to support these new instructions and altering the pipeline structure required an additional 30 lines of PDL code.

Non-processor Designs. PDL is not inherently limited to building processors; as described above, the multiplier and divider in our RV32IM CPU were also implemented in PDL as pipelines to which the main CPU issued requests. Furthermore, PDL can express pipelined modules that carry their own state, and guarantees that requests make updates and observations to that state in order. To demonstrate this feasibility, we built a 2-stage direct-mapped data cache with a write-allocate, write-through policy, in PDL. An abbreviated version of the cache is shown in Figure 3.7. The cache contains two memory references, its `entry` array of cache lines, which would typically be implemented with SRAM, and a `main` interface to DRAM. The first stage is responsible for reading the appropriate cache line and issuing a request to main memory on a cache miss, or responding to the "caller" on a hit. The second stage waits for the response from main memory and then updates the appropriate cache line. We use the PDL Queue Lock to protect the data cache entries, effectively stalling concurrent accesses to the same cache line. We expressed this design in about 50 lines of PDL code.

3.7 Future Work

PDL provides the foundation for a new methodology of processor development that can provide high-level semantics as well as low-level control over hardware design. One potential opportunity enabled by PDL is to explore the development of high-assurance microcontrollers for safety-critical systems, such as pacemakers [100]. While PDL's *current* features are not extensive enough to implement a modern high-performance processor, it should enable rapid but safe microcontroller development.

We also believe the PDL approach of abstracting common microarchitectural structures can be extended to support more advanced designs. For instance, PDL currently only allows out-of-order execution inside branches; new extensions that abstract reorder buffers and instruction schedulers could enable instruction reordering at any point in the pipeline. Another potential extension would be to generalize speculation from branch prediction to arbitrary value speculation.

To increase confidence in PDL's claims, the semantics and correctness guarantees of PDL could be formalized precisely, and the compilation strategy (or compiler itself) could be proven correct. We also believe that PDL's correctness guarantees could be extended to provide *security* guarantees. PDL's explicit treatment of speculation as a language construct may allow simpler reasoning about hardware speculation security properties, such as strictness-ordering [3] or non-interference [139] of speculative state.

3.8 Related Work

An old but short line of work aims to automatically generate pipelined processors from sequential specifications. Paul and Kroening [68] generated the stall and forwarding logic for an ordered list of stages with register assignments and combinational logic. Similarly, Nurvitadhi et al. [88] translate "transactional specifications" into pipelines; their work supports speculation and allows developers to selectively enable bypass paths via an iterative design tool. More recently, Liu et al. [73] demonstrated, with their ASSIST framework, how to synthesize high-performance, customized RISC architectures from a micro-op language. All of these projects are limited to strictly in-order pipelines, and can only generate specific implementations of speculation and bypassing.

Although the ASSIST framework can search through different timings with varied number of stages and bypass paths, it operates at a higher level of abstraction than PDL; the designer has no control over pipeline organization or optimization. This design makes autotuning tractable, but greatly limits the space of possible processor designs. PDL, on the other hand, admits more general pipeline DAGs that do not require fully in-order execution, and is a language, rather than a one-off tool. This enables static analysis and other techniques to improve PDL and provide further guarantees about PDL pipelines. PDL's hazard locks and speculation API offer more flexibility, demonstrating that processor components with a variety of implementations can be abstracted behind a single checkable interface.

TL-Verilog [56] is a language for designing pipelines with abstract timing. As with PDL's stage separators, TL-Verilog allows the designer to split combinational logic into sequences of stages with annotations. However, TL-Verilog's focus is on ensuring an equivalent semantics between the abstractly timed design and the physically timed implementation. TL-Verilog provides designers with low-level control of timing but unlike PDL, does not prevent data hazards.

BSV [87], Koika [20] and the BSV-based Kami [28] are considered high-level HDLs, because their transactional "one-rule-at-a-time" semantics can simplify correctness reasoning. Rules in these languages must execute in a single cycle, and thus their atomicity guarantees cannot automatically provide the *one-instruction-at-a-time* semantics of PDL; in particular, their compilers cannot automatically detect data hazards like PDL can. However, PDL is targeted more specifically at pipeline development, and thus these languages are more general-purpose.

High-level synthesis (HLS) tools [32] might appear similar but aim to solve a very different problem: automatically generating a timed hardware implementation from a sequential, untimed algorithm description. Because HLS primarily focuses on statically scheduling dataflow operations across hardware resources, it is unsuitable for synthesizing processors, which exhibit dynamic data dependencies. There are some recent examples of simple HLS processors [94], but these require the designer to explicitly denote bypassing logic as if they were writing RTL. Researchers have recently proposed using dynamic scheduling to improve the performance of HLS pipelines [61], using load–store queues to both enforce dynamic data dependencies and schedule memory operations. Unlike all of these tools, which rely on automatic scheduling, PDL gives hardware designers direct control over pipeline design and timing. Additionally, PDL supports integration with custom RTL implementations for breaking data dependencies via its lock API, rather than requiring a fixed implementation.

Type systems have been used in recent HLS languages [32] to provide other forms of static guarantees, focusing on *performance* rather than correctness. Dahlia [86] uses affine types to ensure *predictable* performance and to avoid synthesizing complex arbitration logic. Aetherling [41] automatically compiles data-parallel programs to streaming accelerators, using a type-directed search for hardware scheduling. Both of these tools target development of statically scheduled hardware accelerators rather than processors.

The Jade [93] language for distributed computing contains primitives similar to PDL's hazard lock `reserve`. While they are also used to manage concurrent access to shared state, Jade locks do not support speculation and are not statically checked for correct use.

3.9 Conclusion

PDL is a *Pipeline Description Language* that raises the abstraction of RTL to provide one-instruction-at-a-time semantics while enabling efficient, parallel execution by modularizing bypassing and speculation logic. PDL still gives architects low-level control over the microarchitecture and timing of their processor, and is compatible with RTL implementations of modules for branch prediction and hazard resolution. We have shown that a variety of RISC-V microarchitectures can be implemented in PDL with acceptable overhead. Through its flexible abstractions for hazard resolution and speculation, PDL promises to ease the burden of processor verification while still allowing out-of-order and speculative microarchitectures.

CHAPTER 4
SPECVERILOG: ADAPTING INFORMATION FLOW CONTROL FOR SECURE SPECULATION

The prior chapter focused on novel tools for hardware design that can provide higher level guarantees; however, it did not address the ability to verify timing-sensitive security properties of low level hardware descriptions. This chapter presents a security-typed HDL that prevents speculative timing attacks.

4.1 Introduction

Spectre and Meltdown [103, 72] have exposed significant timing-channel vulnerabilities ingrained in the designs of modern processor microarchitectures. In response, the software security community has developed a number of mitigations, formal security conditions, and principled defenses [67, 50, 24, 51, 118]; however, most of this work requires assuming certain microarchitectural behaviors or models. To make the job of software easier, architects have proposed many hardware defenses [129, 127, 4, 3, 74, 97, 101, 123, 126] that provide stronger speculative security guarantees, including invisible speculation [127], transient non-observability [74], and strictness-ordering [3]. Implementations defending these conditions can still allow most speculative execution to minimize run-time overhead, while providing reasonable semantics on which secure software can be built. Despite this plethora of designs, there have been few efforts to ensure that synthesizable implementations of these defenses actually uphold their proposed security guarantees.

One established technique for verifying security properties of Register Transfer Level (RTL) hardware descriptions is Information Flow Control (IFC) [38, 139]. In fact, several aforementioned defenses claim to provide IFC-inspired security conditions [50, 27, 130, 74, 51]. We propose using an IFC type system for an RTL Hardware Description Language (HDL) to statically verify that the hardware synthesized from the design is guaranteed to be free of transient execution vulnerabilities. However, this task is not as simple as implementing a speculative processor in an IFC-typed HDL [47]; limitations of the existing languages make it a challenge to defend speculative noninterference and related security conditions.

In this work, we address these limitations and show how an IFC-typed HDL can be extended to guarantee speculative security. Specifically, we adapt *erasure labels* [29], which express a limited form of temporal IFC policies, to an RTL language. Erasure labels allow us to prevent misspeculated data from persisting and influencing non-transient execution, without needing an explicit functional specification for "misspeculation". We also incorporate a novel form of permissive dynamic label checking to enable dynamic scheduling of concurrent instructions. This is one of the first efforts to statically check RTL hardware designs for speculative security guarantees, and the first that does not require significant manual proof effort by the designer.

This chapter describes our contributions:

- Section 4.2 provides background on transient execution vulnerabilities and also the capabilities and limitations of some existing IFC tools for RTL design.
- Section 4.3 describes SpecVerilog[1], our extension to an existing IFC HDL. We also demonstrate that erasure labels in SpecVerilog can be employed to detect transient execution vulnerabilities and verify secure mitigation mechanisms.
- Sections 4.4 and 4.5 provide formal descriptions of SpecVerilog's security guarantees and how they can be instantiated to prevent transient execution vulnerabilities in an out-of-order (OoO) processor.
- The remainder of the chapter describes several case studies implemented in SpecVerilog, related efforts, and a general discussion of the design of, benefits of, and future opportunities for SpecVerilog.

4.2 Background

4.2.1 Transient Execution Vulnerabilities

Speculation is a critical performance feature in modern processors which introduces *transient instructions* into the processor pipeline. Transient instructions are not part of the intended execution and thus are not allowed to influence any architectural state. Nevertheless, transient execution vulnerabilities such as Spectre, Meltdown, and a host of others [103, 72, 116, 117], exploit the presence of transient instructions to access otherwise-protected data. Such data can then

[1] The system name has been changed to facilitate blind review.

be leaked through well known microarchitectural side-channels. To avoid performance degradation, most defenses to these vulnerabilities seek to allow as much speculation as possible while limiting their effects on microarchitectural state. As understanding of these attacks has progressed, researchers have found an ever-increasing number of side-channels for transiently accessed data. Some are very subtle, such as *speculative interference attacks* [16] which leave little-to-no lasting trace within the microarchitectural state. In turn, new attacks prompt a new wave of defenses or other modifications to close the recently discovered holes.

We seek to end the cycle of attack discovery and defense development with a formal approach to comprehensively prevent transient execution vulnerabilities in realistic hardware implementations at the RTL abstraction. We extend existing IFC techniques for HDLs to safely reason about the potential influence of speculative execution. In this way, any design accepted by our tool is guaranteed to be free of transient execution vulnerabilities—even previously undiscovered ones.

4.2.2 Information Flow Control HDL

Information flow control is a technique for enforcing policies that govern the flow of information, especially policies about confidentiality and integrity [38, 139, 69, 134]. These policies are often formalized as some form of *noninterference*, which states that *high* system state does not influence *low* state. In the case of confidentiality, high corresponds to secret and low to public. When applied to RTL languages, which have explicit semantics for time passing, IFC provides timing-sensitive security guarantees. For instance, IFC has been used to implement many timing-channel resilient hardware modules, from encryption units that protect keys [60, 113], to processors [113, 47, 112] that provide architecture-level security guarantees.

Static IFC tools rely on designer-provided annotations to capture intended security policies. In the case of confidentiality, the designer annotates the secrecy of the system state. The tools then check that the described hardware design obeys the implied policy, and reject unsafe designs. In this work, we build upon and extend one such tool, SecVerilog [139], which uses a type system to check security annotations (labels) on hardware at compile time. We discuss alternative tools for checking hardware IFC properties in § 4.7.

Below is a simple example of the checks made by SecVerilog's IFC type system. Consider a set of security labels `PUBLIC` and `SECRET`, where secret information is not allowed to leak to public locations.

```
1 input d1  { SECRET } ;
2 input d2  { PUBLIC } ;
3 reg o1,o2  { PUBLIC } ;
4 always@( * ) begin
5   o1 = d1;    //FAIL! SECRET->PUBLIC
6   if (d1) o2 = d2; //FAIL! Implicit Flow from d1
7   else     o2 = 0;  //FAIL! Implicit Flow from d1
8 end
```

SecVerilog prevents direct illegal information flows, such as in line 5, by ensuring that the operands on the right-hand side of an assignment are permitted to flow to the destination on the left-hand side. Additionally, SecVerilog prevents *implicit flows*, such as in lines 6–7, by ensuring that expressions used in branch conditions may flow to conditionally assigned destinations.

4.2.3 Dynamic Labels

While SecVerilog is a static tool, it allows policies that depend on run-time behavior. It uses *dynamic labels* [141]: security annotations that are determined by run-time values. In a security-typed HDL like SecVerilog, dynamic labels effectively allow the same physical register to store data from different security levels over time. In the following snippet, the `mode` register is used to describe the secrecy of the `data` register's contents.

```
1 // L(0) = PUBLIC; L(1) = SECRET
2 input new_mode { PUBLIC } ;
3 reg        mode { PUBLIC } ;
4 reg        data { L(mode) } ;
```

The function `L(x)` is a dynamic label that maps run-time values onto security levels as described on line 1. For instance, whenever `mode` stores the value `1`, we know that `data` stores secret information. This kind of dynamic label can model an architecture like ARM TrustZone [76], where the processor can switch between secure and insecure worlds.

Since a given register's security level can change when the clock ticks, to check non-blocking assignments, SecVerilog considers the destination's *next-cycle* label. Using the types above, we can consider how SecVerilog checks a `mode` change:

```
1 reg       data  {L(mode)};
2 always@(posedge clk) begin
3   mode <= new_mode;
4   data <= (new_mode < mode) ? 0 : data;
5 end
```

SecVerilog requires that values being written into register data are allowed to flow to L(new_mode), since that will be the next-cycle label of data. For instance, if mode is currently 1 (and thus data is secret), but ”new_mode is 0 (and thus data” will become public), SecVerilog only accepts the design if the contents of data are overwritten with public information. Line 4 includes the dynamic check necessary for SecVerilog to conclude that the design is secure.

Although powerful, dynamic labels can also introduce subtle security vulnerabilities. One such problem is the “label of labels” consideration [66]; in our prior example, mode itself is a run-time signal and thus has its own label. Therefore, comparisons on dynamic labels can cause implicit flows. SecVerilog avoids these issues with well-formedness assumptions.

Nevertheless, SecVerilog’s dynamic labels cannot sufficiently express and defend speculative security conditions. When speculation is involved, the true security level is only determined in the future when the transient state is either invalidated or affirmed.

4.2.4 Speculative Noninterference Conditions

Prior work has established a variety of noninterference conditions intended to prevent transient execution vulnerabilities [130, 50, 74]. At a high level, these conditions guarantee that speculatively accessed data does not influence attacker-visible state. The definitions of "attacker-visible" and the scope of which speculatively accessed data is protected vary slightly in each, leading to stronger or weaker security guarantees. For this work, we consider a strong, timing-sensitive noninterference condition similar to transient non-observability [74], that we call Transient Noninterference and define formally in § 4.5. Informally, transient non-observability states that transiently accessed instruction operands should not influence the time at which non-transient instructions commit.

To defend Transient Noninterference with an IFC system, we need a lattice of labels that defines both the speculative status of hardware state and when influence is allowed. The $\sqsubseteq$ relation (read: "flows to") defines allowed influence. A first attempt at such a lattice might be the following:

$$\text{COMMIT} \sqsubseteq \text{SPEC} \sqsubseteq \text{MISS}$$

This set of labels allows committed data to influence everything; misspeculated data cannot influence anything; and unresolved speculative data is in the middle. Unfortunately, this set of labels requires violating noninterference to promote data from `SPECULATIVE` to `COMMITTED` once we learn the speculation was correct. For instance, the following example implements logic to relabel data upon discovering misspeculation or commitment, but fails to type-check:

```
1  //S(0) = COMMIT; S(1) = SPEC; S(2) = S(3) = MISS
2  wire       specCorrect, specMiss  {COMMIT};
3  reg [1:0]  isSpec  {COMMIT};
4  reg        specData  {S(isSpec)};
5  always@(posedge clk) begin
6    if (specCorrect)
7      isSpec <= 0; //FAILS: Downgrades SPEC to COMMIT
8    else if (specMiss)
9      isSpec <= 2; //OK: Upgrades data to MISS
10 end
```

Line 7 fails to type-check in SecVerilog. The value of `specData` stays the same, but its label moves down in the lattice, so this assignment appears to SecVerilog to violate noninterference. In the current cycle, `specData` may be speculative and in the next cycle the same data will be treated as committed. This is exactly the designer's intention, but it is not captured by the labels used and cannot be verified as safe by SecVerilog without voiding the language's security guarantees.

Other attempts to map speculative security conditions onto SecVerilog's labels are equally fraught. For instance, consider another dynamic label that only upgrades data upon discovering misspeculation:

```
//S(0) = COMMIT; S(1) = MISS;
reg isMiss, commData {COMMIT}; reg specData {S(isMiss)};
always@(posedge clk) begin
  //If not speculative RIGHT NOW, forget the dynamic label
  commData <= (!isMiss) ? specData : commData;
end
```

Line 5 illustrates the issue with this labeling scheme; the semantics of `S(x)` allow the data to influence committed state on *any cycle* where `isMiss` is false. However, `isMiss` may become true in the future, and thus `commData` may contain misspeculated state.

This problem cannot be solved with conventional IFC labels because they cannot reason about *future* events. SecVerilog's dynamic labels must immediately resolve to a specific security level since they are functions of the current state. Therefore, in order to precisely encode the concept of misspeculation, one would need a function from the current system state to whether or not a speculative event is predicted correctly; this is impossible for any interesting uses of speculation. The whole point of speculation is to optimistically predict the correct execution path to avoid blocking on long-latency operations; if one could immediately determine the correct prediction, speculation would be unnecessary!

4.3 Erasure Policies & Secure Speculation

Our key insight is that we can add speculative security guarantees to SecVerilog by extending it with *erasure policies* [30, 31]. An erasure policy is a form of information-flow policy that allows specifying *when* data must be removed from a system. For example, a software web app might enforce the erasure policy that a user's session data must be deleted after their session expires. In the context of processor development, erasure policies can be used to specify that transiently accessed data (and anything derived from it) must be deleted after misspeculation is discovered. We incorporate erasure policies into an extension of SecVerilog that we call SpecVerilog in order to express speculative security conditions.

SpecVerilog supports erasure policies through *erasure labels*, security annotations that can express erasure policies in an IFC system. We adapt prior work on software erasure labels [30] to RTL hardware design. Our novel contributions include *dynamic* erasure labels and enforcing erasure policies *fully statically* (i.e., without requiring a run-time monitor).

4.3.1 Erasure Labels

In SpecVerilog, erasure labels take the following form:

$$b_1 \ {}^{c(\vec{x})}\!\!\nearrow b_2$$

Here, b_1 and b_2 are (potentially dynamic) security labels, and $c(\vec{x})$ is an *erasure condition*. Erasure labels guarantee that label b_1 is enforced, *until* the current system state implies the erasure condition is true. After that point, b_2, the stricter label, is enforced. This mandatory enforcement of a stricter label is called *erasure*. As in prior work, erasure is end-to-end, meaning that the erasure of some data implies the erasure of all data derived from it as well. End-to-end erasure is specified formally in § 4.5.

Let us consider an example of erasure in SpecVerilog. For brevity, from here on we use $\perp$ to represent the least restrictive label (e.g., `COMMIT`) and $\top$ to represent the most restrictive (e.g., `MISS`).

```
1 input doErase  { ⊥ };
2 reg   data     { ⊥ doErase↗ ⊤ };
3 reg   top      { ⊤ };
4 always@(posedge clk) begin
5   data <= data; //FAILS! doErase may be true
6   top  <= data; //OK! top has high label
7 end
```

Since SpecVerilog does not rely on a run-time erasure mechanism, it must verify that whenever an erasure condition might be fulfilled, the relevant data is either erased or is written only to a more restrictive location. In the above snippet, line 6 is safe because the destination register has a higher label than the upper bound of `data`'s erasure label. However, line 5 is *unsafe* because it is possible that, on any given cycle, `doErase` is true and thus `data` must be erased.

To satisfy the erasure policy and SpecVerilog's type checker, we can change line 5 to:

```
data <= (doErase) ? 0 : data;
```

Effectively, SpecVerilog requires the designer to insert run-time checks which enforce the desired erasure policy.

4.3.2 Ensuring Secure Speculation

Erasure labels are a generic, design-agnostic tool for tracking information flow, but they can be used to prove that hardware designs satisfy speculative security

$$\begin{tikzcd}
i_0^{[]} \arrow[r] \arrow[dr, bend right, "s0"'] & i_1^{[]} \arrow[r] & i_2^{[]} \arrow[r] & \cdots \arrow[r] & i_n^{[]} \\
 & i_1^{[s0]} \arrow[r] \arrow[dr, bend right, "s1"'] & i_2^{[s0]} \arrow[r] & \cdots \arrow[r] & i_n^{[s0]} \\
 & & i_2^{[s0,s1]} \arrow[r] & \cdots \arrow[r] & i_n^{[s0,s1]}
\end{tikzcd}$$

Figure 4.1: Visualization of Temporal Ordering. The green box highlights all i for which `commit`(i) is true. Annotated arrows represent misspeculation, and an arrow from i to j indicates that j directly follows i in program order. `seq` is the relation defined as the transitive closure of these arrows.

guarantees like Transient Noninterference. Depending on how we apply erasure labels to a processor design, we can achieve different levels of precision or ensure different speculative security conditions.

We illustrate one possible approach that provides comprehensive security with a reasonable level of precision. We show how to defend Temporal Ordering, an approximation of Transient Noninterference introduced by Ainsworth [3]. They define Temporal Ordering as a relation between instructions x and y:

$$x \overset{T}{\Rightarrow} y \iff \texttt{commit}(x) \vee \texttt{seq}(x, y)$$

The predicate `commit`(x) is true when instruction x has either already completed or is guaranteed to complete. The predicate `seq`(x, y) is true if x comes before y in (a potentially speculative) program order. Figure 4.1 visualizes a formal definition of these predicates. $i_n^{[s0\ldots sm]}$ is the n^{th} instruction to execute in the program order produced by the sequence of *incorrect* speculative predictions $s0$ through sm. Therefore, only instructions with an empty misspeculation history represent the ISA-defined program order; this is exactly the set of instructions for which `commit`$(i_n^{[s0\ldots sm]})$ is true.

The `seq` relation (i.e., speculative program order) is equivalent to the arrows in Figure 4.1. Instruction *i* is only directly connected to *j* by an arrow if *j* is the next instruction in program order or is predicted to be the next instruction by a misspeculation *sn*. `seq` is the transitive closure over these arrows. In this way, two instructions are only part of the same speculative program order if the earlier instruction's misspeculation history is a prefix of the later's.

If all value and timing influences between instructions respect Temporal Ordering, the processor also exhibits Transient Noninterference. We formalize this property in § 4.5. To enforce Temporal Ordering with SpecVerilog labels, we only need two underlying lattice elements to represent the security of data: $\perp$ (the least restrictive element) for committed instructions and $\top$ (the most restrictive element) for misspeculated instructions. If we could precisely know, at all times, whether or not an instruction would misspeculate in the future, these simple labels would be sufficient to ensure that misspeculated instructions never influence the time at which other instructions commit. The value of erasure labels is that they track the influence of instructions even without knowing a priori whether they will misspeculate.

4.3.3 Incorporating Erasure

At a high level, we enforce Temporal Ordering by associating an index with each instruction which corresponds to its place in the speculative program order. For this design, we assume that the only source of speculation is predicting which instruction to execute next. Later, we discuss how to generalize these labels to other forms of speculation.

We define an erasure label SL(x) for any state associated with instruction x, using an erasure condition INV(x):

$$\mathrm{INV}(x) \equiv \text{missValid} \ \&\& \ \text{missId} < x$$

$$\mathrm{SL}(x) \equiv \bot \overset{\mathrm{INV}(x)}{\nearrow} \top$$

We assume that `missValid` and `missId` are control signals in the processor; on any cycle when `missValid` is non-zero, it indicates that the instruction *after* `missId` was the result of misspeculation (i.e., instruction `missId` made an incorrect prediction).

To illustrate how the SL label can be used to track speculation, we consider the update logic for the program counter (`pc`) register, which contains the address of the next instruction to execute. As the `pc` changes, we need to keep track of how speculative it is; an obvious way is to also store a tag that increments as it changes.

```
1  reg pc, pctag, spec_npc  { SL(pctag) } ;
2  wire missValid, missId, realnpc  { SL(missId) } ;
3  always@(posedge clk) begin
4    pc    <= (missValid && missId < pctag) ?
5                realnpc : spec_npc;
6    pctag <= (missValid && missId < pctag) ?
7                missId : pctag + 1;
8  end
```

There are a few interesting components of this example. First, in line 1, we use the SL label for *both* the program counter and its tag. We cannot use ⊥ for the tag label since the tag itself (i.e., how speculative the next instruction is) is influenced by speculation. Next, in line 2, we introduce the control signals for misspeculation and the "correct" pc value that is meant to fix the misspeculation. All of these are labeled recursively with SL(`missId`) since, intuitively, they must be coming from some instruction *before* the misspeculated instruction in program order. Lastly, the lines to update `pc` and `pctag` include cleanup logic for misspeculation; whenever misspeculation is detected, SpecVerilog requires that `pc` and `pctag` are overwritten with less speculative information.

In the above example, the `pctag` is incremented with each new (speculative) instruction. In this way, the INV erasure condition is consistent with the `seq` relation from Temporal Ordering:

$$\texttt{seq}(x, y) \iff \text{INV}(x) \implies \text{INV}(y).$$

However, in practice the `pctag` register is finite, so this example is unsound since incrementing eventually causes the value to wrap around, violating the above correspondence. Next, we generalize the INV erasure condition to properly track instruction order in modern processor designs.

4.3.4 Leveraging the Reorder Buffer

Speculative, out-of-order processors typically maintain a reorder buffer (ROB), which provides the source of truth for program order. The ROB is a first-in-first-out (FIFO) data structure that stores all of the metadata associated with each instruction; entries are inserted in speculative program order and are only removed when they are *committed*: that is, they have updated architectural processor state. If an entry is found to be the result of misspeculation, that entry must be invalidated before it would be committed.

Since ROB order is a proxy for instruction order, we can define a new erasure condition using the ROB's circular buffer ordering:

$$\text{INV}(x, h) \equiv \text{ missValid } \&\& \text{ } (\text{missId} - h) \,\%\, \text{sz} < (x - h) \,\%\, \text{sz}$$

sz is a design-time constant defining the size of the ROB, `missValid` and `missId` are the same control signals as before, x is the index of a given ROB entry, and h is the index of the oldest ROB entry. This circular buffer ordering uses modular arithmetic to compute the age of a given index.

In this way, the ROB can be the source of truth for all of the labels in the processor and is the only component whose labeling we need to trust. For the code examples in this chapter, we use the unbounded integer erasure conditions that can be compared with < to simplify presentation. Our case study implementations use the (more realistic) circular buffer ordering and thus contain slightly more complex logic to implement modular arithmetic comparisons.

4.3.5 Implementing Secure Modules

Hardware modules may handle both speculative and non-speculative state. Securely managing state from multiple security levels is challenging and error-prone; we show how IFC brings some of the potential pitfalls to light, and how to label secure implementations such that they type-check. For this section we use secure caches as our motivating example, but these methods apply to any hardware module that manages state influenced by multiple instructions.

The labels we've described so far require completely erasing the contents of registers that might contain speculative data. However, real implementations of secure caches [3, 4, 120] use valid bits to mark data as "erased", or even just delay potentially unsafe operations until speculation has been resolved [98]. These optimizations can also be verified as secure in SpecVerilog by using dynamic labels in conjunction with erasure conditions.

Efficient Invalidation. Valid bits can easily be incorporated as dynamic labels to efficiently mark data as unusable. We can modify our erasure labels from the previous example to support this feature:

$$\text{VALID}(b) \equiv \texttt{if}\ (b)\ \bot\ \texttt{else}\ \top$$

$$\text{SLVAL}(b, v) \equiv \text{VALID}(b) \overset{\text{INV}(v)}{\nearrow} \top$$

The VALID label allows data to flow freely whenever the valid bit, b, is set to 1, else it applies $\top$, meaning the data cannot influence anything. In this way, any data with the SLV label can be "erased" upon misspeculation by unsetting its associated valid bit.

```
1 wire missValid, missId  { SL(missId) };
2 reg  sId, sValid        { SLVAL(sValid, sId) };
3 reg  sData              { SLVAL(sValid, sId) };
4 always@(posedge clk) begin
5   sData  <= sData;//OK! erased by marking as invalid
6   sValid <= missValid && missId < sId ? 0 : sValid;
7 end
```

SpecVerilog correctly accepts the above implementation since the SLVAL label uses VALID as its lower bound; in any cycle where `sData` must be erased, its valid bit will be set to 0 and so in the next cycle it will be treated as $\top$ (i.e., *above* the required erasure level). If we had removed line 6, then this snippet would not be accepted by SpecVerilog, since there would be no guarantee that `sData` is invalidated upon misspeculation. Without the use of dynamic labels *in conjunction* with erasure labels, we would not be able to verify this optimization.

Secure Dynamic Scheduling. Secure designs also need to appropriately order or delay operations when they might affect the timing of less speculative ones. Processors that do not correctly schedule operations are potentially vulnerable to SpectreRewind [48] or other speculative interference [16] attacks. To accept implementations that correctly schedule speculative operations, while still rejecting unsound designs SpecVerilog needs to reason precisely about *label comparisons*. In this context, label comparisons are dynamic checks that determine which of two pieces of data is more speculative.

Consider a latency-insensitive interface that uses *ready* and *valid* bits for making requests to a module. Whenever the following implementation of such a module is *not* currently handling a request it sets *ready* to true and will accept *valid* incoming requests:

```
//input and output have the same label, defined by client
input  reqId, reqValid, req  { SLVAL(reqValid, reqId) };
output reqReady              { SLVAL(reqValid, reqId) };
reg    curId, curVaild, cur  { SLVAL(curValid, curId) };
always@(posedge clk) begin
  //FAIL! req might not be allowed to observe cur
  reqReady = ~curValid;
  if (reqValid && !curValid)
    //FAIL! if 0, curValid cannot influence anything
    curValid <= 1; curId <= reqId; cur <= req;
end
```

This logic is, in general, insecure. When the current request is *more speculative* than the incoming one, the incoming request is delayed, leading to a violation of Transient Noninterference. SpecVerilog correctly rejects this design since it cannot prove that SLVAL(`curValid`, `curId`) is allowed to influence SLVAL(`reqValid`, `reqId`).

In this scenario, secure designs must allow less speculative requests to preempt others, but must prevent the opposite; this practice is called *leapfrogging* in prior work [3]. SpecVerilog has a permissive label comparison operator that enables designers to implement leapfrogging without violating SpecVerilog's security guarantees.

Here we demonstrate how to compute the *ready* bit in a secure SpecVerilog implementation:

```
1 if (L(curValid) ⊑ L(reqReady))
2 //!reqValid || (curValid && curId <= reqId)
3   reqReady = ~curValid;
4 else
5   reqReady = 1;
```

Line 1 demonstrates a label comparison in SpecVerilog, which dynamically computes the labels of `curValid` and `reqReady` and evaluates to 1 only if the comparison holds. The comment describes the actual logic that computes the label comparison. We discuss this operator and its limitations further in § 4.4.3. This version safely implements preemption and, with the interface labels from the first example, is accepted by SpecVerilog. Most IFC-based type systems would reject this program because the label comparison can cause an implicit flow. SpecVerilog introduces a novel and more permissive rule that accepts the above code but does not compromise security.

Variable	x		
Level	l	$\in$	$\mathbb{L}$
Function	f	$\in$	$\mathbb{Z}^n \rightarrow \mathbb{L}$
Condition	c	$\in$	$\mathbb{Z}^n \rightarrow \mathbb{B}$
Basic Types	b	$::=$	$l \mid f(\vec{x})$
Label	τ	$::=$	$b \mid b_1 \overset{c(\vec{x})}{\nearrow} b_2 \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$
Typing Context	Γ	$\in$	$x \rightarrow \tau$

Figure 4.2: Syntax of security labels. Label functions and policies are specified with variables. Policy conditions may also contain free variables.

4.4 SpecVerilog Design

In this section, we briefly present SpecVerilog's type system, formal security guarantees, and implementation.

4.4.1 Typing Rules

SpecVerilog extends SecVerilog's existing type system with rules for erasure labels and permissive label comparisons. Our syntactic presentation here differs slightly from prior work [139, 45] but is effectively the same, other than SpecVerilog's novel contributions.

Figure 4.2 presents the syntax for security labels in SpecVerilog. Other than erasure labels and conditions, this syntax is directly borrowed from SecVerilog.

It allows users to define their own underlying security labels, and to define dynamic labels as dependent types (i.e., functions of program state). Erasure labels are formed from two non-erasure labels and an erasure condition[2]. Erasure conditions are functions of run-time state that return true or false; when they return true, the data labeled with this condition must be erased.

Since SpecVerilog is dependently typed, we also refer to the current system state in some of our rules and definitions. In this presentation we keep the state mostly abstract; this table describes our syntax:

Syntax	Operation
σ	Current system state
$\sigma[x]$	Value of variable x
$\sigma[\vec{x}]$	Value of list of variables $\vec{x}$
$\sigma \longrightarrow \sigma'$	Clock-tick transition to next state

Well-formedness. We assume several well-formedness conditions about a program's types, which are defined in Figure 4.3. First, dependent labels must only depend upon variables whose labels are less restrictive; this prevents unwanted information flow channels through label checking. Second, we require that all variables appearing in dependent types are either the same as the variable of the type (i.e., a recursive label) or they must be *sequential* variables (i.e., variables whose values only change on a clock edge). This restriction ensures that any time a dynamic label changes its value, the change is checked for safety by the typing rules. Lastly, we assume that all erasure policies actually repre-

[2] Erasure labels cannot be nested, but this does not limit expressiveness. Taking the least upper bound ($\sqcup$) of multiple erasure labels can achieve the same effect as placing erasure conditions inside of lower or upper bounds.

sent *upgrades*; after erasure, the policy must be at least as restrictive as before erasure.

1. $\forall v \in \textbf{Vars}.\forall v' \in FV(\Gamma(v)).\forall \sigma.\text{obs}(\Gamma(v'))\ {}_{\sigma}\!\sqsubseteq \text{obs}(\Gamma(v))$
2. $\forall v \in \textbf{Vars}.\forall v' \in FV(\Gamma(v)).v' \neq v \implies v' \in \texttt{seq}$
3. $\forall \tau_1\ {}^{c(\vec{x})}\!\!\nearrow \tau_2.\forall \sigma.\tau_1\ {}_{\sigma}\!\sqsubseteq \tau_2$

Figure 4.3: The well-formedness conditions for SpecVerilog type environments. **Vars** is the set of all variables in the program; $FV(\tau)$ is the set of variables referenced in the type τ; obs(τ) is data visibility and is defined in Figure 4.8.

4.4.2 Type Checking

SpecVerilog's type-checking rules primarily rely on a *may-flow-to* relation, $\sqsubseteq$, which describes allowed influences between security types. As is typical for IFC, this relation is reflexive and transitive and relies on the underlying security lattice ordering. The complete definition of $\sqsubseteq$ can be found in Figure B.2 in the appendix.

$\boxed{\tau \sqsubseteq \tau'}$

$$\textsc{Erase-Elim}\ \frac{\tau_2 \sqsubseteq \tau'}{\tau_1\ {}^{c(\vec{v})}\!\!\nearrow \tau_2 \sqsubseteq \tau'} \qquad \textsc{Erase-Intro}\ \frac{\tau \sqsubseteq \tau_1}{\tau \sqsubseteq \tau_1\ {}^{c(\vec{v})}\!\!\nearrow \tau_2}$$

$$\textsc{Erase-Weaken}\ \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2' \qquad \forall \sigma.\sigma \vDash c(\vec{v}) \implies \sigma \vDash c'(\vec{v}')}{\tau_1\ {}^{c(\vec{v})}\!\!\nearrow \tau_2 \sqsubseteq \tau_1'\ {}^{c'(\vec{v}')}\!\!\nearrow \tau_2'}$$

Figure 4.4: The environment-independent may-flow-to relation for erasure labels.

$$\boxed{\tau \downarrow_\sigma \tau'}$$

$$\textsc{Labels}\ \frac{}{l \downarrow_\sigma l} \qquad \textsc{Functions}\ \frac{\vec{v} = \sigma[\vec{x}] \qquad l = f(\vec{v})}{f(\vec{x}) \downarrow_\sigma l}$$

$$\textsc{Erase}\ \frac{\tau_1 \downarrow_\sigma \tau_1' \qquad \tau_2 \downarrow_\sigma \tau_2' \qquad \vec{v} = \sigma[\vec{x}]}{\tau_1 \,{}^{c(\vec{x})}\!\!\nearrow \tau_2 \downarrow_\sigma \tau_1' \,{}^{c(\vec{v})}\!\!\nearrow \tau_2'}$$

Figure 4.5: The function that resolves variables in labels.

$$\textsc{ComMFT}\ \frac{\tau \downarrow_\sigma \tau' \qquad \tau_1 \downarrow_\sigma \tau_1' \qquad \tau' \sqsubseteq \tau_1'}{\tau \;{}_\sigma\!\!\sqsubseteq \tau_1}$$

$$\textsc{SeqMFT}\ \frac{\neg\text{Erase}(\sigma, \tau, \tau_1) \qquad \sigma \longrightarrow \sigma' \qquad \tau \downarrow_\sigma \tau' \qquad \tau_1 \downarrow_{\sigma'} \tau_1' \qquad \tau' \sqsubseteq \tau_1'}{\tau \;{}_\sigma\!\!\sqsubseteq_{\mathbf{next}} \tau_1}$$

Figure 4.6: May-flow-to relations parameterized on a given system state. Rules ComMFT and SeqMFT type-check combinational and sequential assignments, respectively.

$$\textsc{ComAssign}\ \frac{\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \notin FV(\tau') \qquad C \implies pc \sqcup \tau \;{}_\sigma\!\!\sqsubseteq \tau'}{C, \Gamma, pc \vdash x = e}$$

$$\textsc{SeqAssign}\ \frac{\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \notin FV(\tau') \qquad C \implies pc \sqcup \tau \;{}_\sigma\!\!\sqsubseteq_{\mathbf{next}} \tau'}{C, \Gamma, pc \vdash x <= e}$$

$$\textsc{LabelComp}\ \frac{\Gamma(v_1) = \tau_1 \qquad \Gamma(v_2) = \tau_2 \qquad \forall \sigma. \tau_1 \;{}_\sigma\!\!\sqsubseteq \tau_2 \lor \tau_2 \;{}_\sigma\!\!\sqsubseteq \tau_1 \qquad \tau = \tau_1 \sqcap \tau_2}{\Gamma \vdash v_1 \sqsubseteq v_2 \dashv \tau}$$

Figure 4.7: Type checking rules for selected statements and expressions in SpecVerilog. C is a set of constraints about the current (σ) and next (σ') states. *pc* tracks the label of variables read in the current context.

In Figure 4.4, we present the most interesting rules for SpecVerilog: those concerning erasure labels. The ERASE-INTRO rule allows us to add an erasure condition onto an existing policy and the ERASE-ELIM rule allows us to replace an erasure label with a more restrictive label. The ERASE-WEAKEN rule describes how to replace one erasure label with another. Influence is allowed if the lower and upper bounds both may flow, and if the new erasure condition would evaluate to true any time the original condition would evaluate to true, regardless of the system state.

As in SecVerilog, we use two different typing rules based on whether the destination is updated combinationally, or sequentially. Figure 4.6 describes how we resolve variables based on the kind of assignment. For combinational assignments (i.e., blocking), all dependent types are resolved in the current context. For sequential assignments (i.e., non-blocking), the label of the destination is evaluated in the next-cycle context instead. Additionally, for sequential assignments, we require that the source does not need to be explicitly erased with the Erase side condition. This condition returns true when the ERASE-WEAKEN rule must be applied to prove that the flow is allowed *and* the left hand erasure condition in that rule is true this cycle. Figure 4.7 demonstrates the actual typing rules for assignment statements in SecVerilog, which reference the variable resolution rules in Figure 4.6. We omit the rules for when labels are recursive for both combinational and sequential assignments for brevity; they are nearly identical to those from SecVerilog.

4.4.3 Erasure Label Design

The evaluation of erasure conditions in Figures 4.4 and 4.5 has a peculiar-looking definition, so here we justify its design. Unlike normal dynamic labels, erasure conditions are allowed to contain *free variables*. Consider our misspeculation example from § 4.3.3:

$$\mathrm{INV}(x) \equiv \ \mathrm{missValid}\ \&\&\ \mathrm{missId} < x$$

INV has two free variables, missValid and missId. When applying the ERASE-WEAKEN rule to check if INV(x) implies some other condition, the implication must hold for any possible values of missValid and missId, but only for a concrete value of x.

This design is necessary to make erasure labels usable while still ensuring erasure conditions are checked in the future. Obviously, if we resolve all of the variables to values before checking, then the implication will hold on any cycle where INV is true, which would (inappropriately) allow us to stop monitoring erasure conditions. On the other hand, leaving all variables free would prevent typing clearly safe programs. Effectively, we would be unable to leverage knowledge of current and future system state to weaken the label.

The following snippet demonstrates a safe program which would not typecheck under the more restrictive treatment of erasure weakening. In this example, we check an erasure condition and then conditionally copy speculative data into a register.

```
input rId, rData  { SL(rId) } ;
reg   sId, sData  { SL(sId) } ;
always@(posedge clk) begin
//assume erase properly checks the erasure condition for
   rId
  if (!erase) begin
    sData <= rData; sId <= rId;
  end
end
```

If we did not resolve variables in the ERASE-WEAKEN rule, we would need to prove the following to type-check the above code:

$$\forall\, \text{missValid}, \text{missId}, \text{sId}, \text{rId}.$$
$$(\text{missValid} \;\wedge\; \text{missId} < \text{sId}) \implies (\text{missValid} \;\wedge\; \text{missId} < \text{rId})$$

Unfortunately, the above statement does not hold; we cannot actually prove the assignment is safe without relating the current value of `sId` to the new value of `rId` after the assignment.

With the ERASE-WEAKEN rule from Figure 4.4 and resolving variables according to the correct cycle values as in Figure 4.6, we can correctly type-check the prior example. We need to prove:

$$\forall\, \text{missValid}, \text{missId}. (\text{missValid} \;\wedge\; \text{missId} < \text{sId}) \implies$$
$$(\text{missValid} \;\wedge\; \text{missId} < \text{next}(\text{rId}))$$

This can be proven by statically analyzing the program. In the context of the assignment we know sId = next(rId); in this case the condition trivially holds.

Dynamic Label Checks. In SecVerilog (and other IFC type systems with dynamic labels), run-time label comparisons use the same rule as any binary operation: the label of the result is the join ($\sqcup$) of the labels of the operands. Unfortunately, this rule is too restrictive for efficient processor designs; specifically, it makes it impossible to type-check dynamically scheduled modules such as caches. Effectively, when handling concurrent requests with arbitrary labels, there is no way to dynamically schedule them in a timing-safe manner. The scheduling choice will leak information about the *labels* of the concurrent requests to each other via a timing channel.

However, in the special case where the labels are guaranteed to be *ordered* then there is a safe implementation (namely the preemption described in § 4.3.5). We introduce a more permissive typing rule in Figure 4.7: LabelComp, which effectively uses the *lower label* as the label of the result of the comparison and allows SpecVerilog to type-check the safe implementation. This is an *unintuitive* result which relies on the fact that an attacker can be statically sure that one of labels *must* be able to flow to the other. We prove the soundness of this rule in Appendix B.2.

4.4.4 Compiler Implementation

SpecVerilog builds upon the existing SecVerilog type checker, adding erasure labels[3]. To support dependent types, SecVerilog relies on the Z3 SMT solver [36] for type checking. Dynamic labels are specified by the user as Z3 functions, which are referenced by the constraints that the SecVerilog type checker generates.

In SpecVerilog, users also supply erasure conditions as Z3 functions. We

[3] The implementation is a fork of Icarus Verilog and can be found at `https://github.com/dz333/secverilog`.

support new syntax for erasure labels directly in Verilog source code that can reference these erasure conditions. SpecVerilog generates constraints based on the may-flow-to relation and typing rules described in Figures 4.4 and 4.7 and discharges these constraints to the Z3 SMT solver to correctly type-check designs using erasure labels.

We made several other modifications and improvements to the SecVerilog compiler to incorporate features from other research efforts on IFC type systems for hardware [139, 45, 46]. The modifications improved the efficiency of the compiler (i.e., simplified the generated Z3 constraints), enabled us to precisely type-check our most complicated examples, and fixed some bugs in existing type checker implementations.

Permissive Label Comparisons. We have not implemented type checking for the permissive LABELCOMP rule in the current SpecVerilog prototype. The difficulty in implementing this rule is that it requires automatically generating Verilog code *from label definitions* that correctly define the may-flow-to relation, as opposed to generating Z3 constraints *from Verilog code*. It is certainly viable to implement such a translation for a limited, but sufficiently expressive, subset of Verilog expressions; nevertheless, we leave it as future work. In our example processor implementations, we manually translate label comparison operations into Verilog expressions and use a *declassify* statement to explicitly label the result as per the LABELCOMP rule.

$$\boxed{\text{obs}(\tau) = \tau'} \qquad \begin{aligned} \text{obs}(\tau_1 \sqcap \tau_2) &= \text{obs}(\tau_1) \sqcap \text{obs}(\tau_2) \\ \text{obs}(\tau_1 \sqcup \tau_2) &= \text{obs}(\tau_1) \sqcup \text{obs}(\tau_2) \\ \text{obs}(b_1 \overset{c(\vec{x})}{\nearrow} b_2) &= \text{obs}(b_1) \\ \text{obs}(b) &= b \end{aligned}$$

Figure 4.8: The obs function defines the visibility of data with a given label. Erasure labels use their lower bound.

4.5 Security Guarantees

In this section, we formalize SpecVerilog's security guarantees. First, we describe noninterference and end-to-end erasure, conditions that hold for all well-typed SpecVerilog programs. Then, we define Transient Noninterference with respect to an abstract processor model. Finally, we show how to use erasure labels on a real processor design to statically guarantee that it satisfies Transient Noninterference.

4.5.1 Noninterference and Erasure

SpecVerilog provides traditional IFC-style security guarantees, which we formally describe in this section.

Observational Equivalence. First, we define how an observer (i.e., attacker) can distinguish different executions. We assume that an observer can be defined by a level, l. They can observe the value of any variable which may influence data with level l. Additionally, since the labels of variables may change during execution, we consider the *set of variables* that may influence l to be visible to the observer as well. We define an observation function in Figure 4.8 that translates labels in a given environment into the corresponding observable lattice level. The interesting part of this function is for erasure labels, which defines the *lower bound* of the erasure label to be the observable level. Data marked by an erasure label is considered visible at the lower bound until it must be erased.

Given this observation function, we can define *observational equivalence*. When two program states are observationally equivalent with respect to some level, l, any attacker that can observe l *cannot* distinguish the two states.

Definition 4.1 (Observational Equivalence)**.** *We define two states to be observationally equivalent w.r.t level l ($\sigma_1 \approx_l \sigma_2$) when:*

$$\forall x \in \mathit{VARS}.\, o = \mathrm{obs}(\Gamma(x)) \;\wedge\; o_{\sigma_1} \sqsubseteq l \iff o_{\sigma_2} \sqsubseteq l \;\wedge$$
$$o_{\sigma_1} \sqsubseteq l \implies \sigma_1[x] = \sigma_2[x]$$

Well-typed SpecVerilog programs exhibit noninterference. Simply put, noninterference ensures that, if an attacker cannot distinguish two states, then the attacker will not be able to distinguish the result of executing those states.

Definition 4.2 (Noninterference)**.** *A program is noninterfering if for an attacker defined by an observation level l, observational equivalence is preserved during execution:*

$$\forall l \in \mathcal{L}, i \in \{1, 2\}. \sigma_i \longrightarrow \sigma_i' \wedge \sigma_1 \approx_l \sigma_2 \implies \sigma_1' \approx_l \sigma_2'$$

These definitions of observational equivalence and noninterference are standard for IFC systems with dynamic labels and mirror those of SecVerilog.

End-To-End Erasure. All well-typed SpecVerilog programs also enforce *end-to-end erasure*. Intuitively, erasure ensures that, once an erasure policy's condition is fulfilled, the labeled data must only have influenced state that can be observed at or above the upper label. Our definition of erasure is inspired by Hunt and Sands [58], but we modify it to account for our definition of observability, and also to incorporate dynamic labels and semantic erasure conditions.

While the following specification of end-to-end erasure is dense, it can be summarized concisely. If, at any point, some variable will eventually need to be erased, then replacing that variable with an uninterpreted value and continuing execution results in a post-erasure state that is indistinguishable (for a low observer) from an execution that uses the original value.

Definition 4.3 (End-To-End Erasure). *Given an infinite trace of system states: $\sigma_0 \longrightarrow \sigma_1 ... \longrightarrow \sigma_n ...$, if for all variables x, the erasure policy of x in state σ_i is ever satisfied in some future state, σ_j, then replacing x in σ_i with an uninterpreted value ($\bot$) results in a future state that is l-equivalent to σ_{j+1} for any l that the upper bound on x's erasure policy cannot observe.*

$$
\begin{aligned}
\forall i, j, x, c(\vec{y}), b, l. \quad & i \le j \ \wedge\ (c(\vec{y}), b) \in \text{eraseTo}(\Gamma(x)) \ \wedge \\
& \vec{v} = \sigma_i[\vec{y}] \ \wedge\ \sigma_j \vDash c(\vec{v}) \ \wedge\ \sigma'_i = \sigma_i[x \mapsto \bot] \ \wedge \\
& \sigma'_i \longrightarrow^{j-i+1} \sigma'_{j+1} \ \wedge\ b \not\sqsubseteq_{\sigma_i} l \\
& \implies \sigma'_{j+1} \approx_l \sigma_{j+1}
\end{aligned}
$$

This definition relies on the eraseTo function, which returns a set of pairs of erasure conditions and levels. When the condition evaluates to true, the variable must be erased to the given level. For simple erasure labels, eraseTo is specified in the obvious way, returning the erasure condition and the upper bound label. Definition B.1 in the appendix describes the complete eraseTo function.

4.5.2 Speculative Security

Here we formalize a strong and usable definition of speculative security and sketch how SpecVerilog can enforce this condition by applying Temporal Ordering-based labels.

Attacker Model. Most prior models of speculative security [51, 50, 130] are made with respect to an attacker that can make direct observations of microarchitectural state or actions (such as caches, branch predictors, or speculatively loaded addresses). However, these models can both lead to unsoundness by overlooking potential attacks or, conversely, overestimate the attacker's power.

Instead, we model software-level timing side-channel attackers more realistically: attackers can observe the time at which each instruction completes but they may not observe intermediate microarchitectural states. Our model faithfully reflects an attacker that can execute code on the processor and make deductions from the timing of its execution, but does not reflect the power of attackers with physical access to the hardware (who might exploit other side-channels such as power or EM radiation).

Transient Noninterference. As mentioned in § 4.2.4, Transient Noninterference effectively enforces a security condition defined by prior work, transient non-observability. However, in order to formally show that Transient Noninterference is safe with respect to our strong attacker model, we use a definition more similar to Unique Program Execution (UPE) [43], a baseline confidentiality condition for processors. UPE says that if secret architectural state (register or memory contents) can be leaked to an attacker via timing channels, then that state must also affect the values of attacker-visible architectural state[4].

So far, we have argued that erasure labels can be used to enforce Temporal Ordering. However, Temporal Ordering is not sufficient to enforce UPE (and Transient Noninterference) on its own. Processors may pathologically leak information about arbitrary architectural state via timing channels even *without* speculative execution. For instance, a processor satisfying Temporal Ordering could use an arbitrary value in memory to pre-fetch cache lines, or to otherwise delay instruction commit; implementations exhibiting these behaviors would violate UPE. While these sorts of leaks do represent potential bugs, they are of the kind that architects and functional verification tools are likely to find and eliminate since they are likely to either break functionality completely or result in noticeable performance regression in the common case, rather than in the edge case. To get a verifiable guarantee that a processor is free of such bugs, a designer would need to use a tool that concretely defines the ISA specification and relates it to the generated circuit (e.g., PDL, our high-level hardware description language described in Chapter 3) or they would need to manually define the relationship themselves; both of these are out of the scope of SpecVerilog.

Therefore, we prove a slightly weaker theorem by restricting our guarantee to processors that only read architectural state associated with some transiently executed instruction[5]. We formalize this assumption by defining an abstract OoO processor semantics, which Figure B.1 in the appendix depicts.

The processor consists of the following components:

[4] Secret here is defined with respect to an arbitrary architecture–level security policy. "Secret" architectural state is simply an arbitrary partiion of the architectural state.

[5] Note that most defenses to transient execution attacks provide similar guarantees, as it is generally assumed processors do not exhibit these kinds of pathological vulnerabilities.

Syntax	Description
rob	Reorder buffer for in-flight instruction metadata
A	Architectural state (registers, memory, and pc)
μ	Microarchitectural state

The processor can: speculatively *fetch* instructions, placing them into the *rob*; *execute* instructions, updating μ based on the ISA–defined semantics that instruction; *commit* instructions, removing them from the *rob* and updating A; and rollback state upon discovering *mispredictions*. The processor has an abstract scheduler that determines which operations to run each cycle and is a function of μ and any architectural state read by instructions in the *rob*. We assume that the scheduler respects registered hardware semantics (i.e., persistent state can only be written once per clock cycle).

We use the $[\![\cdot]\!]$ notation to extract the (infinite) trace from executing a given processor configuration:

$$[\![\mathcal{P}_1]\!] = A_0\, A_1 \ldots A_n \ldots$$

This trace contains the entire architectural state on every clock cycle during the processor's execution.

We also define observation functions to express both architectural and timing-sensitive attackers.

Definition 4.4 (Low Architectural Observer)**.** *A low architectural observer (O_l) is defined with respect to an arbitrary subset of the architectural state ($A_l \subseteq A$). This observer can view the time-independent sequence of low architectural states.*

$$O_l(A_0\, A_1 \ldots) = \text{ if } (A_{l0} = A_{l1}) \text{ then } O_l(A_1 \ldots) \text{ else } A_{l0}\, O_l(A_1 \ldots)$$

Definition 4.5 (Timing-Sensitive Observer)**.** *A timing-sensitive low observer (T_l) can observe the A_l on every clock cycle.*

$$T_l(A_0\, A_1\, ...) = A_{l0}\, A_{l1}\, ...$$

Lastly, we formally define Transient Noninterference, which can be enforced via Temporal Ordering and SpecVerilog-checked erasure labels.

Definition 4.6 (Transient Noninterference)**.** *Processor $\mathcal{P}$ exhibits Transient Noninterference if, for any partitioning of A, if executions are indistinguishable to a low-architectural observer, then they are indistinguishable to a timing-sensitive observer.*

$$\begin{aligned}&\forall i \in 1, 2.\ \mathcal{P}_i = \langle rob_i, A_i, \mu_i\rangle,\\ &\qquad A_i = \langle A_{li}, A_{hi}\rangle,\ A_{l1} = A_{l2}, \mu_1 = \mu_2, rob_i = pc_i\\ &\qquad\qquad O_l([\![\mathcal{P}_1]\!]) = O_l([\![\mathcal{P}_2]\!]) \implies T_l([\![\mathcal{P}_1]\!]) = T_l([\![\mathcal{P}_2]\!])\end{aligned}$$

4.5.3 Enforcing Transient Noninterference

In this section, we briefly justify both why a processor that satisfies Temporal Ordering must also satisfy Transient Noninterference, and also how properly applied erasure labels enforce Temporal Ordering using the processor semantics from Figure B.1. Note that this model assumes that the only source of speculation is next-instruction prediction; this assumption is simplifying for presentation but not necessary. We discuss how to extend these results to more general speculation in § 4.8.

Temporal Ordering enforces Transient Noninterference. The timing behavior of any processor that refines the semantics in Figure B.1 is only a function of three things: data read by instructions that commit; data read by those that *only transiently* execute; and the initial microarchitectural state.

Temporal Ordering restricts influence so that different paths of speculative execution cannot influence each other; therefore, varying transiently read data has no impact on timing. Furthermore, by assumption, execution results in the same set of low-architectural observations. Since timing is therefore only a function of low-architectural state, time-sensitive observations of low-architectural state are also independent of architectural secrets.

Erasure labels enforce Temporal Ordering. We use the ROB labels described in § 4.3.4 to label our abstract processor. The ROB always contains a reference to the *oldest* instruction that is currently executing and it is guaranteed to eventually commit; we call the index of this instruction in the ROB the *head*. We label each entry in the ROB recursively based on its index, i, with the label: SL(i, *head*). All committed state has the same label as *head*; effectively we stop precisely tracking which instruction influenced architectural state once that instruction is guaranteed to commit. The rest of the processor is labeled such that it type-checks in SpecVerilog (and thus is guaranteed to respect noninterference and end-to-end erasure).

Here, we argue that noninterference and end-to-end erasure when using these labels on an abstract OoO processor guarantees Temporal Ordering. All state in the processor is a function of the architectural state accessed by *some* set of instructions, therefore so is the time at which each instruction commits. We write $\mathcal{I}(x)$ to denote the set of instructions that have influenced register x at some point in the execution. We use the notation from Figure 4.1 to denote instructions across speculative program orders. $i_j^{[s_1,...s_n]}$ denotes instruction j that was the result of the (incorrect) speculative predictions s_1 through s_n.

Theorem 4.1 (Enforcing Temporal Ordering). *For any two registers in a well-typed $\mathcal{P}$, x and y, if x may influence y according to SpecVerilog, then that influence obeys Temporal Ordering.*

$$\forall x.y.i_j^{[s_1,...,s_n]} \in \mathcal{I}(x), i_k^{[p_1,...,p_n]} \in \mathcal{I}(y), \sigma.$$
$$\Gamma(x) \,{}_{\sigma}{\sqsubseteq}\, \Gamma(y) \implies i_j^{[s_1,...,s_n]} \overset{T}{\Rightarrow} i_k^{[p_1,...,p_n]}$$

The proof is relatively straightforward and relies on one main lemma; each register is only influenced by instructions from one speculative program order.

Lemma 4.1. *For any register, x, in a well-typed $\mathcal{P}$,*

$$\forall i_j^{[s_1,...,s_n]}, i_k^{[p_1,...,p_n]} \in \mathcal{I}(x), \quad j \le k \implies [s_1, ...s_n] \preceq [p_1, ..., p_n]$$

where $\preceq$ is prefix order.

Intuitively, Lemma 4.1 means that, following a misspeculation event, there are no remnants of the misspeculated instructions; the processor is always executing down only one speculative road at a time. Lemma 4.1 follows from end-to-end erasure, and induction on the OoO processor semantics. We include a proof sketch in Appendix B.2. Here we sketch the proof for Theorem 4.1.

Proof Sketch. In any given state, if $\Gamma(x) \,{}_{\sigma}{\sqsubseteq}\, \Gamma(y)$, then x is ordered before y in *some* speculative program order (by the processor semantics and ROB labels). By Lemma 4.1, $\mathcal{I}(x)$ and $\mathcal{I}(y)$ must each only contain instructions from a single speculation path; it must in fact be the same speculation path since they are ordered. Therefore, we have $\mathcal{I}(x) \subseteq \mathcal{I}(y)$, which directly implies Theorem 4.1. □

4.6 Case Studies

Case Study	Annotation Burden (Lines)			Register	Solver Time
	Labeled	Changed	Original	Overhead	(sec)
Reorder Buffer	22	5	86	None	0.247
Cache + GM	250	9	1527	None	1.43
Predictor	16	0	68	None	0.104
Predictor + GM	38	6	157	None	0.993
Renaming RF	117	26	366	1 label/replica	36.9

Case Study	Mitigation Strategy			SpecVerilog Features Used
	Delay	Rollback	Partition	
Reorder Buffer	–	✓	–	Label comparisons, Input assertions, Valid-entry labels
Cache + GM	✓	✓	✓	Label comparisons, Valid-bit labels, Explicit erasure
Predictor	✓	–	–	Label comparisons *or* Input assertions
Predictor + GM	✓	✓	✓	Label comparisons, Valid-bit labels
Renaming RF	✓	✓	–	Label comparisons, Input assertions, Valid-bit labels

Table 4.1: A qualitative summary of secure hardware module case studies. Annotation burden is relative to a secure design in plain Verilog. Labeled counts the lines that needed explicit labeling. Changed includes modifications and additions needed to satisfy the typechecker. Original is the total lines of the original Verilog description.

In addition to our theoretical results, we have empirically evaluated the utility of SpecVerilog through a number of case studies. Table 4.1 provides a high-level summary of our case studies, the re-write effort required to satisfy the type-checker, and the mitigation techniques demonstrated by the example. Each of our case studies targets a component that is critical to the design of speculative, OoO processors. Some are known to contribute to transient execution vulnerabilities, while we chose others to illustrate that SpecVerilog can still accept complex, yet safe, designs. For each of these case studies we used the ROB-based labels described in § 4.3.4 to label module inputs and outputs.

4.6.1 Case Study Modules

Reorder Buffer. We implement the skeleton of a reorder buffer to illustrate that our source of truth for labels can be implemented securely and with minimal assumptions about functional correctness. Our ROB supports insertion, misspeculation and instruction commitment; we leave the data stored in each ROB entry abstract for this example.

Discussion. In this module (and all of our case studies), in order to type-check, we had to assume that the oldest entry in the ROB would never be misspeculated; this follows directly from functional correctness and is essentially a minimal assumption. Another way to phrase this would be to say that we assume that the first instruction ever executed is non-speculative and that the misspeculation signal is computed correctly with respect to the ISA semantics.

To implement erasure, we used a dynamic label that only marked entries between the head and tail of the ROB as valid; this enabled us to type-check a normal ROB since resetting the tail pointer upon misspeculation effectively "erased" misspeculated entries.

Cache. We implemented a blocking, direct-mapped L1 cache in SpecVerilog and labeled its interface so that all requests and responses were associated with some instruction. Responding to requests requires multiple cycles and so the cache maintains state associated with the current request. We also built a GhostMinion-like [3] module to store cache lines from speculative requests which were promoted to the original cache on instruction commit.

Discussion. Even with such a simple design, SpecVerilog forced us to implement essentially all of the mitigation mechanisms described in the original GhostMinion work: free-slotting, time-guarding, and leapfrogging. We did not need to add extra state or dynamic checks beyond those needed for the above mitigation techniques and SpecVerilog ensured that our they were implemented safely.

Branch Predictor. We built a standard 2-bit history predictor. To ensure this design was safe we had to insert a dynamic check that ignored speculative updates to its state. Alternatively, this check can be established as an assumed precondition on valid requests. In addition, we made a second version applying the GhostMinion methodology to allow speculative updates to predictor state.

Discussion. Branch predictors can be the targets of speculative fetch redirect vulnerabilities [74]; the usual method of defending against this is by delaying updates to predictor state. Our design represents this delay mitigation by forcing some dynamic check, either in the predictor or in an instantiating module.

Renaming Register File. Renaming register files are not typically vulnerable to transient execution vulnerabilities; however, they do mix speculative and committed state and their safety relies on invariants established by functional correctness. We modify an existing implementation so that it type-checks in SpecVerilog.

Discussion. This was the only module where SpecVerilog forced us to add extra state and/or dynamic checks to convince the solver that the implementation was secure. We needed to add explicit labels for name file replicas that are used to reset the architectural-to-physical name mappings upon misspeculation. This modification would only require a few hundred bits for a realistically sized implementation. In a traditional processor these bits would typically be tracked elsewhere (usually in the ROB), so this rename file overhead would not actually imply any extra storage overhead for the complete processor implementation.

Additionally, we had to change some of the logic that updated the list of free names; the original logic was safe but only due to invariants that could only be established with gate-level reasoning (e.g., a tool like GLIFT [113]). However, we were able to encode some invariants about valid usage of the rename file as input assumptions to reduce the number of redundant dynamic checks that would be established in the processor control logic.

4.6.2 Experience Report

While developing these case studies, we learned several key takeaways about designing secure hardware in SpecVerilog:

- SpecVerilog frequently forced us to fix potential vulnerabilities. These bugs included both forgetting to invalidate misspeculated data or metadata and also incorrectly handling interactions between different speculative requests.
- We did usually have to syntactically alter designs for them to be accepted by SpecVerilog, although this did not often change their functionality or require extra state.
- Some designs are only secure under certain assumptions. These assump-

tions often directly follow from functional correctness, and could be verified separately, either using traditional hardware verification or by establishing the necessary invariants via another hardware module. SpecVerilog required us to make those assumptions explicit as input assertions when typechecking our designs.

At a high level, many of the reported typechecking errors we encountered were subtle and it was difficult to know whether pre-existing code was insecure or if SpecVerilog was incorrectly rejecting a design due to imprecision; most of the time it turned out to be the former. Usually, we discovered this by re-writing the relevant logic from a blank slate, guided by the SpecVerilog type-checker; it was much easier to build a secure design than fix an insecure one.

Annotation Burden. Most of the required design effort is from explicitly defining and annotating labels. We wrote 21 lines of Z3 SMT constraints to specify the definitions of dynamic labels and erasure conditions across all example combined; these label definitions represents a one-time effort and can be reused in other designs. Neither SpecVerilog nor its predecessor, SecVerilog, have label inference, and therefore all registers and wires in the design must have their labels annotated by the programmer. While standard IFC inference algorithms [82] could be used to ease some of this burden, erasure labels and per-element labels [46] complicate this problem and would require further investigation. The other primary source of changes we needed to make to assist in type checking was translating some dynamic checks to use the "flows to" operator, instead of an equivalent logical formulation. Typically, these changes did not alter functionality in any way but allowed SpecVerilog to prove the safety of the design.

Register and Logic Overhead. We almost never needed to add extra registers to verify secure designs since the relevant valid bits or instruction identifiers were often necessary to implement a secure design. However, in a few instances we did need to add redundant dynamic checks that could lead to some unnecessary overhead in the final designs. This redundancy was caused by imprecision in the static analysis used by SpecVerilog to prove relationships between dynamic labels; improving the precision of this analysis could enable removing these redundant checks.

Compile Time Overhead. SpecVerilog imposed little compile-time overhead. All of our examples compile and type-check in less than one minute and most complete in less than one second, despite relying on an SMT solver. The vast majority of the compile time was spent in the solver and Table 4.1 shows those time for each case study. Even the most complex individual queries that relied on the Z3 theory of arrays took no more than ten seconds. When a design is insecure, Z3 provides a counterexample that violates the type-checking constraints.

4.7 Related Work

4.7.1 Architectural Mitigations

Since the discovery of Spectre and Meltdown, dozens of microarchitectural mitigation mechanisms have been proposed (e.g., [129, 4, 3, 127, 74, 97]). Xiong and Szefer [126] survey microarchitectural mitigation techniques. While some designs come with formal security conditions [74, 129], or even security proofs [130], none of these designs (to our knowledge) are formally checked for correctness at the RTL level. Most are implemented in the architecture simulator gem5 [18], which does not accurately capture timing behavior and does not describe synthesizable circuits.

SpecVerilog is an RTL-level language that can be used to implement and verify the security of many of these mitigation mechanisms. Most of these mitigations rely on a combination of delaying potentially leaky operations, rolling back speculative modifications, partitioning state, and taint tracking. SpecVerilog's information flow type system with erasure labels can be used to validate defenses using all of these techniques. However, some defenses leverage randomness [120, 123, 65]; SpecVerilog would likely consider them insecure since it cannot reason about probabilistic security.

Orthogonally, some security-centric architectures [131, 47, 128] include annotations that enable software to specify fine-grained protection of specific processor data. SpecVerilog could be used to check the security of speculative implementations of these architectures.

4.7.2 Secure Hardware Design Tools

In this work, we extend SecVerilog [139, 45, 46], one of a few information flow type systems for secure RTL design [71, 70, 47, 37]. While some of the above allow dependent security labels, none of them have been used to defend speculative security conditions. In addition to type systems, there are a number of other static analysis tools for secure hardware design.

GLIFT [113, 57] tracks information flow at the gate level and leverages properties of boolean logic for high precision. RTLLIFT [7] applies similar techniques but improves verification performance by working at the RTL level; both tools lack scalability due to the inherent complexity of the approach.

Clepsydra [6] and Xenon [115] are designed to check RTL designs for timing side-channels. Clepsydra verifies coarse-grained timing security polices such as constant-time execution and timing isolation. Xenon, an interactive tool for verifying constant-time execution, scales to more complex circuits, including in-order processors. Neither tool has been applied to security of transient execution.

The only static analysis tool that can verify processor speculation security properties is that of Fadiheh et al. [43], based on Unique Program Execution Checking. Unlike SpecVerilog, their tool requires significant manual proof effort from the user.

4.7.3 Information Flow Erasure Policies

Information flow erasure policies [29] were originally defined as part of a type system for software. However, Chong and Myers assume that a run-time monitor enforces erasure policies via dynamic clearing. Hunt and Sands [58] describe a type system that statically enforces erasure, although when to erase data is defined syntactically via scope rather than semantically. Stewart et al. [106] use dependent types to support erasure within data structures.

Our erasure labels expand upon these systems through semantic erasure conditions that specify *when* data should no longer be used, are statically enforceable, and that leverage dependent types to mix and reuse state across security levels.

4.7.4 Speculation-Secure Software

There have been many efforts to verify the security of *software* given various speculative hardware semantics [27, 50, 118, 25, 51, 81]. These all adopt abstract processor semantics and leakage models. However, unlike the semantics we assume in § 4.5.2, they often rely on more specific assumptions about processor behavior and do not incorporate time explicitly into their attacker models. This is a reasonable choice for these tools since the timing and speculative behavior of processors is unspecified by the ISA. However, explicitly timing-sensitive guarantees like Transient Noninterference and UPE provide more complete security and we believe should be the gold standard for secure processor implementations. Guarnieri et al. [51] have proposed hardware–software contracts to bridge this abstraction gap. Transient Noninterference prevents leakage of architecturally accessed state, corresponding to their $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contract.

4.8 Discussion & Conclusion

SpecVerilog enables the verification of speculative security guarantees of RTL designs via the incorporation of erasure labels into an IFC type system. Here, we present a final discussion of some of the benefits, potential and limitations of SpecVerilog with respect to secure processor design and verification.

Processor Verification. Verification of Transient Noninterference or similar conditions fundamentally requires reasoning about functional correctness since the definition of speculation is relative to the ISA-specified behavior. SpecVerilog provides a clean divide between functional verification and security analysis via erasure conditions. Erasure conditions abstract *when* misspeculation occurs without having to reason about *why* it occurs. In this way, traditional processor verification techniques and tools can be used to prove the assumptions needed by erasure labels (such as "the oldest instruction always commits") while SpecVerilog can handle vulnerability checking.

Generalizing Speculation. We have described a single methodology for OoO processor labeling and speculation, but SpecVerilog is not limited to next-instruction prediction or to the ROB labels we chose. SpecVerilog can be used to check other erasure conditions that incorporate multiple sources of speculation, such as branch prediction and value prediction; however, we have yet to implement and type-check such a design. A key subtlety in this case is that not all erasure labels are guaranteed to be ordered, so dynamic scheduling is more difficult to implement securely. Furthermore, labels could be assigned *per branch* rather than *per instruction* to achieve more precise reasoning about potential vulnerabilities; although this requires reasoning about the separation between "front-end" speculation that applies to every instruction fetch, and branch speculation.

Erasure Expressivity. Modern processors also do not necessarily propagate control signals globally in a single cycle, due to latency and power constraints. Additionally, some misspeculation clean-up implementations take multiple cycles and thus do not satisfy our definition of end-to-end erasure. SpecVerilog cannot represent the propagation or resolution of delayed misspeculation: erasure must happen synchronously and immediately. To support these feature, we believe SpecVerilog could incorporate explicit temporal logic operators (e.g., “next”) into erasure conditions.

CHAPTER 5

CONCLUSION

This dissertation has presented three different efforts to address timing channels and the construction of correct and secure hardware. We have shown how IFC can be used to define a contract between hardware and software that can prevent timing channels without completely exposing processor implementation details. We have also demonstrated two techniques for building processors that refine such a secure contract: using a high-level HDL to construct processors whose behavior can easily be related to the software-visible semantics; and leveraging IFC types to securely implement efficient processor modules that are invulnerable to transient execution attacks.

Nevertheless, this dissertation addresses only a small part of the entire hardware–software stack and there remain plentiful research opportunities to expand upon this work. Not only can the solutions and ideas presented here be expanded upon to improve their completeness and utility, but we can also begin to reimagine the underlying design assumptions of the entire hardware–software stack in the context of a new security contract and new hardware implementations. To conclude, we highlight some of these high-level takeaways and implications of this dissertation's contributions.

5.1 Contracts for Timing-Channel Security

Both the ISA presented in Chapter 2 and the speculative security condition presented in Chapter 4 adopt IFC. This is in contrast to traditional ISAs, which only define how instructions should behave, given the current processor state (e.g., "a memory access to a privileged address should cause an exception if the process status register is set to user mode"). These security conditions effectively close the side-channel loopholes, or at least makes their leakage explicit, as opposed to completely undefined.

Going forward, we believe there are two key questions for exploring these new kinds of hardware–software contracts:

1. What features do these contracts still need to support practical systems and what vulnerabilities might these new features expose?
2. How can low-level software and operating systems leverage the ISA's stronger security guarantees to improve performance or provide finer-grained isolation?

5.2 Secure ISA Features and Design

Chapter 2 describes some of the ISA features we need to support realistic systems software. In particular, it addresses key replacements for traditional system call instructions necessary to implement the privileged–unprivileged software boundary and the labeling instructions necessary for memory management. In fact, as a demonstration, we have ported some of the core features of the freeRTOS [13] operating system using our IFC ISA[1]. One property our demonstration enforces (by using the appropriate labels) is that all access to I/O must go through the OS network stack and is invulnerable to confused deputy [53] attacks.

[1] The code can be found at `https://github.coecis.cornell.edu/HyperFlow/freeRTOS-riscv/tree/labels`

However, a significant amount of engineering and security design remains to securely support other ISA features needed to build a fully featured OS: microarchitectural flushing/performance management, interrupt handling, multicore architectures, synchronization primitives, and virtualization. Specifically, *label virtualization* is going to be a key feature for realistic systems; the hardware can only efficiently represent a small number of IFC labels concurrently, and it is likely to be *much smaller* than the number of security domains real systems need to represent.

Supporting virtualization for most hardware components is not likely to pose too difficult a challenge; a trusted component of the operating system can manage allocation of labeled hardware resources (e.g., physical memory). Label virtualization itself is likely a much less trivial (though certainly possible!) addition to these ISA, as it adds a further complication to the "label of labels" problem discussed in prior chapters and in the dynamic IFC literature. With virtualized labels, the hardware needs to maintain structures that translate software labels to those supported natively by the ISA checks, while trusted software needs to create and maintain those mappings.

Another direction worth investigating is how IFC–defined ISAs relate to capability architectures like CHERI [125]. CHERI solves some of the aforementioned challenges with tagged memory architectures, but also provides a very different set of security guarantees.

Finally, many efforts to enable constant-time programming [25, 118] in the face of Spectre assume that the ISA supports speculation fences to limit the influence of speculative execution. It would be interesting to incorporate such primitives into our IFC–inspired ISA to provide even more flexibility to software. In this way, the influence of timing channels could be limited either based on security annotations, expected functional behavior, or some combination thereof.

5.3 Opportunities for Secure Software Stacks

Although speculative security conditions like the one described in Chapter 4 seek just to reclaim the sane hardware semantics we always thought we had, our ISA from Chapter 2 inspires rethinking how to build the entire hardware–software stack.

With strong, hardware-enforced physical memory protection (from both direct access and side-channels), we can start to rethink decades-old assumptions about how software security is enforced. For instance, memory management is traditionally fully intertwined with address translation; in order to benefit from memory protection, software has use virtual memory. For many applications, virtual memory can come at a significant price; a single memory access can translate into five on a TLB miss, or up to 25 when running a virtual machine. We explored the potential benefits of directly accessing *physical memory* and found that certain applications are likely to significantly benefit, while enabling much simpler hardware designs [132]. Other researchers have explored separating translation and protection at the software level [109]; however, there remains many opportunities to leverage novel hardware memory protection mechanisms and architectures.

Another opportunity is improving the security of hypervisors and operating systems, and enabling fast and efficient control transfer for system call–heavy code. Typically, context switching between processes or switching between privileged and user modes is costly since hardware structures like the cache and TLB must be explicitly flushed to prevent timing channels, and address spaces must be switched as well. Our ISA highlights the opportunity that architectures with stronger, *microarchitecture-level*, security guarantees afford; they can support efficient and safe security domain switches.

In particular, such an ISA can enable efficient microkernel operating systems [35] that split the kernel into separate components for each feature. We can use different IFC labels for different components, guaranteeing that each has the minimal authority necessary to execute, while still providing efficient communication between applications or components. Our ISA naturally lends OS and application development to follow the principle of least privilege, potentially without sacrificing any overhead for the increased number of security domains. Even in traditional monolithic kernels, applications could make use of security labels to implement efficient user level sandboxing.

5.4 Designing and Verifying Secure Hardware

While this thesis has made significant headway into verifying the security and correctness of processor implementations, our contributions are only the beginning. The ISA in Chapter 2 requires the processor to defend a condition called *microarchitectural noninterference*. The efforts in Chapters 3 and 4 partially address how to formally guarantee that a hardware implementation satisfied this condition. To completely verify the correctness and security of real processor implementations, we need to combine the functional guarantees of PDL with the security guarantees of SpecVerilog.

These two tools are in this way complementary; with the addition of IFC labels, PDL can be used to verify that a processor design implements the expected behaviors of an IFC ISA; however it is still slightly unclear how one would prove microarchitectural noninterference of the generated circuit. We imagine this can be done in three parts: first, providing stronger guarantees about the logical *schedule* that the PDL compiler generates; second, proving security properties about the linked RTL libraries used to implement locks and external modules; and finally, guaranteeing that the composition of PDL with secure libraries results in a secure overall design. Using this methodology, we could potentially augment PDL and SpecVerilog to build provably secure out-of-order processors.

APPENDIX A

APPENDIX TO CHAPTER 2

A.1 Definitions

We begin by restating some definitions from §2.2. Here, l_i, i and c represent elements of Lbl.

$$(i,c)^{\rightarrow} \triangleq c$$

$$(i,c)^{\leftarrow} \triangleq i$$

$$l_1 \sqsubseteq l_2 \overset{\triangle}{\Leftrightarrow} (l_1^{\leftarrow} \sqsubseteq l_2^{\leftarrow})(l_2^{\rightarrow} \sqsubseteq l_1^{\rightarrow})$$

$$l_1 \sqcup l_2 \triangleq ((l_2^{\rightarrow} \sqcup l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcap l_2^{\leftarrow}))$$

$$l_1 \sqcap l_2 \triangleq ((l_2^{\rightarrow} \sqcap l_2^{\rightarrow}), (l_1^{\leftarrow} \sqcup l_2^{\leftarrow}))$$

$$\bowtie(i,c) \triangleq (c,i)$$

Call stack validity can be formalized as:

$$CST = ((pc_{l0}, pc_{v0}, t_{l0}, t_{v0})...(pc_{ln}, pc_{vn}, t_{ln}, t_{vn}))$$

$$ISVALID(CST) \triangleq \forall i \in (0, n-1).(pc_{li} \sqsubseteq pc_{li+1} \texttt{ or } pc_{li+1} \sqsubseteq pc_{li})$$

$$ISVALID(CST, pc_l) \triangleq ISVALID(CST) \texttt{ and } (pc_l \sqsubseteq pc_{l0} \texttt{ or } pc_{l0} \sqsubseteq pc_l \texttt{ or } CST = \emptyset)$$

ISVALID (*CST*,pc_l) can be read as *"CST is valid for* pc_l*"*, meaning that the call stack itself has ordered entries and the pc_l is ordered with respect to the head of the call stack. This restriction maintains the idea that call gates cannot be freely mixed with moving the pc_l around via `raiselbl`; call gates need to reflect the actual sequence of control transfer agreed upon when the gates are established via `dwncall` or `upcall`.

$$\boxed{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M', L', pc', t_l\rangle}$$

$$\frac{\neg INUPCALL \qquad L(M(pc_v)) \not\sqsubseteq pc_l \vee pc_l \not\sqsubseteq \text{⧖}(pc_l)}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{all\_pc\_error}$$

$$\frac{\neg INUPCALL \qquad L(r_{s1}) \sqcup L(r_{s2}) \not\sqsubseteq pc_l}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{branch\_error}$$

$$\frac{\neg INUPCALL \qquad l' = \gamma(R_{s1}) \qquad (L(r_{s1}) \not\sqsubseteq pc_l) \vee (l' \not\sqsubseteq \text{⧖}(pc_l))}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{uplbl\_arg\_error}$$

$$\frac{\neg INUPCALL \qquad l' = \gamma(R_{s1}) \qquad (L(r_{s1}) \not\sqsubseteq pc_l) \vee (l \not\sqsubseteq \text{⧖}(pc_l))}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{relbl\_arg\_error}$$

$$\frac{\neg INUPCALL \qquad L(r_{s1}) \sqcup L(r_{s2}) \not\sqsubseteq pc_l}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{raiselbl\_arg\_error}$$

$$\frac{\neg INUPCALL \qquad L(r_{s1}) \sqcup L(r_{s2}) \sqcup L(r_{s3}) \sqcup L(r_d) \not\sqsubseteq pc_l}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{upcall\_arg\_error}$$

$$\frac{\begin{array}{c}(GR(R_{s1}) = \emptyset) \qquad \vee \\ (\emptyset \neq CST[head]) \vee (L(r_{s1}) \not\sqsubseteq pc_l) \vee (pc'_l \sqcup t'_l \not\sqsubset pc_l)\end{array}}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (errorpc, pc_l), t_l\rangle}\ \textsc{dwncall\_arg\_error}$$

$$\frac{\neg INUPCALL}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, (pc_v + 4, pc_l), t_l\rangle}\ \textsc{other\_error}$$

$$\frac{INUPCALL}{GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST, M, L, pc, t_l\rangle}\ \textsc{upret\_error}$$

Figure A.1: Operational semantics for error handling rules given a call-gate registry *GR*.

Definition A.1 (Call Stack Validity)**.**

A call stack, CST, is valid with respect to the current pc_l if it represents an uninterrupted sequence of call gate calls and returns.

A configuration is *valid* if it has a valid pc_l, t_l and CST:

Definition A.2 (Configuration Validity)**.**

A configuration, C, is valid iff:

- $pc_l \sqsubseteq ⧖(pc_l)$,
- $pc_l \sqsubseteq t_l$,
- $t_l \sqsubseteq ⧖(t_l)$,
- *and CST is valid for pc_l.*

This validity condition captures the notion that the pc_l and t_l remain *uncompromised*, in addition to call stack validity.

Notation. Two low-equivalent configurations C_1 and C_2 contain state such as pc or t_l which may vary between them or be the same. When the values must be equivalent in both configurations we omit subscripts. When they may differ, we use subscripts to denote to which configuration they belong.

A.2 Proofs

Based on our attacker definition, all secret and untrusted labels are compromised.

Lemma A.1. *For $\mathcal{S}$ and $\mathcal{U}$ sets induced by an attacker:*

$$\forall l \in Lbl.\, l \in \mathcal{S} \cap \mathcal{U} \implies l \not\sqsubseteq ⧖(l)$$

Proof. Recall that attacker-induced sets are defined as upward-closed sets with a minimum confidentiality c_A and a maximum integrity i_A. The label (i_A, c_A) is itself compromised and by Definition 2.1, we have $(i_A, c_A) \not\sqsubseteq (c_A, i_A)$.

The upward closure property implies $\forall l \in \mathcal{S} \cap \mathcal{U}.(i_A, c_A) \sqsubseteq l$. We show that $l \sqsubseteq ⧖(l) \implies (i_A, c_A) \sqsubseteq (c_A, i_A)$ and by contrapositive, l must be compromised. We represent l explicitly as (l_i, l_c).

$$\begin{aligned}
& l \sqsubseteq ⧖(l) \equiv (l_i, l_c) \sqsubseteq (l_c, l_i) \\
& \equiv (l_i \sqsubseteq l_c) \wedge (l_i \sqsubseteq l_c) && \textit{(By definition of} \sqsubseteq \textit{)} \\
& \implies i_A \sqsubseteq l_c && \textit{(By } (i_A, c_A) \sqsubseteq l \implies (i_A \sqsubseteq l_i)\textit{)} \\
& \implies i_A \sqsubseteq c_A && \textit{(By } (i_A, c_A) \sqsubseteq l \implies (l_c \sqsubseteq c_A)\textit{)} \\
& \equiv (i_A \sqsubseteq c_A) \wedge (i_A \sqsubseteq c_A) \\
& \equiv (i_A, c_A) \sqsubseteq (c_A, i_A) && \textit{(By definition of} \sqsubseteq \textit{)}
\end{aligned}$$

□

Lemma A.2 (No Compromised Call Stack Entries)**.** *No valid call stack will contain any entries whose pc_l is compromised.*

Proof. Any instruction may only execute successfully if the pc_l itself is *valid*. The pc_l validity condition requires that it is an uncompromised label. Therefore, any time an `upcall` or `dwncall` succeeds and places an entry onto the stack, the label of that entry (the current pc_l) must be valid and uncompromised. □

Configuration validity is preserved under our instruction semantics.

Lemma A.3 (Validity of Configurations). *If C is a valid configuration and $C \longrightarrow^{*} C'$, configuration C' is also valid.*

Proof. We show that no instruction will step to an *invalid* configuration and by induction, this lemma holds. Instructions `upcall`, `dwncall`, and `raiselbl` check that the *ISVALID* (pc_l,t_l) condition holds for the new pc_l and t_l. If the new labels would be invalid, then the current (and valid) labels are retained. Furthermore, when the `upret-done` or `dwnret` instructions execute, by Lemma A.2, the resulting pc_l and t_l will be uncompromised (and a similar argument fulfills the remainder of the *ISVALID* (pc_l,t_l) condition).

Therefore, the only validity condition we must check is call stack validity. The only instructions which change *CST* are `upcall/upret-done` and `dwncall/dwnret`. Instructions `upcall` and `dwncall` explicitly require that the current pc_l is ordered with respect to the new pc (pc_l'). For `upcall`, $pc_l \sqsubseteq pc_l'$ and for `dwncall` $pc_l' \sqsubseteq pc_l$. Since the new "head" of the call stack will have an entry label of pc_l, validity is preserved. Inductively, any `dwnret` or `upret-done` instruction will also preserve call stack validity since the pc_l of *CST*'s first entry must either be ordered with respect to *CST*'s second entry or *CST* has only one entry.

The `raiselbl` instruction is prohibited from changing pc_l or t_l whenever *CST* is nonempty; therefore the only other instruction which changes the pc_l cannot violate the ordering relationship between *CST* [head] and pc_l.

In particular, the `dwncall` and `upcall` restrictions require that $pc_l{}'$ and pc_l are ordered, and then push pc_l onto the call stack. The `raiselbl` restriction ensures that pc_l can only be raised when already in an upcall or when the call stack is empty. In the latter case, validity is trivially preserved. In the former, the `upcall` restrictions ensure that the label of $CST[head] \sqsubseteq pc_l$, which means raising pc_l will maintain that ordering. □

Lemma A.4 (Call Stack Upcall History). *For any two valid configurations C_1 and C_2, If $C_1 =_{\mathcal{L}} C_2$ and $pc_l \in \mathcal{L}$, then either both configurations are in an* `upcall` *region, or neither is.*

Proof. If either one of the call stacks is empty, then, by Definition 2.6, the other must also be empty or all of its entries have a high pc label (which we'll denote pc_{cs}). If both are empty, then neither is in an `upcall` region.

If only one is empty, a `dwncall` from high to low must have generated the head entry of that stack. An `upcall` could not have executed successfully since `upcall` requires that the pc label of the executing context flows to the new pc label. Since pc_l is low and pc_{cs} is high, $pc_{cs} \not\sqsubseteq pc_l$ (by upward closure of $\mathcal{H}$). Furthermore, since a `dwncall` can only be made when the call stack is empty, there could have been no prior `upcall` executed without a corresponding return. In this case, one configuration is inside a `dwncall` region and the other is inside no call-gate region.

If both call stacks are nonempty, let $CST_i[head] = ((pc_{v_i}, pc_{l_i}), (t_{vi}, t_{li}))$. If $pc_{l_i} \in \mathcal{H}$ then, by the same argument as before, both configurations must be inside a `dwncall` region but not inside an `upcall` region. If $pc_{l_i} \in \mathcal{L}$, then $pc_{l1} = pc_{l2}$. This means that the most recent call gate instruction was either an `upcall` or a `dwncall`, but it must be the same for both configurations. If it was a `dwncall`, then $CST_i[tail] = \emptyset$ and neither configuration is in an `upcall` region. □

Lemma A.5 (Low Equivalence of Error Rules in Low Contexts)**.** *For any set of low labels, $\mathcal{L}$, and any two valid configurations C_1, C_2 and $C_1 \longrightarrow C_1'$ by applying an* ERROR *rule, and* **insn** $\not\equiv$ DWNLBL,DWNCALL,DWNRET

$$(C_1 =_{\mathcal{L}} C_2) \land (pc_l \in \mathcal{L}) \implies (C_1' =_{\mathcal{L}} C_2')$$

Proof. First, we show that this lemma holds for all of the error rules in Figure A.1. For any check in the form of $L(var) \sqsubseteq pc_l$[1], we can guarantee that the result of that check is the same for low-equivalent configurations when $pc_l \in \mathcal{L}$.

If $L_1(var) \in \mathcal{L}$, then by the low equivalence of configurations, $L_2(var) = L_1(var)$. The evaluation of $L(var) \sqsubseteq pc_l$ therefore results in the same outcome in both configurations. If $L_1(var) \in \mathcal{H}$, then by the low equivalence of configurations, $L_2(var) \in \mathcal{H}$. In both cases, the check will fail since $\mathcal{H}$ is upward closed and $pc_l \in \mathcal{L}$. By Lemma A.4, either both configurations are in an upcall region, or neither is. Therefore the `INUPCALL` check has the same result for C_1 and C_2. All of the rules in Figure A.1 contain only checks of the form $L(var) \sqsubseteq pc_l$[2] and `INUPCALL` checks. Since the outcome of the label checking must be the same in both configurations, updating the pc_v to **errorpc** does not violate low equivalence in the resulting configurations.

We now consider the label checks specific to various instructions.

Branch/Jump:

These label checks only contains checks of the form $L(var) \sqsubseteq pc_l$. By our earlier argument, the BRANCH_ERROR rule from Figure A.1 always results in low-equivalent configurations.

[1]Or of the form $L(var) \not\sqsubseteq pc_l$

[2]Or $L(var) \sqsubseteq \between(pc_l)$ which does not change our reasoning.

Compute/Load/Store:

The checks for these instructions ensure that all of the labels of the operands flow to the label of the destination. Wlog. we can analyze a check of the form $L(v_1) \sqsubseteq L(v_2)$.

Let us consider the four possible cases that C_1 can evaluate:

Case 1) $L(v_1), L(v_2) \in \mathcal{L}$

In this case, by the low equivalence of configurations both $L_1(v_1) = L_2(v_1)$ and $L_1(v_2) = L_2(v_2)$ and so the success or failure of the label check is the same for both configurations.

Case 2) $L(v_1) \in \mathcal{L}, L(v_2) \in \mathcal{H}$

In this case, $L_1(v_1) = L_2(v_1)$ and $L(v_2)$ is high in both configurations. It is possible that this label check fails in C_1 but succeeds in C_2 or vice versa.

Case 3) $L(v_1) \in \mathcal{H}, L(v_2) \in \mathcal{L}$

In this case the check will fail in both configurations, since $\mathcal{H}$ is upward closed and $L(v_1)$ is high in both configurations.

Case 4) $L(v_1) \in \mathcal{H}, L(v_2) \in \mathcal{H}$

Like case 2, this label check could fail in only one of the configurations, since not all high values flow to one another.

In cases 1 and 3, both configurations will fail (this lemma considers only the cases where C_1 fails label checking) and therefore neither will update memory or label state, which implies that low equivalence will be preserved. In cases 2 and 4, the $L(v_2)$ is a high label. This means that even if the operation succeeds in one configuration and not the other, the changes to memory happen only on elements with high labels. Therefore, memory low equivalence is preserved.

Uplabel:

$$UPLBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqsubseteq l') \wedge (l' \sqsubseteq \bowtie(pc_l))$$

As argued above, if the UPLBL_ARG_ERROR rule is matched, low equivalence is preserved since both configurations will fail this check and step to the error program counter. Let us now consider the case where an UPLBL instruction fails label checking but the UPLBL_ARG_ERROR rule does not apply. In this case, $L_1(r_{s1}) \sqsubseteq pc_l$ and is therefore low. This implies that R_{s1} is equal in both configurations and therefore $l'_1 = l'_2$. The $l' \sqsubseteq \bowtie(pc_l)$ component of the label check will therefore result in the same value for both configurations. Lastly, we consider the requirement: $pc_l \sqsubseteq l \sqsubseteq l'$.

Again there are 4 cases to analyze:

Case 1) $l, l' \in \mathcal{L}$

In this case, $l_1 = l_2$ by low equivalence, so the label-checking result will be the same in both configurations.

Case 2) $l' \in \mathcal{L}, l \in \mathcal{H}$

In this case, both l_1 and l_2 are in $\mathcal{H}$ by low equivalence and by upward closure of $\mathcal{H}$, $l \sqsubseteq l'$ will fail in both configurations.

Case 3) $l \in \mathcal{L}, l' \in \mathcal{H}$

By low equivalence, we have $l_1 = l_2$ and both configurations compute the same label-check result.

Case 4) $l, l' \in \mathcal{H}$

In this case, l_1 and l_2 are both in the high domain of L. Even if this label check succeeds in one configuration, it will only change the value of a high label to another high label. This does not affect low equivalence so $L'_1 =_{\mathcal{L}} L'_2$.

RaiseLbl:

If the RAISELBL_ARG_ERROR rule is not matched then the pc_l, pc'_l and t'_l must be the same in both configurations. The only two cases to consider are based on the current value of t_l. If $t_l \in \mathcal{L}$, then both configurations will fail the label check and low equivalence is preserved. If $t_l \in \mathcal{H}$, then even if only one configuration succeeds $t'_{l1} =_{\mathcal{L}} t'_{l2}$ since both have high labels.

Upcall:

If the UPCALL_ARG_ERROR rule is not matched, then R_d, $endpc$, pc'_l and t'_l will be the same in both configurations. Since both configurations are valid we know that $pc_l \sqsubseteq t_l$ for both configurations. By the earlier argument, both configurations will resolve the $pc_l \sqsubseteq pc'_l \sqsubseteq t'_l$ check the same way. If true then that implies the restriction $t_l \sqsubseteq t'_l$ is true for both configurations as well. Therefore, if either configuration passes the label checking process, then both configurations must pass it. By contrapositive, if either configuration fails then both will fail label checking.

Upret:

By Lemma A.4, both configurations will either be in an `upcall` region or not, meaning that this instruction either causes an error or does not in both configurations. In the case where an error occurs, no state is updated, and therefore the resulting configurations are low-equivalent.

□

Lemma A.6 (High PC Call Stack Noninterference)**.** *For any two valid configurations C_1 and C_2, where neither configuration executes an* UPRET-DONE *step.*

$$pc_l \in \mathcal{H}, C_1 =_{\mathcal{L}} C_2, C_i \longrightarrow C_i' \implies CST_1' =_{\mathcal{L}} CST_2'$$

Proof. By the definition of call stack equivalence, both call stacks begin with equivalent prefixes of entries with low pc_l, or neither contains any entries with a low pc_l.

In the former case, the heads of these stacks must contain low-equivalent entries, where the pc label of these entries ($pc_{l_{entry}}$) is $\in \mathcal{L}$. In both configurations, $pc_l \in \mathcal{H}$ and therefore this entry must have been produced by an `upcall` instruction. While in an upcall region, the only stack-modifying instruction that can successfully return is UPRET-DONE. Since we are excluding that rule in this lemma, the resulting call stacks in this case must always be low-equivalent, because they cannot be modified.

In the latter case, call stacks produced by these configurations are low-equivalent. Popping off entries from the stack cannot introduce entries with low pc_l and pushing on entries uses the current pc_l, which is high. Therefore any call-stack modifying instruction will result in low-equivalent call stacks.

□

This also leads to a useful corollary: if two configurations have a high pc_l and neither is inside of a upcall with a low entry label, their call stacks are always noninterfering. As noted in Lemma A.6, if the entries on the call stack all have high pc_l, then no operation will place a low-labeled entry on the stack. Therefore call-stack low equivalence is preserved.

Corollary A.1. *For any two valid configurations C_1 and C_2, where $CST_i[head] = \emptyset$ or $CST_i[head] = ((pc_{vi}, pc_{li}), (t_{vi}, t_{li}))$ and $pc_{li} \in \mathcal{H}$.*

$$pc_l \in \mathcal{H}, C_1 =_{\mathcal{L}} C_2, C_i \longrightarrow C'_i \implies CST'_1 =_{\mathcal{L}} CST'_2$$

Now we show that configurations with high pc_l cannot make any low-visible changes to state without the use of downgrading instructions or call gates. We then use this result to prove the more general noninterference of high pcs lemma.

Lemma A.7 (Noninterference of High PCs Modulo Call Gates)**.** *For any two valid configurations, C_1 and C_2, if* $\mathsf{insn} \not\equiv$ DWNCALL,DWNRET,UPCALL,UPRET-DONE

$$(pc_l \in \mathcal{H}) \wedge (C_i \longrightarrow C'_i) \wedge (C_1 =_{\mathcal{L}} C_2)$$
$$\implies \langle CST'_1, M'_1, L'_1, pc'_1, t'_{l1} \rangle =_{\mathcal{L}} \langle CST'_2, M'_2, L'_2, pc'_2, t'_{l2} \rangle$$

Proof. First, we note that C_1 and C_2 may or *may not* be executing the same instruction; therefore, this proof must rely on reasoning about the visible effects of any given single configuration.

Label Mappings:

We show that $L_i =_{\mathcal{L}} L'_i$ to prove noninterference of label mappings since $(L_1 =_{\mathcal{L}} L_2) \wedge (L_i =_{\mathcal{L}} L'_i) \implies L'_1 =_{\mathcal{L}} L'_2$. For each instruction we must show that the low and high domains of L do not change, and that low labels are not modified at all. To show the former, we need to show that any label being changed remains in the same quadrant of the lattice after modification. In general, if any label-modifying instruction fails, low equivalence is preserved since L cannot change.

First, we consider the `uplbl` instruction, which sets some location's label from l to l'. Recall that the primary restriction for this instruction is:

$$UPLBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqsubseteq l') \wedge (l' \sqsubseteq ⧖(pc_l))$$

There are four cases to consider here: $l \in \mathcal{L}$; $\mathcal{H} = \mathcal{S} \wedge l \in \mathcal{S} \cap \mathcal{T}$; $\mathcal{H} = \mathcal{U} \wedge l \in \mathcal{P} \cap \mathcal{U}$; $l \in \mathcal{S} \cap \mathcal{U}$.

Case 1) $l \in \mathcal{L}$

UPLBL requires $pc_l \sqsubseteq l$, so this case cannot succeed.

Case 2) $\mathcal{H} = \mathcal{S} \wedge l \in \mathcal{S} \cap \mathcal{T}$

UPLBL requires $l \sqsubseteq l' \wedge l' \sqsubseteq ⧖(pc_l)$. We show by contradiction that if $l' \in \mathcal{S} \cap \mathcal{U}$ and $pc_l \in \mathcal{S}$, this instruction cannot succeed. (And consequently, $l' \in \mathcal{S} \cap \mathcal{T}$ when the instruction does succeed.)

$$\begin{aligned}
&\textit{Assume: } l' \sqsubseteq ⧖(pc_l) \\
&\equiv (l'^{\leftarrow} \sqsubseteq pc_l^{\rightarrow}) \wedge (pc_l^{\leftarrow} \sqsubseteq l'^{\rightarrow}) \quad (\textit{By definition}) \\
&\implies (A^{\leftarrow} \sqsubseteq pc_l^{\rightarrow}) \qquad (l' \in \mathcal{U} \implies A^{\leftarrow} \sqsubseteq l'^{\leftarrow}) \\
&\implies (A^{\leftarrow} \sqsubseteq A^{\rightarrow}) \quad (pc_l \in \mathcal{S} \implies pc_l^{\rightarrow} \sqsubseteq A^{\rightarrow}) \\
&\equiv i_A \sqsubseteq c_A \equiv (i_A, c_A) \sqsubseteq ⧖((i_A, c_A))
\end{aligned}$$

This assumption contradicts our definition of attacker, so we have $l' \not\sqsubseteq ⧖(pc_l)$. In conclusion, if $l' \in \mathcal{S} \cap \mathcal{U}$, the instruction will not succeed.

In the case where the instruction succeeds, $l' \in \mathcal{S} \cap \mathcal{T}$.

Case 3) $\mathcal{H} = \mathcal{U} \wedge l \in \mathcal{P} \cap \mathcal{U}$

This follows the same logic as in case 2: we show by contradiction that, if $l' \in \mathcal{S} \cap \mathcal{U}$, and $pc_l \in \mathcal{U}$, then $l' \not\sqsubseteq ⧖(pc_l)$. When the instruction succeeds, $l' \in \mathcal{P} \cap \mathcal{U}$.

Case 4) $l \in \mathcal{S} \cap \mathcal{U}$

Since $l \sqsubseteq l'$, $l' \in \mathcal{S} \cap \mathcal{U}$.

Next we consider the `dwnlbl` instruction, which modifies the label of some location ($l = L(r_d)$) to some new value ($l' = \gamma(R_{s1})$). The primary restriction on executing this instruction is:

$$RELBL(pc_l, l, l') \triangleq (pc_l \sqsubseteq l \sqcap l') \wedge (l \sqsubseteq ⧖(l)) \wedge (l' \sqsubseteq ⧖(pc_l))$$

Again, there are 4 cases to consider: $l \in \mathcal{L}$; $\mathcal{H} = \mathcal{S}, l \in \mathcal{S} \cap \mathcal{T}$; $\mathcal{H} = \mathcal{U}, l \in \mathcal{P} \cap \mathcal{U}$; $l \in \mathcal{S} \cap \mathcal{U}$.

Case 1) $l \in \mathcal{L}$

RELBL requires $pc_l \sqsubseteq l \sqcap l'$, so this case cannot succeed.

Case 2) $\mathcal{H} = \mathcal{S}, l \in \mathcal{S} \cap \mathcal{T}$

If the instruction succeeds, $l' \notin \mathcal{P}$ since *RELBL* requires $pc_l \sqsubseteq l \sqcap l'$. Let us assume $l' \in \mathcal{S} \cap \mathcal{U}$. In that case, by the same reasoning as for `uplbl`, $l' \not\sqsubseteq ⧖(pc_l)$. Therefore, if the label check passes in this case, $l' \in \mathcal{S} \cap \mathcal{T}$.

Case 3) $\mathcal{H} = \mathcal{U}, l \in \mathcal{P} \cap \mathcal{U}$

This case has exactly the same reasoning as case 2. If label checks succeed, then $l' \in \mathcal{P} \cap \mathcal{U}$.

Case 4) $l \in \mathcal{S} \cap \mathcal{U}$

By Lemma A.1, $l \not\sqsubseteq \bigtimes(l)$, and therefore this case will fail the label check and not modify L.

Memories:

For all of the instructions which write to M, the pc_l must flow to the label of the modified memory location. Since no label modification instructions change the low domains of memory this implies that neither configuration can make low visible modifications to memory: $(pc_l \in \mathcal{H}) \wedge (M_1 =_{\mathcal{L}} M_2) \wedge (L_i \approx L'_i) \implies M'_1 =_{\mathcal{L}} M'_2$

Program Counters and PC/Time Labels:

First we consider the `raiselbl` instruction. It requires $pc_l \sqsubseteq pc'_l$ and $t_l \sqsubseteq t'_l$, ensuring that pc'_l and t'_l are also high labels, and therefore pc'_v will be low-equivalent across the configurations.

If the executing instruction fails for any reason (thus triggering the ERROR rule), the pc_l and t_l remain high and no other state is modified. □

Theorem 2.1 (Noninterference Modulo Downgrading and Call Gates)**.** *For any two valid configurations, C_1 and C_2 and any low set of labels, $\mathcal{L}$, if* $\mathsf{insn} \not\equiv$ DWNLBL,DWNRET,UPCALL,UPRET-DONE,DWNCALL

$$C_i \longrightarrow^* C_i^* \wedge C_1 =_{\mathcal{L}} C_2 \implies C_1^* =_{\mathcal{L}} C_2^*$$

Proof. We prove this by structural induction over the $\longrightarrow$ operator: $C_i \longrightarrow C'_i$. By Property 2.2, we know that μ and t are noninterfering, such that $\mu'_1 =_{\mathcal{L}} \mu'_2$ and

$t'_{v1} =_{\mathcal{L}} t'_{v2}$. When the STALL rule is applied, no architectural state changes and therefore $C'_1 =_{\mathcal{L}} C'_2$.

Therefore, we now only need to consider the $\longrightarrow_{\mathcal{A}}$ function and the configuration it produces.

As a reminder:

$$GR \vdash \langle CST, M, L, pc, t\rangle \longrightarrow_{\mathcal{A}} \langle CST', M', L', pc', t'_l\rangle$$

We show that

$$\langle CST'_1, M'_1, L'_1, pc'_1, t'_{l1}\rangle =_{\mathcal{L}} \langle CST'_2, M'_2, L'_2, pc'_2, t'_{l2}\rangle$$

holds for all possible applications of $\longrightarrow_{\mathcal{A}}$.

By Lemma A.7, when $pc_l \in \mathcal{H}$

$$\langle CST'_1, M'_1, L'_1, pc'_1, t'_{l1}\rangle =_{\mathcal{L}} \langle CST'_2, M'_2, L'_2, pc'_2, t'_{l2}\rangle.$$

Now we consider the case when $pc_l \in \mathcal{L}$. By Lemma A.5, we know that in situations where one or more of the configurations fails label checking, low equivalence of configurations is preserved. We now consider only scenarios where both configurations pass label checking.

Both configurations must be executing the same instruction (by the ALL_PC rule which ensures that the instruction itself is located in low-labeled memory).

Label Mappings:

Unlike in Lemma A.7 we can allow labels to change their current quadrant, as long as the change results in low-equivalent label maps. Again, UPLBL is the only instruction which modifies L. This instruction updates label $l \equiv L(r_d)$ to $l' \equiv \gamma(R_{s1})$ and requires $l \sqsubseteq l'$.

r_d and r_{s1} must both have the same respective value across configurations since they are part of the instruction.

$l'_1 = l'_2$ since $L(R_{s1}) \sqsubseteq pc_l \implies R_{s1}$ has the same value in both configurations.

The three possible cases are: $l, l' \in \mathcal{L}$; $l \in \mathcal{L}$ and $l' \in \mathcal{H}$; and $l, l' \in \mathcal{H}$. Since there is only 1 case where $l \in \mathcal{H}$, $l'_1 = l'_2$ and $L_1 =_{\mathcal{L}} L_2$, both configurations must execute the same case.

Case 1) $l, l' \in \mathcal{L}$

Since the only mapping updated in L' is $r_d \mapsto l'$ and it is changed equivalently in L_1 and L_2, $L'_1 =_{\mathcal{L}} L'_2$.

Case 2) $l \in \mathcal{L}, l' \in \mathcal{H}$

In this case, the low and high domains of L_1 and L_2 change, but they will still both change in the same way ($L(r_d)$ goes from $\mathcal{L}$ to $\mathcal{H}$ in both configurations).

Case 3) $l, l' \in High$

The domains of L_1 and L_2 do not change and none of the labels in the low domains change, so this does not change the low equivalence of L_1 and L_2.

Memories:

All of the compute instructions require that the labels of the operands flow to the labels of the destination. Furthermore, the location in memory to be updated is part of the instruction and therefore is equivalent in both configurations. Therefore, if the label of the destination is low, the operands will have low labels and be equivalent. This ensures that changes to low memory happen the same way in each configuration. Additionally, the proof that $L'_1 =_{\mathcal{L}} L'_2$ implies that the new low domains of memory will be equivalent as well ($L'_1 \approx L'_2$).

Program Counters and PC/Time Labels:

All branch/jump instructions require that their operands flow to pc_l. Therefore, with a low pc_l all of the operands will have low labels and be equivalent in successful configurations: $pc'_{v1} = pc'_{v2}$.

The `raiselbl` instruction can raise the pc_l and t_l. This is analogous to the UPLBL rule for memory locations and for similar reasons ensures that $pc'_1 =_{\mathcal{L}} pc'_2$ and $t'_{l1} =_{\mathcal{L}} t'_{l2}$.

$\square$

Corollary A.2 (Timing-Sensitive Noninterference Modulo Downgrading and Call Gates)**.** *For any two valid configurations, C_1 and C_2 and any low set of labels, $\mathcal{L}$, if insn $\not\equiv$ DWNLBL,UPCALL,UPRET,DWNCALL,DWNRET and, for all configuration steps $pc_l \in \mathcal{L} \implies t_l \in \mathcal{L}$,*

$$C_i \longrightarrow^* C_i^* \land C_1 =_{\mathcal{L}} C_2 \implies C_1^* =_{\mathcal{L}} C_2^*$$

Proof. This follows directly from the previous theorem and Property 2.2. If $t_l \in \mathcal{L}$, then low equivalence of t ensures that their values will be equal after each transition ($t_{v1} = t_{v2}$ and $t'_{v1} - t_{v1} = t'_{v2} - t_{v2}$ implies $t'_{v1} = t'_{v2}$). Essentially, the two configurations must execute in lock step. In this scenario the only instruction which can raise t_l is RAISELBL, so as long as $t_l \in \mathcal{L}$ and no RAISELBL instruction raises the t_l above pc_l, timing sensitivity is preserved. $\square$

Executing a complete upcall region from a low context will preserve low equivalence. Since the end state of the upcall is determined a priori by low-equivalent configurations, it ensures that both configurations will reach a state where they are again low-equivalent and have exactly the same call stack state as they did initially.

Lemma A.8 (High Upcall Noninterference)**.** *For any two valid configurations, C_1 and C_2, if $pc_l \in \mathcal{L}$, $C_1 =_{\mathcal{L}} C_2$, $insn \equiv$ `upcall`, $C_i \longrightarrow C'_i$ and $pc'_l \in \mathcal{H}$, then:*

$$(C_i \longrightarrow^* C''_i \equiv \langle CST_i, M''_i, L''_i, \mu''_i, (endpc, pc_{li}), (t''_{vi}, t_{li})\rangle) \wedge (C''_1 =_{\mathcal{L}} C''_2)$$

Proof. $C_i \longrightarrow C'_i$ and $C_1 =_{\mathcal{L}} C_2$.

By Lemma A.5 either both configurations execute the UPCALL or neither does.

If they execute the UPCALL, the labels of r_{s1}, r_{s2}, r_{s3} and r_d flow to pc_l, which implies that $pc'_l, t'_l, endpc$ and R_d have equal respective values in both configurations. Additionally, both configurations will add a new call stack entry with the following form: $((endpc, pc_l), (endt_i, t_{li}))$. These call stack entries are low-equivalent: pc_l and $endpc$ are the same in both traces. If $t_l \in \mathcal{H}$, then $endt_1 \neq endt_2$ but their values need not be equal for low equivalence. If $t_l \in \mathcal{L}$, then $t_{v1} = t_{v2}$ and by low equivalence of R_d, $endt_1 = endt_2$.

Pushing low-equivalent entries onto low-equivalent call stacks preserves low equivalence: $CST'_1 =_{\mathcal{L}} CST'_2$.

Additionally, since pc'_l and t'_l are low-equivalent across configurations, $pc'_1 =_{\mathcal{L}} pc'_2$ Therefore, $C'_1 =_{\mathcal{L}} C'_2$.

Once the upcall is executing, no `dwncall` or `upcall` instructions can execute. Therefore, by Lemma A.7, we can conclude that these configurations will only step to other low-equivalent configurations. Additionally, their call stacks will not be modified until they return from the upcall region via the UPRET-DONE rule.

At some point (when $t_{vi} = endt_i$) each configuration executes the UPRET-DONE rule. This may not happen after the same number of transition steps, and if $t_{li} \in \mathcal{H}$, it may not even happen at the same wall-clock time for each configuration.

When each configuration executes the UPRET-DONE rule, it pops the head of its call stack. We know that the original call stacks were low-equivalent, so popping off the heads results in the original call stacks.

Additionally, they will restore the original pc_l and t_l values, which were low-equivalent in the original configuration and both will set $pc_v = endpc$. Therefore, the `upcall` region executes without making any changes to state that violate low equivalence and the call stacks, pc and t_l restored by the UPRET-DONE are also low-equivalent.

Note that, if $t_l \in \mathcal{L}$, then end_t is also low-equivalent and these two configurations must execute the UPRET-DONE instruction at the same time. They may have executed a different number of instructions while in the call-gate region, but the wall-clock time will be identical. □

Theorem 2.2 (Noninterference Modulo Downgrading and Dwncalls)**.** *For any two valid configurations, C_1 and C_2, and any low set of labels, $\mathcal{L}$, where no instruction is a* `dwnlbl`*,* `dwncall` *or* `dwnret`*.*

$$(C_i \longrightarrow^* C_i') \wedge (C_1 =_{\mathcal{L}} C_2) \implies C_1' =_{\mathcal{L}} C_2'$$

Proof. This follows naturally from Theorem 2.1 and Lemma A.8.

As long as the current instruction is not an `upcall` or `upret-done`, both configurations will step ($\longrightarrow$) to low-equivalent configurations, by Theorem 2.1.

If the current instruction is an `upcall` or the UPRET-DONE rule applies, then we must consider both the high and low pc_l cases.

$pc_l \in \mathcal{L}$. By Lemma A.5, `upcall` will fail or succeed in both configurations. Let us first consider what happens when `upcall` executes successfully.

If $pc_l' \in \mathcal{H}$ then by Lemma A.8, both configurations will only step to low-equivalent states while in the upcall and will exit the upcall in low-equivalent states.

If $pc_l' \in \mathcal{L}$, then by Theorem 2.1 as long as no more `upcall` instructions execute, low equivalence is preserved. It is important to note that, by the UPCALL rule, $pc_l \sqcup t_l \sqsubseteq pc_l'$. In this case, it implies that $t_l \in \mathcal{L}$ and therefore the end time of the call gate will be exactly the same time in both configurations (same start and end, not just duration). If another UPCALL is executed while the upcall is running, it will fail in both configurations since they cannot be nested.

When the UPRET-DONE rule is applied, both configurations will step to low-equivalent configurations with equal pc_v and t_v. Since the implementation is deterministic (Property 2.1), the initial configurations are low-equivalent and the t_l in the call gate is exactly the same between both configurations, we are guaranteed that the same number of $\longrightarrow_{\mathcal{A}}$ steps execute in both configurations before the call gate expires at time *endt*. Without this guarantee, it would be possible for one upcall to exit having completed fewer instructions than the other and in that case their end states would not necessarily be low-equivalent.

$pc_l \in \mathcal{H}$. Either both configurations are inside a low-pc-originating upcall or neither is (by the same argument as in Lemma A.6, regarding call stack low-equivalent prefixes). If they are executing inside an `upcall` region, then this reduces to a case in Lemma A.8, which ensures they will eventually return outside of the upcall and will maintain low equivalence.

If neither one is in a low-pc-originating upcall, then their call stacks may differ: one of them may be in a upcall while the other configuration is not. However, we can apply both the corollary to Lemma A.6 and Lemma A.7 here to show that these configurations will step to low-equivalent configurations. The corollary states exactly the condition we have: if neither configuration is in a low-pc-originating upcall, and $pc_l \in \mathcal{H}$, then the resulting call stacks are low-equivalent. Lemma A.7 ensures that all other state is noninterfering for high pcs. Therefore, high pcs are always noninterfering as long as the configuration is valid.

□

Corollary A.3 (Timing-Sensitive-Noninterference Modulo Downgrading and Dwncalls)**.** *For any two valid configurations, C_1 and C_2, and any low set of labels, $\mathcal{L}$, where no instruction is a `dwnlbl`, `dwncall` or `dwnret` and for all pc, $pc_l \in \mathcal{L} \implies t_l \in \mathcal{L}$.*

$$(C_i \longrightarrow^* C_i') \wedge (C_1 =_{\mathcal{L}} C_2) \implies C_1' =_{\mathcal{L}} C_2'$$

Proof. This is a direct corollary to Theorem 2.2 and is strictly stronger than

Corollary A.2 since it also allows the use of low-deterministic UPCALL instructions (i.e. no timing mitigation). □

Theorem 2.3 (Nonmalleable Information Flow). *For attacker induced high label sets $\mathcal{S}$ and $\mathcal{U}$ and their respective complements, $\mathcal{P}$ and $\mathcal{T}$, and valid configurations,* $\forall\{s,u\} \in \{1,2\}, C_{su}$

$$C_{su} \longrightarrow C'_{su} \;\forall u \in \{1,2\}.\, C_{1u} =_{\mathcal{P}} C_{2u} \;\forall s \in \{1,2\}.\, C_{s1} =_{\mathcal{T}} C_{s2}$$
$$\Longrightarrow$$
$$C'_{11} =_{\mathcal{P}} C'_{21} \implies C'_{12} =_{\mathcal{P}} C'_{22} \quad \wedge \quad C'_{11} =_{\mathcal{T}} C'_{12} \implies C'_{21} =_{\mathcal{T}} C'_{22}$$

Proof. For all of the instructions other than `dwncall/dwnret` and `dwnlbl`, Theorem 2.2 implies the nonmalleability condition. Therefore, we only need to consider how the new instructions affect processor state.

Additionally, since the conditions are exactly dual we only prove the first of the two requirements:

$$C'_{11} =_{\mathcal{P}} C'_{21} \implies C'_{12} =_{\mathcal{P}} C'_{22}$$

First we consider the `dwnlbl` instruction. We have already proven that this instruction results in low-equivalent configurations for high pcs in Lemma A.7. Therefore, if $pc_l \in \mathcal{S} \cup \mathcal{U}$, `dwnlbl` results in low equivalent configurations for both $\mathcal{L} = \mathcal{P}$ and $\mathcal{L} = \mathcal{T}$.

Therefore, we now consider the case where $pc_l \in \mathcal{P} \cap \mathcal{T}$. The `dwnlbl` instruction modifies the label of r_d ($l = L(r_d)$) to a new label value ($l' = \gamma(R_{s1})$). In the case where RELBL_ARG_ERROR rule matches, all configurations will step to the error pc and not modify any other state (by the same argument as for

UPLBL_ARG_ERROR in Lemma A.5). If that error rule does not match, l' is equivalent in all four configurations since $L(r_{s1})$ flows to pc_l and $pc_l \in \mathcal{P} \cap \mathcal{T}$.

Since l' is equivalent in all configurations, then the label checks: $pc_l \sqsubseteq l'$ and $l' \sqsubseteq \bowtie(pc_l)$ will both succeed or fail in all four configurations. If they fail, then all four new configurations still maintain low equivalence since $L_{\mathbf{su}}$ is not modified in any of them.

Next let us consider the label check $l \sqsubseteq \bowtie(l)$.

$L_1 =_{\mathcal{L}} L_2 \implies L_1 \approx L_2$ for any set of low labels, $\mathcal{L}$. Additionally, domain equivalence ($\approx$) is transitive. Therefore, all four starting configurations agree on the domain of all locations ($L_{\mathbf{11}} \approx L_{\mathbf{21}} \approx L_{\mathbf{12}} \approx L_{\mathbf{22}}$). We will now consider the four possible quadrants in which l can reside:

Case 1) $l \in \mathcal{P} \cap \mathcal{T}$

In this case, l is exactly the same in all four configurations. Therefore, label checking will succeed or fail in all four configurations and result in the same modifications to L (l' and r_d are also equal in all four configurations). The low equivalence relations between configurations will be maintained.

Case 2) $l \in \mathcal{P} \cap \mathcal{U}$

For any two public equivalent ($=_{\mathcal{P}}$) configurations then this will also result in the same label checking result and the same modifications to L. Any configurations which were public equivalent before this instruction are still public equivalent.

$$C_{\mathbf{1u}} =_{\mathcal{P}} C_{\mathbf{2u}} \implies C'_{\mathbf{1u}} =_{\mathcal{P}} C'_{\mathbf{2u}}$$

Case 3) $l \in \mathcal{S} \cap \mathcal{T}$

For any two trusted equivalent ($=_{\mathcal{T}}$) configurations then this will result in the same label checking result and same modifications to L. Any configurations which were trusted equivalent before this instruction are still trusted equivalent.

$$C_{\mathbf{s1}} =_{\mathcal{T}} C_{\mathbf{s2}} \implies C'_{\mathbf{s1}} =_{\mathcal{T}} C'_{\mathbf{s2}}$$

In this case, we know that $L'_{\mathbf{s1}} =_{\mathcal{T}} L'_{\mathbf{s2}}$ and we assume in the premise that $L'_{\mathbf{11}} =_{\mathcal{P}} L'_{\mathbf{21}}$. Therefore, $L'_{\mathbf{12}} \approx L'_{\mathbf{11}} \approx L'_{\mathbf{21}} \approx L'_{\mathbf{22}}$.

Next, we will consider the four possible quadrants for l':

Case 1) $l' \in \mathcal{P} \cap \mathcal{T}$

In this case, l' has the same value in each resulting configuration by transitivity of low-equivalent labels ($L'_{12} =_{\mathcal{P}\cap\mathcal{T}} L'_{22}$). Regardless of whether or not configurations $C_{\mathbf{12}}$ and $C_{\mathbf{22}}$ both successfully executed the `dwnlbl` instruction, they resulted in updating the same location to the same label value. Therefore, $C'_{\mathbf{12}}$ and $C'_{\mathbf{22}}$ are public equivalent.

Case 2) $l' \in \mathcal{P} \cap \mathcal{U}$

If this instruction succeeds in $C_{\mathbf{12}}$ it will modify both the $\mathcal{T}$ domain and the $\mathcal{P}$ domain (i.e. a label is going from high to low in confidentiality and low to high in integrity). However, if it fails then it will not modify either set of domains. Since $L'_{12} \approx L'_{22}$ this success or failure must be the same in both of these configurations (failure in one but not the other would imply $L'_{12} \not\approx L'_{22}$). Therefore it must either succeed or fail in both of these configurations. If it fails, L'_{12} and L'_{22} are trivially low-equivalent. If it succeeds, L'_{12} and L'_{22} are modified in the same way (since l' has the same value across all four configurations) and are still public equivalent.

Case 3) $l' \in \mathcal{S} \cap \mathcal{T}$

In this case, whether or not each configuration successfully executed the `dwnlbl` instruction has no bearing on the public equivalence of L'_{12} and L'_{22}. Since l' is a secret label, its exact value may differ in these two label mappings. Therefore, C'_{12} and C'_{22} remain public equivalent.

Case 4) $l' \in \mathcal{S} \cap \mathcal{U}$

By the same logic as in case 3, C'_{12} and C'_{22} remain public equivalent in this case as well.

Case 4) $l' \in \mathcal{S} \cap \mathcal{U}$

By Lemma A.1, if $l \in \mathcal{S} \cap \mathcal{U}$, then l is compromised and the label check does not pass. Therefore, L is not updated in any configuration and the resulting configurations remain low-equivalent.

Dwncalls:

Now we show that the `dwncall` and `dwnret` instructions maintain nonmalleability.

Based on the pc_l of the starting configurations, there are 3 cases to prove, based on its quadrant (since pc_l is *valid* it cannot be in $\mathcal{S} \cap \mathcal{U}$, by Lemma A.1).

Case 1) $pc_l \in \mathcal{P} \cap \mathcal{T}$

First, if the $L(r_{s1}) \sqsubseteq pc_l$ check fails then this will fail in all configurations and low equivalence is preserved. Similarly, if the check that this call is going down in the lattice ($pc'_l \sqcup t'_l \sqsubset pc_l$) fails, this will fail in all four configurations.

Otherwise, pc' and t' will be the same in all four configurations, since they reference the same entry in the gate registry.

If the `dwncall` executes successfully in C_{12} because CST_{12} is empty, then CST_{11} must either also be empty or contain only public entries. By Lemma A.2, it cannot contain any secret-untrusted entries and by trusted equivalence with CST_{12}, it cannot contain any secret-trusted entries. By the premise ($C'_{11} =_{\mathcal{P}} C'_{21}$) CST_{21} must be empty if CST_{11} is empty and must otherwise contain public entries. By trusted equivalence, CST_{22} must also be either empty or contain only public entries. Since $CST_{12} =_{\mathcal{P}} CST_{22}$, the latter must also be empty.

This argument is symmetric (we could have started reasoning with CST_{22} and concluded that CST_{12} must be empty), the `dwncall` succeeds in C_{12} if and only if it succeeds in C_{22}.

In the case where the instruction succeeds, both of the configurations will push on public equivalent call stack entries and will jump to the same new pc_l and pc_v based on the gate registry entry.

In any case where the instruction fails in both configurations, no state changes other than the pc_v and they will therefore remain low-equivalent.

Case 2) $pc_l \in \mathcal{P} \cap \mathcal{U}$

If the $L(r_{s1}) \sqsubseteq pc_l$ check fails in C_{12} then it will also fail in C_{22}. Similarly, if the check that this call is going down in the lattice ($pc'_l \sqcup t'_l \sqsubset pc_l$) fails in C_{12} it will also fail in C_{22}.

Otherwise, pc' and t' will be the same in both configurations, since they reference the same entry in the gate registry.

The reasoning from Case 1 about call stack equivalence holds in this case as well since it does not depend at all on the current pc_l of the configurations. Therefore, C_{12} and C_{22} will either both fail or both succeed on this `dwncall` instruction. In the event the instruction succeeds, since their pc_l are public equivalent, the call stack entries that are pushed will be public equivalent as well, resulting in public equivalent configurations.

Case 3) $pc_l \in \mathcal{S} \cap \mathcal{T}$

In this case, if either C_{12} or C_{22} successfully executes a `dwncall`, and the resulting pc labels (pc'_l) are both secret, then the resulting configurations are public equivalent. By Lemma A.6 (with $\mathcal{L} = \mathcal{P}$ and $\mathcal{H} = \mathcal{S}$), the resulting call stacks are public equivalent. Similarly, the pc and t_l are public equivalent, since they remain secret.

Alternatively, either configuration may execute a `dwncall` such that $pc'_l \in \mathcal{P} \cap \mathcal{T}$. It is impossible for pc'_l to be untrusted, since `dwncall` requires that $pc'_l \sqsubseteq pc_l$.

In this case, C_{11} must also be empty. Since CST_{11} is valid, the label of its first entry must be ordered with respect to pc_l. Since no label in $\mathcal{S} \cap \mathcal{T}$ flows to any label in $\mathcal{P} \cap \mathcal{U}$ and no label in $\mathcal{P} \cap \mathcal{U}$ flows to any label in $\mathcal{S} \cap \mathcal{T}$, that first entry must not be public and untrusted. However, by trusted equivalence with CST_{12}, it cannot contain any secret entries either, and must therefore be empty.

By combining this with the call stack equivalence reasoning used in cases 1 and 2, this means that either all four call stack configurations are empty, or they all contain public, trusted entries.

Therefore, if C_{12} executes a `dwncall` successfully, then so must C_{11}, since all other arguments to the instruction flow to the pc_l and are equal between the two configurations, and as reasoned above, both must have empty call stacks.

Since the call gate entries will be the same across C_{12} and C_{11} they will both end up with the same pc_v and pc_l in their new configurations. Similarly, they will push trusted equivalent call stack entries (whose labels are secret and trusted) and $C'_{12} =_{\mathcal{T}} C'_{11}$.

Furthermore, $pc'_l \in \mathcal{P} \cap \mathcal{T}$, which implies that C'_{21} will have the same pc'_l and pc'_v as configuration C'_{12}, and will also have a secret, trusted call stack entry. Since C_{21} must have successfully executed the same `dwnlbl` instruction to end up in this state, by trusted equivalence C_{22} also must execute the same `dwnlbl` instruction and will also end up with the same new pc'_l and pc'_v as the other configurations. Lastly, since the new call stack entries are all secret and trusted, $CST'_{12} =_{\mathcal{P}} CST'_{22}$ because neither contains any public entries.

Dwnret:

This is essentially analogous to `dwncall` instruction, except we are restoring saved pc_v and pc_l from the stored call stack entries. Since the reasoning is so similar we will omit a full proof and include only a sketch.

Case 1) $pc_l \in \mathcal{P}$

In this case both C_{12} and C_{22} will either execute the `dwnret` or not, since the INUPCALL check must have the same result in both cases; if either configuration executes a `dwnret` successfully, then all four configurations must successfully execute a similar `dwnret`.

In the case where the call stack entry is public, then both configurations must have the same call stack entry and will end up in public equivalent configurations. If the call stack entry is secret and trusted, then by similar transitive arguments for `dwncall`, all four configurations will execute a `dwnret` to secret, trusted *pc*s and will all have empty call stacks.

Case 2) $pc_l \in \mathcal{S} \cap \mathcal{T}$

If C_{12} executes a `dwnret` successfully, the resulting pc_l' must also be secret and trusted. Similarly, its new call stack is empty and therefore still public equivalent to the call stack of C_{22} (which must have only contained secret entries, or no entries at all). In this case, $C_{12}' =_{\mathcal{P}} C_{22}'$ since it must also have a secret pc (the only instruction for lowering the pc_l is `dwncall` and by the earlier portion of this proof, it could not have executed a successful `dwncall`).

Similarly, if C_{12} executes a `dwnret` which does not pass label checking, the result will remain public equivalent with C_{22}'.

□

APPENDIX B

APPENDIX TO CHAPTER 4

B.1 Definitions

Definition B.1 (Erase-To Function). *Each label implies a particular erasure policy. For lattice levels and label functions, no erasure is required. For erasure labels, the accompanying condition implies a minimum label of erasure. Meets and joins of erasure labels imply multiple required erasure conditions; therefore this function returns a list of conditions, with their accompanying label.*

$\boxed{\text{eraseTo}(\tau) \dashv L}$

$$\textsc{Label}\ \frac{}{\text{eraseTo}(l) \dashv [\,]}$$

$$\textsc{Function}\ \frac{}{\text{eraseTo}(f(x)) \dashv [\,]}$$

$$\textsc{Erase}\ \frac{}{\text{eraseTo}(b_1 \overset{c(\vec{x})}{\nearrow} b_2) \dashv [(c(\vec{x}), b_2)]}$$

$$\textsc{Join}\ \frac{\begin{array}{c} L_l = \text{eraseTo}(\tau_1) \qquad L_r = \text{eraseTo}(\tau_2) \qquad \forall (c_l, l_l) \in L_l,\ (c_r, l_r) \in L_r. \\ (c_l, l_l \sqcup \tau_2) \in L \qquad (c_r, l_r \sqcup \tau_1) \in L \qquad (c_l \wedge c_r, l_l \sqcup l_r) \in L \end{array}}{\text{eraseTo}(\tau_1 \sqcup \tau_2) \dashv L}$$

$$\textsc{Meet}\ \frac{\begin{array}{c} L_l = \text{eraseTo}(\tau_1) \qquad L_r = \text{eraseTo}(\tau_2) \qquad \forall (c_l, l_l) \in L_l,\ (c_r, l_r) \in L_r. \\ (c_l, l_l \sqcap \tau_2) \in L \qquad (c_r, l_r \sqcap \tau_1) \in L \qquad (c_l \wedge c_r, l_l \sqcap l_r) \in L \end{array}}{\text{eraseTo}(\tau_1 \sqcap \tau_2) \dashv L}$$

$$\boxed{\mathcal{P} = \langle rob, A, \mu \rangle}$$

$$\textsc{Fetch}\ \frac{\begin{array}{c} rob = \_ :: x \\ rob' = rob :: x' \\ x' = \text{predict}(\text{arch}(x, A), \mu) \\ \mu' = \text{update}(\text{arch}(x, A), \mu) \end{array}}{\mathcal{P} \xrightarrow[\text{fetch } x]{} \mathcal{P}'} \qquad \textsc{Exec}\ \frac{\begin{array}{c} rob = \_ :: x :: \_ \\ \mu' = \text{compute}(\text{arch}(x, A), \mu) \end{array}}{\mathcal{P} \xrightarrow[\text{exec } x]{} \mathcal{P}'}$$

$$\textsc{Commit}\ \frac{\begin{array}{c} rob = x :: rob' \\ A' = \text{commit}(\text{arch}(x, A), \mu) \end{array}}{\mathcal{P} \xrightarrow[\text{commit } x]{} \mathcal{P}'} \qquad \textsc{Miss}\ \frac{\begin{array}{c} rob = rob'' :: x :: \_ \\ rob' = rob'' :: x :: x' \\ x' = \text{real-npc}(\text{arch}(x, A), \mu) \\ \mu' = \text{miss-upd}(\text{arch}(x, A), \mu) \end{array}}{\mathcal{P} \xrightarrow[\text{miss } x]{} \mathcal{P}'}$$

$$\textsc{Clock}\ \frac{}{\mathcal{P} \xrightarrow[\text{clock}]{A} \mathcal{P}} \qquad \frac{d = \text{sched}(\text{arch}(rob, A), \mu) \qquad \mathcal{P} \xrightarrow[d]{} \mathcal{P}'}{\mathcal{P} \rightarrow \mathcal{P}'}\ \textsc{Schedule}$$

Figure B.1: Semantics for Abstract Out-of-Order Processors

B.2 Proofs

B.2.1 Processor Guarantees

The preservation of Temporal Ordering in a speculative processor relies on the fact that all trace of a given speculative program order is removed once that misspeculation is discovered. This property translates in to the following lemma:

Lemma 4.1. *For any register, x, in a well-typed $\mathcal{P}$,*

$$\forall i_j^{[s_1,\ldots,s_n]}, i_k^{[p_1,\ldots,p_n]} \in \mathcal{I}(x), \quad j \le k \implies [s_1, \ldots s_n] \preceq [p_1, \ldots, p_n]$$

where $\preceq$ is prefix order.

$\boxed{\tau \sqsubseteq \tau'}$

$$\textsc{Labels}\ \frac{l_1 \sqsubseteq_l l_2}{l_1 \sqsubseteq l_2} \qquad \textsc{Transitive}\ \frac{\tau \sqsubseteq \tau'' \qquad \tau'' \sqsubseteq \tau'}{\tau \sqsubseteq \tau'} \qquad \textsc{Reflexive}\ \frac{}{\tau \sqsubseteq \tau}$$

$$\textsc{Join-Intro}\ \frac{\tau \sqsubseteq \tau_1 \vee \tau \sqsubseteq \tau_2}{\tau \sqsubseteq \tau_1 \sqcup \tau_2} \qquad \textsc{Join-Elim}\ \frac{\tau_1 \sqsubseteq \tau \qquad \tau_2 \sqsubseteq \tau}{\tau_1 \sqcup \tau_2 \sqsubseteq \tau}$$

$$\textsc{Meet-Intro}\ \frac{\tau \sqsubseteq \tau_1 \qquad \tau \sqsubseteq \tau_2}{\tau \sqsubseteq \tau_1 \sqcap \tau_2} \qquad \textsc{Meet-Elim}\ \frac{\tau_1 \sqsubseteq \tau \vee \tau_2 \sqsubseteq \tau}{\tau_1 \sqcap \tau_2 \sqsubseteq \tau}$$

$$\textsc{Erase-Elim}\ \frac{\tau_2 \sqsubseteq \tau'}{\tau_1 \ {}^{c(\vec{v})}\!\!\nearrow \tau_2 \sqsubseteq \tau'} \qquad \textsc{Erase-Intro}\ \frac{\tau \sqsubseteq \tau_1}{\tau \sqsubseteq \tau_1 \ {}^{c(\vec{v})}\!\!\nearrow \tau_2}$$

$$\textsc{Erase-Weaken}\ \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2' \qquad \forall \sigma . \sigma \vDash c(\vec{v}) \implies \sigma \vDash c'(\vec{v}')}{\tau_1 \ {}^{c(\vec{v})}\!\!\nearrow \tau_2 \sqsubseteq \tau_1' \ {}^{c'(\vec{v}')}\!\!\nearrow \tau_2'}$$

Figure B.2: The complete environment-independent may-flow-to relation. $\sqsubseteq_l$ is the ordering relation of the lattice of basic security labels.

Proof Sketch.

Base Case. At first, this vacuously holds since no state is yet influenced by an instruction.

Fetch. Each newly fetched instruction has a larger instruction index than any prior instruction, and either keeps the same speculative path or adds a new prediction:

$$\frac{pc \approx i_j^{[s_1,\ldots,s_n]} \qquad \mathcal{I}(pc') = \mathcal{I}(pc) + i_{j+1}^{[s_1,\ldots,s_n]} \vee \mathcal{I}(pc') = \mathcal{I}(pc) + i_{j+1}^{[s_1,\ldots,s_n,s_{n+1}]}}{\text{fetch } pc}$$

In the above, $pc \approx i_j^{[s_1,\ldots,s_n]}$ denotes the logical instruction associated with the current instruction address pc.

Exec & Commit. Interim execution cannot introduce *new* speculative paths or instructions and thus cannot affect this invariant.

Miss. The miss case is similar to the *Fetch* case, in that the new instruction x' has the same influence set as instruction x, with a higher instruction index and *does not extend* x's speculative path.

In this case, μ and the ROB may now contain registers influenced by x' and instructions along the misspeculated path, violating our invariant. However, end-to-end erasure effectively allows us to remove the influence of erased instructions from influence sets (since their values have no impact on future execution).

All instructions ordered after x in ROB order (i.e., $j \leq k$) are erased since they must all have labels l such that $SL(x, head)_{\sigma} \sqsubseteq l$; this is exactly the set of all instructions whose speculation paths are *not* prefixes of instruction x's.

After this influence removal, all registers' influence sets now contain only instructions that satisfy our invariant. □

B.2.2 Noninterference Guarantees

Ordered label comparisons can be typed more permissively than arbitrary dynamic label comparisons. The *lower* of the two labels gives a safe visibility bound of the comparison instead of the *higher*.

Theorem B.1 (Safety of Ordered Label Comparisons)**.** *The result of any label comparison (on ordered labels) is guaranteed to be low equivalent at or above the meet of the labels of the labels.*

$$\begin{aligned}
&\forall l.x, y \in VARS. \\
&\qquad \sigma_1 \approx_l \sigma_2 \ \wedge\ (\forall \sigma. \Gamma(x) \,{}_{\sigma}\!\sqsubseteq \Gamma(y) \vee \Gamma(y) \,{}_{\sigma}\!\sqsubseteq \Gamma(x)) \\
&\qquad\quad \wedge\ \Gamma(x) \sqcap \Gamma(y) \,{}_{\sigma_1}\!\sqsubseteq l \implies \\
&\qquad\qquad \Gamma(x) \,{}_{\sigma_1}\!\sqsubseteq \Gamma(y) = \Gamma(x) \,{}_{\sigma_2}\!\sqsubseteq \Gamma(y)
\end{aligned}$$

Proof. By the definition of observational equivalence, both σ_1 and σ_2 agree on whether $\Gamma(x)$ and $\Gamma(y)$ are in the high ($\mathcal{H}$) or low ($\mathcal{L}$) sets: any label which flows to l is in the low set, everything else is in the high set. There are four possible cases which correspond to the sets that contain $\Gamma(x)$ and $\Gamma(y)$ respectively:

$\mathcal{L}$, $\mathcal{L}$. In this case, by observational equivalence and the well-formedness of labels, any free variables that occur in either $\Gamma(x)$ or $\Gamma(y)$ have the same values in both σ_1 and σ_2. The result of the flows to check is decided only by the syntax of the labels (which is static) and these free variables; thus the result will be the same regardless of which σ is used.

$\mathcal{L}$, $\mathcal{H}$. In this case, $\Gamma(y) \,{}_{\sigma}\!\not\sqsubseteq \Gamma(x)$ by the definition of $\mathcal{L}$. Therefore, the expression must return true in both σ_i since either $\Gamma(y) \,{}_{\sigma}\!\sqsubseteq \Gamma(x)$ or $\Gamma(x) \,{}_{\sigma}\!\sqsubseteq \Gamma(y)$ by the ordering assumption.

$\mathcal{H}, \mathcal{L}$. This is symmetric to the prior case, except the expression must return false.

$\mathcal{H}, \mathcal{H}$. In this case we cannot be sure that the result is equivalent in both σ_i since there are no equality constraints on x or y. However, $\Gamma(x) \sqcap \Gamma(y)$ must be exactly equal to either $\Gamma(x)$ or $\Gamma(y)$ since they are ordered; therefore $\Gamma(x) \sqcap \Gamma(y) \in \mathcal{H}$(i.e., $\Gamma(x) \sqcap \Gamma(y) \,_{\sigma}\not\sqsubseteq l$). □

Noninterference. We primarily rely on SecVerilog's semantics for Verilog and their proof of Noninterference [138]. However, we *do not* include dynamic clearing semantics in assignments and have slightly different type well formedness assumptions (see Figure 4.3). Here, we present a modified version of their semantics and amendments to any lemmas and proofs from their report needed to accommodate those modifications.

The *main difference* between their work and ours is the differentiation between combinationally assigned variables and sequentially assigned variables, which we adopt from Ferraiulolo et al. [45]. We associate all variables with either a `seq` or `com` annotation and check the following:

- All `seq` variables are assigned in a sequential (clock-triggered) thread, via non-blocking assignment.
- All `com` variables are assigned in *any thread* via blocking assignment.

$\boxed{\langle \sigma, c, AS, NB\rangle \Downarrow \langle \sigma', AS', NB'\rangle}$

$$\textsc{Skip}\ \frac{}{\langle \sigma, \texttt{skip}, AS, NB\rangle \Downarrow \langle \sigma, AS, NB\rangle}$$

$$\textsc{Seq}\ \frac{\langle \sigma, c_1, AS, NB\rangle \Downarrow \langle \sigma'', AS'', NB''\rangle \qquad \langle \sigma'', c_2, AS'', NB''\rangle \Downarrow \langle \sigma', AS', NB'\rangle}{\langle \sigma, c_1; c_2, AS, NB\rangle \Downarrow \langle \sigma', AS', NB'\rangle}$$

$$\textsc{Assign}\ \frac{\langle \sigma, e\rangle \Downarrow n \qquad \sigma' = \sigma\{v \mapsto n\} \qquad \Gamma(v) \downarrow_{\sigma'} \tau \qquad AS' = AS \cup \{(v, n, \tau)\}}{\langle \sigma, v = e, AS, NB\rangle \Downarrow \langle \sigma', AS', NB\rangle}$$

$$\textsc{Assign-NB}\ \frac{\langle \sigma, e\rangle \Downarrow n \qquad \Gamma(v) \downarrow_{\texttt{next}\ \sigma} \tau \qquad NB' = NB \cup \{(v, n, \tau)\}}{\langle \sigma, v \Leftarrow e, AS, NB\rangle \Downarrow \langle \sigma, AS, NB'\rangle}$$

$$\textsc{If}\ \frac{\langle e, \sigma\rangle \Downarrow n \qquad i = (n \neq 0)\,?\,1 \,:\, 2 \qquad \langle \sigma, c_i, AS, NB\rangle \Downarrow \langle \sigma', AS', NB'\rangle}{\langle \sigma, \text{if } (e)\, c_1 \text{ else } c_2, AS, NB\rangle \Downarrow \langle \sigma', AS', NB'\rangle}$$

Figure B.3: Big-step operational semantics of commands.

$$\textsc{Step}\ \frac{\vec{c} \neq \emptyset \qquad \vec{c'} = (\vec{c} - \{c_1\}) \cup \mathcal{B} \downharpoonleft_{AS} \qquad \langle \sigma, c_1, \emptyset, NB\rangle \Downarrow \langle \sigma', AS, NB'\rangle}{\langle t, \sigma, \vec{c}, \mathcal{B}, NB\rangle \rightarrow \langle t, \sigma', \vec{c'}, \mathcal{B} - \mathcal{B} \downharpoonleft_{AS}, NB'\rangle}$$

$$\textsc{Tick}\ \frac{\vec{c} = \mathcal{B} \downharpoonleft_{NB} \cup \mathcal{S} \qquad \sigma' = \text{apply}(\sigma, NB)}{\langle t, \sigma, \emptyset, \mathcal{B}, NB\rangle \xrightarrow{(t,\sigma)} \langle t+1, \sigma', \vec{c}, C - \mathcal{B} \downharpoonleft_{NB}, \emptyset\rangle}$$

Figure B.4: Big-step operational semantics of threads. $\mathcal{S}$ is the set of sequential threads, and C is the set of combinational threads.

$\boxed{\mathcal{M}, \Gamma, pc \vdash c}$

$$\textsc{ComAssign}\ \frac{\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \notin FV(\tau') \qquad C(\bullet\eta) \implies pc \sqcup \tau \,{}_{\sigma}{\sqsubseteq}\, \tau'}{\mathcal{M}, \Gamma, pc \vdash x =_{\eta} e}$$

$$\textsc{ComAssign-Rec}\ \frac{\begin{array}{c}\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \in FV(\tau') \\ x' \notin \Gamma \qquad \tau'' = \tau'\{x'/x\} \qquad \textit{if}\ x \notin \mathcal{M}, C(\bullet\eta) \implies pc \,{}_{\sigma}{\sqsubseteq}\, \tau' \\ C(\bullet\eta), x' = \lfloor e \rfloor_a \implies pc \sqcup \tau \,{}_{\sigma}{\sqsubseteq}\, \tau''\end{array}}{\mathcal{M}, \Gamma, pc \vdash x =_{\eta} e}$$

$$\textsc{SeqAssign}\ \frac{\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \notin FV(\tau') \qquad C(\bullet) \implies pc \sqcup \tau \,{}_{\sigma}{\sqsubseteq}_{\textbf{next}}\, \tau'}{\mathcal{M}, \Gamma, pc \vdash x \Leftarrow_{\eta} e}$$

$$\textsc{SeqAssign-Rec}\ \frac{\begin{array}{c}\Gamma \vdash e \dashv \tau \qquad \Gamma(x) = \tau' \qquad x \in FV(\tau') \\ \textit{if}\ x \notin \mathcal{M}, C(\bullet) \implies pc \,{}_{\sigma}{\sqsubseteq}\, \tau' \qquad C(\bullet\eta) \implies pc \sqcup \tau \,{}_{\sigma}{\sqsubseteq}_{\textbf{next}}\, \tau'\end{array}}{\mathcal{M}, \Gamma, pc \vdash x \Leftarrow_{\eta} e}$$

$$\textsc{If}\ \frac{\Gamma \vdash e \dashv \tau \qquad \mathcal{M} \cap DA(\eta), \Gamma, pc \sqcup \tau \vdash c_1 \qquad \mathcal{M} \cap DA(\eta), \Gamma, pc \sqcup \tau \vdash c_2}{\mathcal{M}, \Gamma, pc \vdash \text{if}_{\eta}(e)\, c_1\ \text{else}\ c_2}$$

Figure B.5: Typing rules for commands.

We also have a special typing rule for programs which ensures that all sequential variables are not implicitly downgraded. If a variable is not assigned, then its label this cycle must flow to its label next cycle:

Definition B.2 (No Implicit Downgrades). *A program,* prog*, is well-typed if, for all variables, v:*

$$v \in seq \wedge \langle t, \sigma \rangle \xrightarrow{(t,\sigma)} \langle t+1, \sigma' \rangle \wedge \text{Not Assigned}(v) \implies \Gamma(v) \downarrow_{\sigma} \sqsubseteq \Gamma(v) \downarrow_{\sigma'}$$

Lastly, we also assume the same well-formedness assumptions as prior work:

- No combinational (`com`) variables generate inferred latches.
- There are no combinational loops.
- There are no race conditions on any assignments (so the non-deterministic semantics leads to deterministic updates to registered state).

Figures B.3 and B.4 depict the stepping rules for commands and threads. Commands generate a new variable environment, a set of combinationally assigned variables (*AS*), and pending non-blocking assignments (*NB*). To aid the proof, we annotate combinational assignments with the *new* labels of the variables after the assignment; these are not actually compiled into the circuits, nor necessary for execution.

Threads execute (combinational) commands non-deterministically, which causes other commands to become unblocked. Once no more unblocked commands remain, the clock ticks and all pending non-blocking assignments are applied. The `TICK` step also increments a counter that corresponds to real-time clock ticks.

Figure B.5 depicts the complete typing rules for commands. First, we briefly describe notation and rules that were not included in the main chapter.

Like the original SecVerilog type system [138], we use a *definite assignment* analysis to remove unnecessary checks for sensitive upgrades. For each `If` statement, the definite-assignment analysis is used to determine the set of variables that can be assigned to without a traditional no-sensitive-upgrade check.

In most of the rules, we rely on an abstract program analysis, C, which contains facts about program variables that are either invariants,$C(\bullet)$ or true before a given program statement, η, executes, $C(\bullet\eta)$. The analysis used in the SpecVerilog compiler is based on the one presented in the original SecVerilog work.

For the following proofs, we omit notation when irrelevant. For instance, we may use the shorthand $\langle\sigma, c\rangle \Downarrow \sigma'$ for command execution when the set of assigned variables is unnecessary.

Proof of Noninterference. We proceed by following the proofs by Zhang et al. [138] with modifications where necessary to account for our language changes. We use some of their lemmas (restated here) without proof since the proofs remain unchanged. In those cases we use the same lemma numbering to facilitate reference to the original proofs.

First, we consider noninterference from executing a single command:

Theorem B.2 (Single-Command Noninterference)**.**

$$\forall l \in \mathcal{L}, i \in \{1,2\}. (\vdash \Gamma) \wedge (\vdash \Gamma c) \wedge (\sigma_1 \approx_l \sigma_2) \wedge (\langle\sigma, c_i\rangle \Downarrow \sigma'_i) \implies \sigma'_1 \approx_l \sigma'_2$$

For the purpose of the proof, we extend Single-Command Noninterference to include the low-equivalence of the compiled combinational assignments, AS. The projection up to level l ($\upharpoonright_l$) is the longest subsequences of AS such that, $\forall (x, v, \tau) \in AS \upharpoonright_l . \tau \sqsubseteq l$. We define low-equivalence of these accumulated assignments using this projection: $AS_1 \approx_l AS_2 \iff AS_1 \upharpoonright_l = AS_2 \upharpoonright_l$.

We define the same operations on non-blocking assignments, NB. Note that the label in NB represents the *next-cycle* evaluation of the label (and not the label of the variable immediately after the assignment, since the assignment is delayed). The next-cycle function is defined by the next environment returned from the execution trace via the `TICK` rule in Figure B.4.

Theorem B.2 (Single-Command Noninterference)**.**

$$\forall l \in \mathcal{L}, i \in \{1,2\}. (\vdash \Gamma) \land (\vdash \Gamma c) \land (\sigma_1 \approx_l \sigma_2) \land (\langle \sigma, c_i \rangle \Downarrow \sigma'_i) \implies \sigma'_1 \approx_l \sigma'_2$$

Proof. By induction on the structure of c.

Assign: $v = e$**.**

Let $\Gamma \vdash e \dashv \tau$ and (v, n_i, τ_i) be the assignment generated for AS_i.

First, consider the case when $\tau_1 \sqsubseteq l$:

By the typing rules, $\tau \,{}_{\sigma_1}\!\sqsubseteq \tau_1 \sqsubseteq l$. By Lemma 8, $n_1 = n_2$. So $\sigma'_1[v] = \sigma'_2[v]$.

Next, we show that $\tau_2 \sqsubseteq l$. The only variables that may appear in $\Gamma(v)$ are either of `seq` type or are v itself and must be equivalent in both configurations:

For each variable $u \in FV(\Gamma(v))$, $\Gamma(u) \,{}_{\sigma}\!\sqsubseteq \Gamma(v)$. Since $\Gamma(v) \,{}_{\sigma_1}\!\sqsubseteq l$, $\Gamma(u) \,{}_{\sigma_1}\!\sqsubseteq l$, and by the inductive hypothesis, $\Gamma(u) \,{}_{\sigma_2}\!\sqsubseteq l \land \sigma_1[u] = \sigma_2[u]$. Since $\sigma'_1[v] = \sigma'_2[v]$, $\tau_2 = \tau_1 \sqsubseteq l$. Therefore $AS'_1 \approx_l AS'_2$.

By well-formedness, $v \in$ com and by well-formedness of the typing environment ($\vdash \Gamma$), the only variable whose label could change after the assignment is v, if $v \in FV(\Gamma(v))$. Therefore, $\forall u \neq v. (\Gamma(u) \downarrow_{\sigma}) = (\Gamma(u) \downarrow_{\sigma'})$. Additionally, no other variables are modified by the assignment: $\forall u \neq v. \sigma_i[v] = \sigma'_i[v]$.

Thus, $\sigma'_1 \approx_l \sigma'_2$.

Second, consider the case when $\tau_1 \not\sqsubseteq l$:

In this case, $\tau_2 \not\sqsubseteq l$ as well (otherwise, $\tau_1 \sqsubseteq l$). Therefore, $AS'_1 \approx_l AS'_2$.

By the same argument as above, the environments otherwise preserve low-equivalence as no other variables or their labels change.

In both cases, $NB'_1 \approx_l NB'_2$ is trivial since $NB_i = NB'_i$.

Assign-NB: $v \Leftarrow e$**.**

Since non-blocking assignment does not change σ or AS, $\sigma'_1 \approx_l \sigma'_2$ and $AS'_1 \approx_l AS'_2$ trivially.

Let $\Gamma \vdash e \dashv \tau$ and (v, n_i, τ_i) be the assignment generated for NB_i.

First, consider the case when $\tau_1 \sqsubseteq l$.

By the typing rule for assignment, $\tau_{\sigma_1} \sqsubseteq \tau_1 \sqsubseteq l$. By Lemma 8, $n_1 = n_2$. So $\sigma'_1[v] = \sigma'_2[v]$. Similar to blocking assingment, we can prove $\tau_2 \sqsubseteq l$. Therefore $NB'_1 \approx_l NB'_2$.

Second, consider the case when $\tau_1 \not\sqsubseteq l$.

It must be that $\tau_2 \not\sqsubseteq l$ as well (otherwise, $\tau_1 \sqsubseteq l$). Therefore $NB'_1 \approx_l NB'_2$.

Seq: $c_1; c_2$**.**

This holds by the induction hypothesis and transitivity of low-equivalence.

If: if $(e)\,c_1$ **else** c_2**.**

Let $\Gamma \vdash e \dashv \tau$.

First, consider the case where $\tau_{\sigma_1} \sqsubseteq l \wedge \tau_{\sigma_2} \sqsubseteq l$.

In this case, both configurations execute c_1 and by the induction hypothesis $\sigma'_1 \approx_l \sigma'_2$.

Next, consider the case where $\tau_{\sigma_1} \not\sqsubseteq l \wedge \tau_{\sigma_2} \not\sqsubseteq l$ and each configuration executes a different branch (i.e., $\langle \sigma_i, e \rangle \Downarrow n_i \wedge n_1 \neq n_2$).

Let $\texttt{assign}(c^i) = (AS'_i - AS_i) \cup (NB'_i - NB_i)$.

Consider the typing rule for if statements: $\mathcal{M}, \Gamma, pc \vdash c$. By confinement (Lemma 11), $\forall (v_i, n_i, \tau_i) \in \texttt{assign}(c^i)$, $pc \sqcup \tau_{\sigma_i} \sqsubseteq \tau_i$. Since $\tau_{\sigma} \not\sqsubseteq l$, $\tau_{i\,\sigma_i} \not\sqsubseteq l$, and thus any new assignments from c^i are low-equivalent, thus $AS'_1 \approx_l AS'_2$. Simiarly, for all variables assigned in both executions ($\texttt{assign}(c^1) \cap \texttt{assign}(c^2)$), σ'_1 and σ'_2 are low-equivalent.

Any $(v, n, \tau_1) \in \texttt{assign}(c^1) - \texttt{assign}(c^2)$ must not be definitely-assigned, and by the typing rule for assignments: $pc \sqcup \tau_{\sigma_1} \sqsubseteq \Gamma(v)$. Therefore, $\Gamma(v)_{\sigma_1} \not\sqsubseteq l$ and by low-equivalence, $\Gamma(v)_{\sigma_2} \not\sqsubseteq l$. Since v is not assigned to in c^2, $\sigma_2[v] = \sigma'_2[v]$ and $\Gamma(v)_{\sigma_2} \not\sqsubseteq' l$. Therefore, σ'_1 and σ'_2 agree on these variables. The reverse case is symmetric.

For any variables not assigned in either c^i, low-equivalence is also preserved.

Lastly, when $\tau_{\sigma_1} \not\sqsubseteq l \wedge \tau_{\sigma_2} \not\sqsubseteq l$ but both execute the same branch, the induction hypothesis still proves preservation of low-equivalence. Note (by previously described logic) that $\tau_{\sigma_1} \sqsubseteq l \iff \tau_{\sigma_2} \sqsubseteq l$ so there are no other cases.

□

Timing-Sensitive Noninterference. Next, we state timing-sensitive form of noninterference at the thread-level semantics for well-formed circuits.

We write $\langle\sigma, \text{Prog}\rangle \hookrightarrow \mathcal{T}$ as shorthand for: $\langle 0, \sigma, \emptyset, \emptyset, \emptyset\rangle$ producing the sequence of observations $\mathcal{T}$.

Traces are low-equivalent if they are element-wise low-equivalent; the timestamps must be equal and the environments must be low-equivalent.

Theorem B.3 (Timing-Sensitive Noninterference)**.**

$$\vdash \Gamma \wedge \vdash \mathit{Prog} \implies \sigma_1 \approx_l \sigma_2 \wedge \langle\sigma_1, \mathit{Prog}\rangle \hookrightarrow \mathcal{T}_1 \wedge \langle\sigma_2, \mathit{Prog}\rangle \hookrightarrow \mathcal{T}_2 \implies \mathcal{T}_1 \approx_l \mathcal{T}_2$$

The proof of this theorem exactly follows that of Zhang et al. [138], except for an intermediate lemma (Lemma 15), which we prove below.

First, we extend the *apply* function to clear the values of combinational state. Since we only use apply when register values are updated and we know the design is free of inferred latches (by well-formedness), no "unknown" values are ever read in a command. This is equivalent to saying that all combinational values are written before they are read in any given clock cycle.

Definition B.3 (Delayed Assignment Application)**.**

$$\frac{\begin{array}{c}\forall v_c \in \texttt{com}\,.\,\sigma'[v_c] = \bot \\ \forall v_s \in \texttt{seq}\,.\,\sigma'[v_s] = \texttt{foldLeft}(\sigma[v_s], NB)(\lambda((v,n),o)\,.\,v = v_s\,?\,n : o)\end{array}}{\texttt{apply}(\sigma, NB) = \sigma'}$$

This is the same as the definition of apply from the original paper for sequential variables: a variable's value remains unchanged unless there is a pending write and, in the case of multiple writes to a given register, the final write in the list is used as the new value. write to a register), except all combinational values are overwritten with an empty value. For combinational variables, we overwrite their value with an empty or unknown value, $\bot$.

Finally, we prove that register assignments preserve low equivalence of environments.

Lemma B.1 (Delayed Assignment (Lemma 15))**.**

$$NB_1 \approx_l NB_2 \land \sigma_1 \approx_l \sigma_2 \implies apply(\sigma_1, NB_1) \approx_l apply(\sigma_2, NB_2)$$

For simplicity of this proof, we assume there are no duplicate writes to the same location as the determinism assumption guarantees that race conditions should not influence which write will be last in the sequence.

Proof. Let the first entry in NB_i be denoted (v_i, n_i, τ_i). Let $\text{apply}(\sigma_i, NB_i) = \sigma'$. Note that the original low-equivalence of updates, $NB_1 \approx_l NB_2$, is defined with respect to which variables *will have low labels after application* (i.e., with respect to σ'_i).

By induction on the length of NB_1 and NB_2 we first show that σ'_1 and σ'_2 agree on the *values* of all low variables.

First, when $\tau_1 \sqsubseteq l$ and $\tau_2 \sqsubseteq l$, it must be true that $v_1 = v_2$ and $n_1 = n_2$ by the definition of low-equivalence. Thus $\sigma'[v_1] = \sigma'[v_2]$. By induction, all other writes to low variables agree on their new values.

Second, when $\tau_1 \not\sqsubseteq l$, the write does not affect the value of any low variables by definition.

Third, when a variable is *not written* it retains its old value. By our program typing constraint, Definition B.2, any sequential variables which are not written will have a new label that is no less restrictive than its label before the other non-blocking writes are applied.

Let $\tau_v = \Gamma(v) \downarrow_{\sigma}$ and $\tau'_v = \Gamma(v) \downarrow_{\sigma'}$. For sequential variables not written by any non-blocking write, it must be that $\tau_v \sqsubseteq \tau'_v$ in any environment and thus $\tau'_v \sqsubseteq l \implies \tau_v \sqsubseteq l$. Finally, low-equivalence of the starting environments implies that both new environments agree on the value of these low variables.

Lastly, combinational variables (which are not written by any non-blocking event) are all set to $\perp$ by the semantics of apply and thus both environments agree on the values of all combinational variables.

Agreement on Low Labels. We have shown that σ'_1 and σ'_2 agree on the values of all low variables. It immediately follows, by the well-formedness assumption that all free variables in a variable's type must flow to the variable itself, that all variables with low labels also have exactly the same label values.

$\square$

B.2.3 Erasure Guarantees

The main novel result for SpecVerilog is erasure.

We use the notation σ_i to refer to the i^{th} environment of the trace $\mathcal{T}$.

Theorem B.4 (End-To-End Erasure)**.**

$$\vdash \Gamma \wedge \vdash \mathit{Prog} \wedge \langle \sigma, \mathit{Prog} \rangle \hookrightarrow \mathcal{T} \implies$$

$$\forall i, j, l.\, \mathit{Let}\ \sigma'_i = \{\, v \mapsto n \mid n =$$

$$(\forall (e(\vec{y}), b) \in \mathit{eraseTo}(\Gamma(v)).\, b \,{}_{\sigma_i}\!\not\sqsubseteq l \wedge \sigma_j \vDash e(\sigma_i[\vec{y}]))\ ?\ \bot\ :\ \sigma_i[v]\,\}$$

$$i \leq j \wedge \langle \sigma'_i, \mathcal{C} \cup \mathcal{S}, \emptyset, \emptyset \rangle \xrightarrow{\ ...(j+1,\sigma'_{j+1})\ }{}^{j-i+2} \langle ... \rangle \implies \sigma'_{j+1} \approx_l \sigma_{j+1}$$

Note that environments produced by the trace are *end of cycle* values and thus we have a complex stepping rule in the above definition. If, by the end of cycle j, we have discovered that a variable from cycle i must be erased, then we can re-execute cycle i and any intermediate cycles, execute cycle j and then we should see no influence from the variable below the erasure level, l. We slightly abuse the notation for thread operational semantics to indicate that enough steps are taken to produce $j - i + 2$ trace events, the last of which is $(j + 1, \sigma'_{j+1})$.

We need $(j + 1) - i + 1 = j - i + 2$ events since the first event produced will be for cycle i. Consider the case where $i = j$; we need to produce 2 events to capture the new register values computed in cycle i and applied in cycle $i + 1$.

Definitions and Lemmas. We prove a few useful lemmas before tackling end-to-end erasure.

A key lemma is the relationship between the eraseTo function and the may-flows-to relation. If any label must be erased above a given level, then any label which that one flows to also must be erased, or is already above the erasure level.

Lemma B.2 (Erasure Monotonicity).

$$\forall \tau_1, \tau_2 . \tau_1 \sqsubseteq \tau_2 \wedge (e(\vec{y}), b) \in \mathit{eraseTo}(\tau_1) \implies$$

$$b \sqsubseteq \tau_2 \quad \vee \quad (e'(\vec{y'}), b') \in \mathit{eraseTo}(\tau_2) \wedge (e(\vec{y}) \implies e'(\vec{y'})) \wedge b \sqsubseteq b'$$

Proof. By induction on the structure of τ_1,τ_2.

For τ_1 the only cases of interest are when it has an erasure condition:

First, $\tau_1 = b_1 \,^{e(\vec{y})}\!\nearrow b$ and its erasure conditions are $(c, b) \in$ eraseTo(τ_1) where $c = e(\vec{y})$.

$\tau_2 = l$**.**

The only typing rule allowing $\tau_1 \sqsubseteq \tau_2$ requires that $b \sqsubseteq \tau_2$.

$\tau_2 = f(n)$**.**

By the same reasoning as above.

$\tau_2 = b'_1 \,^{e'(\vec{y'})}\!\nearrow b'$**.**

By the typing rule for two erasure policies, $c \implies e'(\vec{y'}) \wedge b \sqsubseteq b'$. $(e'(\vec{y'}), b') \in$ eraseTo(τ_2), and therefore satisfies our lemma.

$\tau_2 = \tau_l \sqcup \tau_r$.

By the typing rule for join, $\tau_1 \sqsubseteq \tau_l \vee \tau_1 \sqsubseteq \tau_r$.

If $\tau_1 \sqsubseteq \tau_l$, by induction either $b \sqsubseteq \tau_l$, in which case $b \sqsubseteq \tau_2$, or τ_l must have an erasure condition satisfying the lemma: (c', b'). By the definition of eraseTo, $(c', b' \sqcup \tau_r) \in \text{eraseTo}(\tau_2)$. This erasure condition also satisfies the lemma requirements since $b \sqsubseteq b' \sqsubseteq (b' \sqcup \tau_r)$.

The case where $\tau_1 \sqsubseteq \tau_r$ is symmetric.

$\tau_2 = \tau_l \sqcap \tau_r$.

By the typing rule for meet, $\tau_1 \sqsubseteq \tau_l \wedge \tau_1 \sqsubseteq \tau_r$. By the induction hypothesis both τ_l and τ_r must indiviually satisfy the lemma.

If neither τ_l nor τ_r have erasure conditions, then it must be that $b \sqsubseteq \tau_l \wedge b \sqsubseteq \tau_r$, which implies that $b \sqsubseteq \tau_l \sqcap \tau_r = \tau_2$.

If τ_l has an erasure condition, (c', b'), but τ_r does not, then τ_2 has $(c', b' \sqcap \tau_r)$. By the above logic, $b \sqsubseteq \tau_r$ and $b \sqsubseteq b'$ therefore $b \sqsubseteq b' \sqcap \tau_r$.

The reverse case is symmetric.

If both τ_l and τ_r have erasure conditions, (c_i, b_i), then $b \sqsubseteq b_i \wedge c \implies c_i$. In this case $c \implies c_1 \wedge c_2$ and $b \sqsubseteq b_1 \sqcap b_2$.

Next, we consider when $\tau_1 = \tau_l \sqcup \tau_r$ which has potential erasure conditions: $(c_l, b_l \sqcup \tau_r), (c_r, b_r \sqcup \tau_l), (c_l \wedge c_r, b_l \sqcup b_r)$

$\tau_2 = l$.

By the typing rule for join, $\tau_l \sqsubseteq \tau_2 \wedge \tau_r \sqsubseteq \tau_2$; therefore by induction $b_l \sqsubseteq l$ and $b_r \sqsubseteq l$. Furthermore, since $b_l \sqsubseteq l \wedge b_r \sqsubseteq l$ it is also true that $b_l \sqcup b_r \sqsubseteq l$.

$\tau_2 = f(n)$.

Same reasoning as above.

$\tau_2 = \tau_{2l} \sqcup \tau_{2r}$.

Either $\tau_l \sqsubseteq \tau_{2l}$ or $\tau_1 \sqsubseteq \tau_{2r}$.

By the induction hypothesis, if $\tau_1 \sqsubseteq \tau_{21}$, then for any $(c, b) \in \text{eraseTo}(\tau_1)$, there exists $(c', b') \in \text{eraseTo}(\tau_{21})$ such that $c \implies c' \wedge b \sqsubseteq b'$ or $b \sqsubseteq \tau_{21}$. It must be the case that $(c', b' \sqcup \tau_{2r}) \in \text{eraseTo}(\tau_2)$ (which satisfies our lemma statement) or $b \sqsubseteq \tau_2$ (by the typing rule of join).

The reverse case ($\tau_1 \sqsubseteq \tau_{2r}$) is symmetric, and since at least one case must be true the lemma holds.

$\tau_2 = \tau_{2l} \sqcap \tau_{2r}$.

By the typing rule for meet, $\tau_l \sqsubseteq \tau_{21} \wedge \tau_1 \sqsubseteq \tau_{2r}$.

By the induction hypothesis, if $\tau_l \sqsubseteq \tau_{2l}$, then we have the same assumptions as in the previous case: $(c', b') \in \text{eraseTo}(\tau_{2l}) \wedge c \implies c' \wedge b \sqsubseteq b'$ or $b \sqsubseteq \tau_{2l}$.

The same holds for τ_{2r}: $(c'', b'') \in \text{eraseTo}(\tau_{2r}) \wedge c \implies c'' \wedge b \sqsubseteq b''$ or $b \sqsubseteq \tau_{2r}$.

If there is an erasure condition in τ_{2l}, but not in τ_{2r}, $(c', b' \sqcap \tau_{2r} \in \text{eraseTo}(\tau_2)$ and this satisfies our lemma since $b \sqsubseteq b'$ and $b \sqsubseteq \tau_{2r}$ so $b \sqsubseteq b' \sqcap \tau_{2r}$.

The reverse case is symmetric.

If there is an erasure condition in both τ_{21} and τ_{2r}, then $(c' \wedge c'', b' \sqcap b'') \in \text{erasetTo}(\tau_2)$, which satisfies our lemma since $c \implies c' \wedge c \implies c''$ and $b \sqsubseteq b' \wedge b \sqsubseteq b''$.

Next, we consider when $\tau_1 = \tau_l \sqcap \tau_r$, and has erasure conditions $(c_l, b_l \sqcap \tau_r), (c_r, b_r \sqcap \tau_l), (c_l \wedge c_r, b_l \sqcap b_r)$.

$\tau_2 = l$**.**

By the typing rule for meet, $\tau_l \sqsubseteq l \vee \tau_r \sqsubseteq l$.

If $\tau_l \sqsubseteq l$ then, by the induction hypothesis, $b_l \sqsubseteq l$. Therefore: $b_l \sqcap \tau_r \sqsubseteq l$ and $b_r \sqcap \tau_l \sqsubseteq l$ and $b_1 \sqcap b_r \sqsubseteq l$.

The other case is symmetric.

$\tau_2 = f(n)$**.**

This is the same as above.

$\tau_2 = \tau_{2l} \sqcup \tau_{2r}$**.**

By the typing rule for join: $\tau_1 \sqsubseteq \tau_{2l} \vee \tau_1 \sqsubseteq \tau_{2r}$.

If $\tau_1 \sqsubseteq \tau_{2l}$ then, by the induction hypothesis, for each erasure condition in τ_1 either $b \sqsubseteq \tau_{2l}$ or $(c', b') \in \text{eraseTo}(\tau_{2l})$ where $c \implies c' \wedge b \sqsubseteq b'$.

If $b \sqsubseteq \tau_{2l}$ then $b \sqsubseteq \tau_2$ and so that case satisfies the lemma. If $(c', b') \in \text{eraseTo}(\tau_{21})$ then $(c', b' \sqcup \tau_{2r}) \in \text{eraseTo}(\tau_2)$, which also satisfies the lemma, since $b \sqsubseteq b' \sqsubseteq b' \sqcup \tau_{2r}$.

The reverse case is symmetric.

$\tau_2 = \tau_{2l} \sqcap \tau_{2r}$.

By the typing rule for meet introduction: $\tau_1 \sqsubseteq \tau_{21} \wedge \tau_1 \sqsubseteq \tau_{2r}$.

By the induction hypothesis, for each of τ_{2i} either: $b \sqsubseteq \tau_{2i}$ or $(c_i, b_i) \in \text{eraseTo}(\tau_{2i}) \wedge c \implies c_i \wedge b \sqsubseteq b_i$.

If neither τ_{2i} has an erasure condition, then $b \sqsubseteq \tau_{2l} \wedge b \sqsubseteq \tau_{2r}$ and therefore $b \sqsubseteq \tau_2$.

If only one has an erasure condition (say τ_{2l}), then $b \sqsubseteq \tau_{2r}$ and $(c_l, b_l \sqcap \tau_{2r}) \in \text{eraseTo})(\tau_2)$. $c \implies c_l$ and $b \sqsubseteq b_l \sqcap \tau_{2r}$ so we satisfy the lemma.

If both have an erasure condition, then $(c_l \wedge c_r, b_l \sqcap b_r) \in \text{eraseTo})(\tau_2)$ which satisfies our lemma. $\square$

Next, we define a weaker notion of observational equivalence, that allows disagreement on some variables. We define a general notion of *projection equivalence* to capture agreeing on the values of a set of variables:

Definition B.4 (Projection Equivalence). *Let* $\sigma_i|^{\mathcal{V}} = \{\, v \mapsto \sigma_i[v] \mid v \in \mathcal{V} \,\}$

$$\sigma_1 \approx_l^{\mathcal{V}} \sigma_2 \iff \sigma_1|^{\mathcal{V}} \approx_l \sigma_2|^{\mathcal{V}}$$

We extend the definition of projections and projection equivalence to write sets (*AS* and *NB*) in the obvious way. The projections erase entries from the lists for variables that are not included in the projection.

We consider permanent projections, the set of variables that will never be erased above a given observation level in a trace of environments.

Definition B.5 (Permanent Projection)**.** *The permanent projection of an environment up to a level l, $\sigma|^{P_l:\mathcal{T}}$, is the subset of the environment for which variables do not have an erasure policy specifying erasure above level l, or whose erasure conditions are not ever fulfilled in a given trace.*

$$\sigma|^{P_l:\mathcal{T}} = \{\, v \mapsto \sigma[v] \mid \forall(e(\vec{y}), b) \in \mathit{eraseTo}(\Gamma(v)).b \,{}_{\sigma}\!\sqsubseteq l \vee (\forall \sigma_i \in \mathcal{T}, \sigma_i \not\models e(\sigma_i[\vec{y}]))\,\}$$

For the sake of simplifying the following proofs, we assume that erasure conditions only contain sequential variables so that the set of erased variables does not change within a given cycle; however, we do not believe this is necessary for the following theorems to hold.

Executing commands preserves equivalence of permanent projections.

Lemma B.3 (Single-Command Permanent Equivalence)**.**

$$\sigma_1 \approx_l^{P_1:(\sigma_1)} \sigma_2 \wedge \sigma_i \Downarrow \sigma_i' \implies \sigma_1' \approx_l^{P_1:(\sigma_1)} \sigma_2'$$

Proof. By induction on the structure of commands.

Assign: $v = e$**.**

Let $\Gamma \vdash e_i \dashv \tau_i'' \wedge \tau_i'' \downarrow_{\sigma_i} = \tau_i$. Let $\Gamma(v) \downarrow_{\sigma_i'} = \tau_i'$.

First, if e only contains variables from the permanent projection, then the value in v will be low-equivalent, by Theorem B.2, single-command noninterference.

Additionally, whether v is in the permanent projection of σ_i' is only a function of $\sigma_i'[v]$, $\Gamma(v)$, and variables that must have been equivalent in σ_1 and σ_2, since they had equivalent permanent projections before the assignment.

Otherwise, the value, n_i, assigned into v may differ in each environment; to preserve permanent equivalence the label of v in both environments must be above l, or must also be outside of the permanent projections.

By the typing rule for assignment $\tau_i \sqsubseteq \tau_i'$; therefore by Lemma B.2, when evaluated in the final environments, v, is either outside the permanent projection $((\_, b') \in \text{eraseTo}(\Gamma(v)) \wedge b' \not\sqsubseteq l)$ *or* is in the high set $(\Gamma(v) \not\sqsubseteq l)$. Which of these two statements hold must be the same in both environments since it is a function only of $\Gamma(v)$.

By the above logic, the accumulated write sets (AS_1' and AS_2') will also be permanent equivalent. The nonblocking write sets are unchanged and thus trivially equivalent.

Assign-NB: $v \Leftarrow e$**.**

This follows exactly the same logic as in blocking assignment, except we are only arguing about the permanent equivalence of NB since neither σ nor AS change.

Seq: $c_1; c_2$.

By the induction hypothesis and transitivity of permanent equivalence.

If: **if** $(e)\, c_1$ **else** c_2.

First, as with assignment, if all of the variables referenced in e are in the permanent projection, then both configurations will execute the same branch and by the induction hypothesis will be permanent equivalent. This logic also holds if the two configurations happen to execute the same branch.

Otherwise, the two environments diverge and the configurations execute different branches.

The typing rule for assignment ensures that the label of e is captured by the pc, and by the typing rules for assignment, we can appeal to the same logic as above. All writes that occur in either configuration and in either branch must be to variables that will be erased or in the high set of labels. Therefore, erasure equivalence is still preserved. □

Permanent equivalence straightforwardly extends to threads (the traces are element-wise permanent equivalent). Essentially, this relies on the same logic as preserving non-interference across clock cycle boundaries when applying register updates. Thus, we omit a detailed proof:

Lemma B.4 (Thread Permanent Equivalence)**.**

$$\sigma_1 \approx_l^{P_1:(\sigma_1)} \sigma_2 \wedge \langle \sigma_i, Prog \rangle \hookrightarrow \mathcal{T}_i \implies \mathcal{T}_l \approx_l^{P_1:(\sigma_1)} \mathcal{T}_2$$

Now we move on to actually erasing variables; we show that, if a variable's erasure condition is fulfilled in the current cycle, then it has no impact on the set of variables in the next cycle which are below its erasure level.

We define this with a new projection, which we call the *retained projection* and it consists of all variables that don't need to be erased this cycle:

$$\sigma|^{R_l} = \{\, v \mapsto \sigma[v] \mid \forall (e(\vec{y}), b) \in \text{eraseTo}(\Gamma(v)).\, b \,{}_{\sigma}{\sqsubseteq}\, l \vee \sigma \nvDash e(\vec{y}) \,\}$$

Now we define erasure equivalence: when two environments have equivalent retained projections, they will preserve that equivalence *and* will only accrue low-equivalent updates to registers.

Lemma B.5 (Single-Cycle Command Erasure Equivalence)**.**

$$\forall \sigma_1, \sigma_2, l.\ \sigma_1 \approx_l^{R_1} \sigma_2 \wedge AS_1 \approx_l^{R_1} AS_2 \wedge NB_1 \approx_l NB_2 \wedge \sigma_i \Downarrow \sigma_i' \implies$$
$$\sigma_1' \approx_l^{R_1} \sigma_2' \wedge AS_1' \approx_l^{R_1} AS_2' \wedge NB_1' \approx_l NB_2'$$

Proof. By induction on the structure of commands.

Assign: $v = e$**.**

The only case not covered by Lemma B.3 is when e has an erasure condition in its type and $b \,{}_{\sigma}{\not\sqsubseteq}\, l \wedge \sigma \nvDash e(\vec{y})$; in this case the two assignments have to agree on the result, but they will also agree on the value of e and therefore the new label and value of v.

For NB this is trivial. This statement has no effect on the accumulated non-blocking writes.

Assign-NB: $v \Leftarrow e$**.**

Let $\Gamma \vdash e \dashv \tau$.

This only updates *NB* and so retained equivalence is maintained on σ and *AS*.

First, when the two configurations agree on the value of e, (i.e., all of its variables are in the retained projection), then low-equivalence of *NB* is maintained by noninterference.

Otherwise, it must be the case that $(e(\vec{y}), b) \in \text{eraseTo}(\tau) \wedge \sigma \vDash e(\vec{y}) \wedge b\ {}_{\sigma}{\not\sqsubseteq}\ l$: at least some variable in e has its erasure condition fulfilled and thus the two configurations may compute different values of e.

If $(e'(\vec{y'},) \in \Gamma(v)$, by the typing rules from non-blocking assign, the erasure condition must not be fulfilled. By Lemma B.2, $\sigma \vDash e(\vec{y}) \implies \sigma \vDash e'(\vec{y'})$ and therefore v must not have an erasure condition if the assignment is well-typed.

The only other way for this to be well-typed is if $\Gamma(y) = \tau' \wedge b\ {}_{\sigma}{\sqsubseteq}_{\textbf{next}}\ \tau'$, which implies $\tau' \downarrow_{\sigma'} \not\sqsubseteq\ l$. This in turn implies low-equivalence of the update to NB'.

Seq: $c_1; c_2$**.**

By the induction hypothesis and transitivity of equivalence.

If: if $(e)\, c_1$ **else** c_2**.**

As with the other theorems, the only case that need be considered is when e needs to be erased to a level *above* l, and the two configurations execute different commands.

Since the *pc* captures the erasure condition that must be present in the type of e (as argued above), there will be no non-blocking writes to any level below l and no blocking writes to the retained projections of σ. □

This straightforwardly extends to threads as well:

Lemma B.6 (Single-Cycle Thread Erasure)**.**

$$\forall \sigma_1, \sigma_2, l. \, \sigma_1 \approx_l^{R_1} \sigma_2 \wedge \langle t, \sigma_i, Prog, \vec{c}, \mathcal{B}, \emptyset \rangle \hookrightarrow_{::} (t+1, \sigma_i') :: \mathcal{T}_i \implies \sigma_1' \approx_l \sigma_2'$$

We omit a complete proof since this follows almost directly from Lemma B.5 and the well-formedness assumptions of race-freedom for programs.

We only prove the equivalent of Lemma B.7 for erasure:

Lemma B.7 (Delayed Assignment After Erasure)**.**

$$NB_1 \approx_l NB_2 \wedge \sigma_1 \approx_l^{R_1} \sigma_2 \implies apply(\sigma_1, NB_1) \approx_l apply(\sigma_2, NB_2)$$

Proof. This follows almost directly from the same logic as Lemma B.7, the only change is for unwritten sequential variables.

In this case, the only unwritten sequential variables whose values are not low-equivalent are those with an erasure condition that is fulfilled in the current cycle.

By the typing constraint in Definition B.2, the label of these variables must flow to their next cycle label. If their erasure condition is fulfilled, then, by Lemma B.2 their next cycle label is either in the high set (and thus low-equivalence is preserved) or its erasure condition is also fulfilled (and thus is not well-typed). □

Finally, we prove that well-typed programs provide end-to-end erasure.

Theorem B.4 (End-To-End Erasure)**.**

$$\vdash \Gamma \wedge \vdash \mathit{Prog} \wedge \langle \sigma, \mathit{Prog} \rangle \hookrightarrow \mathcal{T} \implies$$

$$\forall i, j, l.\, \mathit{Let}\ \sigma'_i = \{\, v \mapsto n \mid n =$$

$$(\forall (e(\vec{y}), b) \in \mathit{eraseTo}(\Gamma(v)).\, b \mathrel{{}_{\sigma_i}{\not\sqsubseteq}} l \wedge \sigma_j \vDash e(\sigma_i[\vec{y}]))\ ?\ \bot\ :\ \sigma_i[v] \,\}$$

$$i \leq j \wedge \langle \sigma'_i, \mathcal{C} \cup \mathcal{S}, \emptyset, \emptyset \rangle \xrightarrow{...(j+1, \sigma'_{j+1})}{}^{j-i+2} \langle ... \rangle \implies \sigma'_{j+1} \approx_l \sigma_{j+1}$$

Proof.

Base case: $i = j$**.**

In this case, end-to-end erasure reduces to single-cycle thread erasure (Lemma B.6), since $\sigma_j \approx_l^{R_1} \sigma'_j$.

Otherwise: $i < j$**.** Without loss of generality, consider an i such that $\forall k.\, i \leq k < j \wedge \sigma_k \nvDash e(\sigma_i[\vec{y}])$ (i.e., there is no intermediate cycle between i and j when the erasure condition holds).

Let $\mathcal{T}_{ij}$ be the subset of the execution trace between σ_i and σ_j (inclusive). In this case, $\sigma_i \approx_l^{P_1:\mathcal{T}_{ij}} \sigma'_i$, which by Lemma B.4 is preserved across cycles, so $\sigma_{i+1} \approx_l^{P_1:\mathcal{T}_{ij}} \sigma'_{i+1}$: inductively, $\sigma_j \approx_l^{P_1:\mathcal{T}_{ij}} \sigma'_j$.

At this point, since the erasure condition is true in cycle j, permanent equivalence in cycle j implies retained equivalence, which again reduces to single-cycle thread erasure. $\square$

BIBLIOGRAPHY

[1] IEEE standard for Verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–560, 2006.

[2] Johan Agat. Transforming out timing leaks. In *27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, January 2000.

[3] Sam Ainsworth. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 592–606. Association for Computing Machinery, 2021.

[4] Sam Ainsworth and Timothy M Jones. Muontrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[5] Jos Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. Cryptology ePrint Archive, Report 2015/1241, 2015. https://eprint.iacr.org/2015/1241.

[6] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 147–154. IEEE, 2017.

[7] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1691–1696. IEEE, 2017.

[8] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, pages 297–307. ACM, 2010.

[9] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[10] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221. IEEE, 2012.

[11] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. *International Journal of Information Security*, 3(1):42–60, 2004.

[12] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, pages 73–78. IEEE, 2002.

[13] Richard Barry et al. FreeRTOS. *Internet, Oct*, 4, 2008.

[14] 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*, CCS '14, pages 1267–1279, New York, NY, USA, 2014. ACM.

[15] Andrew Bedford, Stephen Chong, Josée Desharnais, Elisavet Kozyri, and Nadia Tawbi. A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version). *Computers & Security*, 71:114–131, 2017.

[16] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *26th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1046–1060, 2021.

[17] Daniel J Bernstein. Cache-timing attacks on AES.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The Gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[19] Gregor V Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, 31(03):223–231, 1982.

[20] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of Bluespec: A core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257, 2020.

[21] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[22] Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 65–79. IEEE, 2014.

[23] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *20th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 289–301. ACM, 2015.

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

[25] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 913–926, 2020.

[26] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 1875–1891. ACM, October 2017.

[27] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019.

[28] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, and Adam Chlipala. Kami: A platform for high-level parametric hardware specification and its modular verification. In *Int'l Conf on Functional Programming (ICFP)*, pages 1–30, 2017.

[29] S. Chong and A.C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, 2005.

[30] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 241–254, June 2005.

[31] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *IEEE Computer Security Foundations Symp. (CSF)*, pages 98–111, June 2008.

[32] Cong, Jason and Liu, Bin and Neuendorffer, Stephen and Noguera, Juanjo and Vissers, Kees and Zhang, Zhiru. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[33] Embedded Microprocessor Benchmark Consortium. Eembc. `https://www.eembc.org/coremark/index.php`, 2021. Accessed: 2021-08-01.

[34] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[35] L BLACK DAVID, B GOLUB DAVID, and P JULIN DANIEL. Microkernel operating system architecture and mach. *Journal of information processing*, pages 11–30, 1992.

[36] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y. Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. SecChisel framework for security verification of secure processor architectures. In *8th Int'l Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2019.

[38] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[39] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[40] Dominique Devriese and Frank Piessens. Noninterference through secure

multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[41] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–422, 2020.

[42] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[43] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Mller, Jrg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Transactions on Computers*, 72(1):222–235, 2023.

[44] Andrew Ferraiuolo. Security results for SIRRTL, a hardware description language for information flow security, 2017.

[45] Andrew Ferraiuolo, Weizhe Hua, Andrew C. Myers, and G. Edward Suh. Secure information flow verification with mutable dependent types. In *54th Design Automation Conference (DAC)*, June 2017.

[46] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2017.

[47] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. Hyperflow: A processor architecture for nonmalleable, timing-safe information-flow security. In *25th ACM Conf. on Computer and Communications Security (CCS)*, October 2018.

[48] Jacob Fustos, Michael Bechtel, and Heechul Yun. SpectreRewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, pages 117–126, 2020.

[49] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, pages 1:1–1:9, New York, NY, USA, 2018. ACM.

[50] 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)*, pages 1–19. IEEE, 2020.

[51] Marco Guarnieri, Boris Kpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1868–1883, 2021.

[52] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[53] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[54] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Int'l Symp. on Low Power Electronics and Design*, 2007.

[55] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[56] Steven F Hoover. Timing-abstract circuit design in transaction-level Verilog. In *2017 IEEE Int'l Conf. on Computer Design (ICCD)*, pages 525–532. IEEE, 2017.

[57] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 7(3):1067–1080, 2012.

[58] Sebastian Hunt and David Sands. Just forget it–the semantics and enforcement of information erasure. In *European Symposium on Programming*, 2008.

[59] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[60] Zhenghong Jiang, Hanchen Jin, G Edward Suh, and Zhiru Zhang. Designing secure cryptographic accelerators with information flow enforcement: A case study on aes. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[61] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *2018 ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 127–136, 2018.

[62] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing testing with formal verification in Intel® CoreTM i7 processor execution engine validation. In *Int'l Conf. on Computer Aided Verification (CAV)*, pages 414–429, 2009.

[63] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *2011 IEEE Symposium on Security and Privacy*, pages 413–428. IEEE, 2011.

[64] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 1–17. Springer, 2009.

[65] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12:221–230, 2008.

[66] Elisavet Kozyri, Fred B Schneider, Andrew Bedford, Josée Desharnais, and Nadia Tawbi. Beyond labels: Permissiveness for dynamic information flow enforcement. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 351–35115. IEEE, 2019.

[67] Kernel page-table isolation. Wikipedia, 2022.

[68] Daniel Kroening and Wolfgang J. Paul. Automated pipeline design. In *38th annual Design Automation Conf. (DAC)*, pages 810–815, 2001.

[69] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST)*, September 2003.

[70] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[71] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[72] 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.

[73] Gai Liu, Joseph Primmer, and Zhiru Zhang. Rapid generation of high-quality RISC-V processors from functional instruction set specifications. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[74] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[75] Marc Löw. Overview of Meltdown and Spectre patches and their impacts. *Advanced Microkernel Operating Systems*, page 53, 2018.

[76] ARM Ltd. ARM Security Technology: Building a Secure System using TrustZone Technology, 2009.

[77] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 635–646, Washington, DC, USA, 2014. IEEE Computer Society.

[78] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLcheck: Verifying the memory consistency of RTL designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 463–476, New York, NY, USA, 2017. ACM.

[79] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[80] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In Ruby B. Lee and Weidong Shi, editors, *Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[81] 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*, New York, NY, USA, 2022. Association for Computing Machinery.

[82] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[83] Andrew C Myers and Andrew C Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[84] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[85] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[86] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 393–407, 2020.

[87] R. Nikhil. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *ACM and IEEE Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 69–70, 2004.

[88] Eriko Nurvitadhi, James C Hoe, Timothy Kam, and Shih-Lien L Lu. Automatic pipelining from transactional datapath specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–454, 2011.

[89] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[90] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.

[91] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In *Int'l Conf. on Computer Aided Verification (CAV)*, pages 42–58, 2016.

[92] Berkeley Architecture Research. Sodor core. `https://chipyard.readthedocs.io/en/dev/Generators/Sodor.html`, 2021. Accessed: 2021-08-01.

[93] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[94] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. What you simulate is what you synthesize: Designing a processor core from C++ specifications. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[95] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[96] Andrei Sabelfeld and David Sands. Dimensions and principles of declas-

sification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE, 2005.

[97] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An" undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[98] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 723–735, 2019.

[99] Jerome H Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

[100] Mostafa Sayahkarajy, E Supriyanto, MH Satria, and Hasri Samion. Design of a microcontroller-based artificial pacemaker: An internal pacing device. In *2017 International Conference on Robotics, Automation and Sciences (ICORAS)*, pages 1–5. IEEE, 2017.

[101] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[102] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: a framework for design and verification of information flow control systems. In *13th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 287–305, 2018.

[103] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016.

[104] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned AES T-tables. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 200–214. Springer, 2013.

[105] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[106] Gordon Stewart, Anindya Banerjee, and Aleksandar Nanevski. Depen-

dent types for enforcement of information flow and erasure policies in heterogeneous data structures. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, 2013.

[107] James E Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W Rhett Davis, Paul D Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, et al. FreePDK: An open-source variation-aware design kit. In *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*, pages 173–174. IEEE, 2007.

[108] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, January 1986.

[109] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, et al. CARAT CAKE: Replacing paging via compiler/kernel cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 98–114, 2022.

[110] G. Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, 2004.

[111] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[112] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[113] Mohit Tiwari, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[114] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[115] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. Solver-Aided Constant-Time Hardware Verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 42–444, 2021.

[116] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symp.*, August 2018.

[117] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic.

[118] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. *arXiv preprint arXiv:2005.00294*, 2020.

[119] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, IEEE(CSFW)*, volume 00, page 156, 06 1997.

[120] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[121] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[122] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-wasm: Type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL):77:1–77:29, January 2019.

[123] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*, pages 675–692, 2019.

[124] Steven JE Wilton and Norman P Jouppi. CACTI: An enhanced cache ac-

cess and cycle time model. *IEEE Journal of solid-state circuits*, 31(5):677–688, 1996.

[125] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, 2014.

[126] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 2021.

[127] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[128] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing.

[129] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data. In *IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*.

[130] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (STT): A formal analysis. *University of Illinois at Urbana-Champaign and Tel Aviv University, Tech. Rep*, 2019.

[131] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019.

[132] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. The cost of software-based memory management without virtual memory. September 2020. `https://arxiv.org/abs/2009.06789`.

[133] F. Zaruba and L. Benini. The Cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[134] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 15–23, June 2001.

[135] Steve Zdancewic and Andrew C Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 29–43. IEEE, 2003.

[136] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[137] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012. ACM.

[138] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for efficient control of timing channels. Technical Report http://hdl.handle.net/1813/36274, Cornell University Computing and Information Science, April 2014.

[139] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 503–516, March 2015.

[140] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. Synthesizing environment invariants for modular hardware verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 202–225. Springer, 2020.

[141] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), March 2007.