Foundational Verification of Hardware Against Timing-Channel-Aware Contracts

by

Stella Lau

B.A., University of Cambridge (2017)
M.Eng., University of Cambridge (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2026

Authored by: Stella Lau
Department of Electrical Engineering and Computer Science
August 3, 2026

Certified by: Adam Chlipala
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Foundational Verification of Hardware Against Timing-Channel-Aware Contracts

by

Stella Lau

Submitted to the Department of Electrical Engineering and Computer Science
on August 3, 2026, in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Abstract

Modern processors can leak secrets not only through what they compute but also through how long they take to compute. These timing channels lie outside the traditional instruction-set contract between hardware and software and remain one of the most persistent challenges in designing secure hardware-software systems: despite years of mitigations, new attacks continue to be discovered. This thesis contributes methodologies for proving, by machine-checked proof, that a hardware design leaks no secrets—through its timing or otherwise—beyond what a timing-channel-aware contract permits. Rather than targeting specific attacks or patching individual mechanisms, this thesis establishes a formal foundation for ruling out entire classes of timing leakage—including those not yet discovered.

We consider the challenge of running security-critical code on the same machine as untrusted code without leaking secrets, decomposing the problem into two parts: (1) attackers should neither be able to extract secrets from, nor interfere with, security-critical code due to being colocated on the same machine; and (2) security-critical code should not itself leak secrets when running independently on its own machine. This thesis contributes two methodologies—MTIsolation, addressing (1), and Granite, addressing (2)—together with their underlying specification technique, *leakage-aware refinement*, which constrains a design's correctness and information-flow dependencies without overspecifying exact timing. MTIsolation specifies strong timing isolation through a model based on "air-gapped machines", in which colocated security domains behave as if each ran on its own machine, and rules out leakage through shared resources by construction. Granite verifies both functional correctness and nonleakage of circuit-level processors against hardware-software leakage contracts; its proofs compose with a certified software static analysis to derive a single theorem about the cycle-by-cycle confidentiality of a hardware-and-software cryptographic implementation, eliminating even the hardware-software contract from the trusted computing base. Both MTIsolation and Granite are mechanized in the Rocq proof assistant and evaluated on synthesizable designs.

Together, MTIsolation and Granite show that hardware support for isolation and confidentiality can be established by machine-checked proof, down to the circuit level and in the presence of timing channels. By elevating timing from an unspecified implementation detail to a first-class, provable component of hardware-software contracts, this thesis charts a principled path toward trustworthy hardware-software systems that are provably secure against timing-channel attacks.

Thesis Supervisor: Adam Chlipala
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am immensely grateful to the many people who supported and inspired me during my PhD.

First and foremost, I thank my advisor, Adam Chlipala. His unwavering support, patience, and encouragement over the years has made working on this thesis a pleasure. I learned a lot from Adam, from taste in research problems to the correct usage of commas and hyphens (I still get it wrong sometimes).

I also thank Mengjia Yan and Frans Kaashoek for serving on my thesis committee. I greatly appreciate all the valuable insights and thoughtful comments they brought to this work. I thank my collaborators, in addition to Adam and Mengjia: Arvind, Thomas Bourgeat, Andres Erbsen, Clément Pit-Claudel, and Yuheng Yang. Working together was incredibly enriching and their unique perspectives shaped the work in this thesis. This thesis was greatly improved thanks to feedback from Aleksandar Krastev, Alexandra Henzinger, Anish Athalye, Derek Leung, Dongjae Lee, Jules Drean, Kyle Hogan, Samuel Gruetter, Srini Devadas, Upamanyu Sharma, the Programming Languages and Verification group, and anonymous reviewers. I thank all my colleagues and friends from the programming languages, computer architecture, systems, and security communities in CSAIL for creating such a wonderful lab atmosphere. Our many conversations shaped my values, both in and outside of research, and made the day-to-day fun.

Finally, I am deeply grateful to my family and friends for their love and support. Some thanks are better said privately. Verifiable commitment (as a Merkle hash) below.

```
755642b05547c263528acdfd9b5a546bcdcac4c5245e3de04c0b3d1d048254d3
```

Contents

Figures and Tables

Chapter 1

Introduction

A recurring goal in secure systems design is to run security-critical code on a machine it shares with untrusted code without leaking secrets. For example: a cloud-computing environment holds one tenant's sensitive medical records or proprietary model weights on a server that runs another tenant's code; a mobile phone stores a user's biometrics and payment credentials alongside hundreds of third-party applications; a web browser runs a user's banking session alongside untrusted JavaScript from a different site; a cryptographic library signs with a private key on a processor shared with an attacker.

We decompose this goal into two parts: *isolation*, such that domains sharing a machine cannot extract secrets from nor interfere with one another due to being physically colocated on the same machine, and *confidentiality*, such that security-critical code does not leak its own secrets even when running independently on its own machine. Software relies on hardware to provide the necessary primitives to support isolation and confidentiality, and yet the traditional contract between hardware and software—the instruction-set architecture, or ISA—provides no such primitives: it specifies *what* a program computes but not *how long* it takes, intentionally allowing the underlying hardware the freedom to implement performance optimizations and, unintentionally, to leak secrets via timing.

The core problem tackled by this thesis is that modern processors can leak secrets not only through what they compute but also through how long they take to compute. These timing channels remain one of the most persistent challenges in designing secure hardware-software systems, and despite years of mitigations, new attacks continue to be discovered. This thesis contributes two methodologies, MTIsolation and Granite, that together show that hardware support for isolation and confidentiality can be established by machine-checked proof, down to the circuit level and in the presence of timing channels. Rather than targeting specific attacks or patching individual hardware mechanisms, this thesis establishes a formal foundation for ruling out entire classes of timing leakage—including those not yet discovered—by proving, with machine-checked proof, that a hardware design leaks no secrets, through its timing or otherwise, beyond what an explicit timing-channel-aware contract permits.

1.1 Timing channels: a tale as old as time

Work on this thesis began in 2018. That same year, it was publicly disclosed [40] that nearly every high-performance processor shipped since the mid-1990s was vulnerable to a new class of attacks called *speculative-execution* attacks, of which Spectre [48] and Meltdown [53] are the seminal examples. These attacks exploit speculative execution, a core processor optimization in which processors guess their way past slow operations[1] such as unresolved branches and pending memory accesses, executing instructions *speculatively* and discarding the work if the guesses are wrong. The ISA guarantees that these misspeculated executions leave no observable side effects on architectural state (registers and memory) but makes no such guarantee for microarchitectural state (the caches and predictors whose purpose is to improve performance), thereby allowing side effects with downstream effects on latency to remain. By influencing the processor to read a secret speculatively and imprint it on the cache, an attacker recovers that secret through a later timing measurement. The attack was devastating because it was not simply a bug but a conceptual design flaw—with broad scope and no immediately satisfying fix—that bypassed the isolation boundaries relied on by system security: between user and kernel [53], between processes [56], between cloud tenants [39], between virtual machines sharing a host [37, 83], and even between a web page and its browser [43–45].

Spectre and Meltdown were not two isolated bugs but the first of a class, and what follows is an ongoing cat-and-mouse game of attack and defense. Each new attack—Foreshadow [18], then the microarchitectural data-sampling attacks [19, 70, 79], and many more—exploited a different speculative mechanism or microarchitectural structure; each was met with mitigations aimed at it; and each mitigation was, in time, demonstrated insufficient by an attack exploiting some structure its designers had never thought to defend. These timing attacks are not mere academic curiosities, but have been shown to be exploitable to extract cryptographic secrets from production cloud environments [39, 91, 92] and widely used web services [45]. Yet for all the alarm caused by these attacks, this cat-and-mouse game around timing channels is not new.

Timing channels were an actively researched, unsolved problem well before 2018. In 1973, Lampson's note on the confinement problem identified the challenge of stopping a program from leaking information through covert channels its designers never intended, among them the performance of a system [50]. By the 1980s, the analysis of covert channels had become a requirement for high-assurance systems [51][2]. The notion that one computation can observe another through nothing but how long it takes was thus understood long before

[1] See https://xkcd.com/1938 for an analogy based on the trolley problem.

[2] The Trusted Computer System Evaluation Criteria (TCSEC, or the Orange Book) required formal analysis of covert storage channels but allowed informal analysis of covert timing channels as formal analysis thereof was "beyond current technology". The Orange Book states, "In the future, use of formal verification will be extended to the source level and covert timing channels will be more fully addressed". The future is now!

timing channels had been exploited in microarchitecture. In 1996, Kocher showed that an attacker could recover secret keys from RSA and other cryptosystems by measuring how long operations take to run [49]; Brumley and Boneh later demonstrated in 2005 that such an attack could succeed against a remote server across a network [17]; Bernstein showed that AES keys can be recovered through cache timing [11]; and in 2006 Osvik, Shamir, and Tromer distilled the technique into the Prime+Probe primitive [64] that, with the later Flush+Reload [87], remains the workhorse of microarchitectural timing attacks to this day. Even the branch predictor, weaponized by Spectre in 2018, was turned into a side channel as early as 2006 [3].

Defenses were developed in response to each of these attacks, but the channels were never truly closed. Developers used the constant-time discipline to protect high-assurance cryptographic implementations from timing attacks by writing code free from secret-dependent branches and memory accesses according to the architectural semantics specified by the ISA, yet Spectre and subsequent attacks revealed that these techniques were insufficient on the microarchitecture of modern hardware [20]; cache channels were answered with partitioning schemes that carved the shared cache into isolated regions [46] but left the many other shared microarchitectural resources (such as the TLB and on-chip interconnects) as open avenues for the same class of contention attack. The defenses were effective within their local scope, but each closed a single channel and as microarchitectures continued to evolve, the cat-and-mouse game continued.

Left unclosed, timing channels thwart attempts at building the next generation of trustworthy systems. The story of Intel SGX tells a cautionary tale about attempting to design secure systems with strong confidentiality guarantees while leaving timing side channels as an open attack surface. Architectures such as Intel SGX are designed to protect *enclaves* in trusted execution environments (TEEs) from a malicious operating system—the confidential computing model for cloud computing that is industry's attempt to shrink the trusted computing base (TCB) so that the cloud provider does not need to be trusted—but they exclude microarchitectural side channels from their threat model [27]. A series of attacks including Foreshadow [18], which applied the speculative-execution attack underlying Meltdown to SGX, decisively dismantled the security objectives of SGX implementations by extracting the core cryptographic keys from enclaves. The successors—AMD SEV-SNP, Intel TDX, ARM CCA—implement mitigations against timing attacks but ultimately leave timing channels unclosed, leaving the door open to numerous attacks breaking isolation and confidentiality guarantees [21, 67]. The existence and continued discovery of these attacks is not surprising, as these systems were fundamentally not designed to be broadly secure against timing channels.

The lesson of these past decades is that ignoring or playing whack-a-mole with timing channels is not a promising path towards building trustworthy systems. The core issue is that the traditional ISA contract justified only through informal reasoning is an unreliable foundation for confidentiality and security. Current hardware-software contracts are either

over-abstracted, specifying what a program computes but saying nothing about how long it takes (and so the microarchitecture beneath it remains free to leak in ways no one ever promised it would not) or *unsound*, promising nonleakage guarantees that the hardware does not uphold (e.g. speculative execution violating traditional constant-time contracts on modern processors). In both cases, a semantic gap exists between the architectural specification and the microarchitectural reality observed through timing. As long as this gap remains, software is left defenseless against timing side channels, and the bedrock of our systems remains shifting sand.

1.2 Goal: ruling out timing leakages in hardware

The goal of this thesis is to close this semantic gap and develop methodologies to make timing-channel security a first-class, machine-checked property of hardware down to the circuit level. We aim to provably restore hardware support for isolation and confidentiality in the presence of timing channels, so that we can build hardware-software systems that do not leak secrets. We require an explicit *hardware-software contract* [38] that states what a program's execution may reveal, through microarchitectural timing or otherwise, together with a means of *proving* that a hardware design adheres to it. As modern microarchitectures are far too intricate to trust on the strength of testing or informal argument, we take the position that the proof should be *machine-checked*—expressed in a mathematical logic and validated by a proof checker—so that its proven properties can be reviewed based on the theorem statement alone without auditing the implementation.

The problem: secure execution on shared machines in the presence of timing channels. We consider the age-old problem of executing security-critical code on shared hardware without leaking secrets; we decompose the problem into two parts, each threatened by a distinct class of timing channel:

1. **Isolation**, threatened by *interdomain* timing channels—those that cross the boundaries between security domains. When mutually distrusting domains (e.g. processes, virtual machines, web pages, cloud tenants, enclaves) run on shared hardware, contention for shared microarchitectural resources such as caches and branch predictors lets one domain observe another's secrets—this is the defining threat of the public cloud, whose economics depend on colocating mutually distrusting tenants on the same hardware. Isolation requires that colocated domains can neither observe nor interfere with one another, including via such channels.

2. **Secret-independent timing**, threatened by *intradomain* timing channels—those that operate within a single domain. Even in perfect isolation, a program leaks its secrets if its execution time depends on secrets. These attacks have been shown to be exploitable

to extract cryptographic keys remotely over the network [71]. Secret-independent timing requires that a program's observable timing be independent of its secret data.

Programmers have traditionally sought isolation through *architectural* process isolation provided by primitives such as virtual memory and context switching, and secret-independent timing through constant-time programming. As the preceding sections have shown, however, neither guarantee holds on modern microarchitecture, whose shared, speculative, and aggressively optimized structures leak across both classes of channels: microarchitectural timing side channels are exploited to extract secrets from both colocated processes [39, 92] and code traditionally deemed constant time [48, 53, 71]. This thesis shows that both guarantees can be established with machine-checked proof through two complementary methodologies: **MTIsolation** (Part I), for proving strong timing isolation between colocated domains, so an attacker on the same machine learns no more than one on a separate machine; and **Granite** (Part II), for proving secret-independent timing down to RTL, so that a processor's cycle-by-cycle timing is determined solely by observables specified in an ISA contract.

1.3 Challenges with ruling out timing channels

Pursuing the goal of obtaining a machine-checked proof that a hardware design adheres to a timing-channel-aware contract raises two challenges, concerning *what the specification should say* and *how to make the verification tractable*. These challenges shape the approach presented in Section 1.4.

Formally specifying security. The first question to ask of any system claimed to be verified is: what's the spec? A specification should be both trustworthy—ruling out illegitimate implementations while remaining easy to audit—and useful—covering everything the rest of the system relies upon without ruling out legitimate implementations.

A specification is trustworthy only if it is easy to audit: a reader should be able to understand the guarantee from the formal theorem statement alone without inspecting the implementation or (ideally) reasoning through informal metatheory arguments. This discipline allows us to harness the power of machine-checked proofs, allowing systems to be audited based on the spec while enabling implementation optimizations that do not break the spec. Auditability rules out the design-specific invariants and low-level safety properties that pervade hardware verification, the kind that can only be understood by following a signal through the pipeline or trusting annotations in the implementation; such properties may be true and provable, but they leave the implementation in the trusted computing base. Similarly, simply enumerating known attacks or channels is unsatisfactory, as it does not create robustness to attacks not-yet-discovered. Furthermore, one must be careful about assumptions made in the formal theorem statement—for example, prior work proves secret-independent timing assuming functional correctness of an implementation [31, 74, 80], but if such assumptions are

design-specific and require inspecting internal implementation state, then the implementation remains in the TCB. Ideally, a specification of a hardware-software contract should be composable with a software-level proof such that the intermediate hardware-software-contract specification is eliminated from the TCB.

A specification is useful only if it covers all aspects of a component's behaviour relied upon by the rest of the system (i.e. the software stack above it and the external world) without overspecifying and ruling out legitimate implementations. Reconciling these two demands—avoiding both over- and under-specifying—is especially challenging with timing channels. First, there is an abstraction gap between the ISA, which operates at the level of instructions and architectural state, and the cycle-level, wire-level behaviours of circuits that are the target of timing channels. Second, the specification must reason about observable behaviour at cycle granularity, yet it must also allow microarchitecture the freedom to do the very thing it exists to do, namely alter cycle-level behaviour in pursuit of performance. For example, a cache that serves a hit in one cycle but a miss in twenty is a legitimate optimization that changes cycle-accurate timing. The specification must therefore permit timing to vary—a form of unspecified behaviour and nondeterminism.

Soundly allowing nondeterminism, however, is subtle and is a central challenge tackled by this thesis. It is not enough for a specification to merely permit a set of timing behaviours. Under the standard notion of correctness—that every behaviour of the implementation is one that the specification allows—a specification that simply admits timing variation is unsound for use as a security specification because an implementation remains free to resolve nondeterministic choices as a function of secret data[3]. A specification must therefore allow nondeterminism while constraining what may influence nondeterminism: for example, a memory access may take a number of cycles that depends on the public history of accesses, but never on secret store data.

Making hardware verification tractable. Modern microarchitectures are complex and concurrent. A high-performance processor holds many instructions in-flight at once, executed speculatively and out-of-order, with a myriad of predictors, buffers, and interacting microarchitectural state. The resulting state space is astronomical and model checking—the industry-dominant method for hardware verification that works by exploring all reachable states of a system to check that a property holds—runs into the state-space-explosion problem[4]. Timing channels compound the difficulty. They are frequently *emergent*, arising not from any single structure but from the interaction of several—for example, contention for a shared execution port or speculative interference between in-flight instructions—so they

[3] Indeed, this underspecified nondeterminism is arguably the underlying cause for why timing channels are *side* channels in the first place.

[4] A design with n bits of state has a state space up to 2^n. This growth is exponential in the number of state-holding elements (and compounded by the number of cycles being explored). For example, a design with 100 flip-flops gives up to 2^{100} states—more than the number of atoms in the observable universe!

cannot generally be ruled out by examining components in isolation.

Much of that state-space explosion is artificial, an artifact of reasoning about circuits at the wrong level of abstraction. Consider a FIFO (a first-in-first-out queue) of capacity n implemented as a circular buffer. The logical state "the queue holds one element" corresponds to roughly n distinct circuit states, one for each position the head and tail pointers might occupy. A model checker enumerating raw circuit states sees n states, whereas the canonical representation of a FIFO as a list has one state. This form of multiplicity compounds across every buffer, counter, etc. in the design.

In order to scale, abstraction and modularity should be made first-class, decomposing the global guarantee into per-component obligations so that submodules can be designed and verified independently (and then their proofs composed), and a local design change does not trigger an avalanche of proof breakages nor require rechecking the entire design. Crucially, these submodule specifications must encompass not only the necessary functional correctness but also the security constraints needed for timing-channel reasoning, while allowing flexibility for performance optimizations: a cache's specification, for instance, must expose that its latency may depend on the public history of accesses but never on secret data, so that the composed proof yields a global nonleakage guarantee, but should not constrain exact latencies. A challenge here is that traditional hardware-description languages (HDLs) like Verilog and VHDL are structural languages used to describe interconnections of boxes (of Boolean gates and registers) and were not designed to support modular reasoning about designs.

1.4 Overview and approach

This thesis develops a specification and verification approach for formally verifying secure hardware under timing-channel-aware contracts. Concretely, this thesis develops two methodologies for provably ruling out leakages in hardware: MTIsolation, which rules out interdomain leakage between security domains (isolation), and Granite, which rules out intradomain leakage of secrets (secret-independent timing). The two methodologies share an underlying specification approach, *leakage-aware refinement via determinism*. We first introduce the principles motivating our approach (Section 1.4.1), then give an overview of MTIsolation (Section 1.4.2) and Granite (Section 1.4.3), and finally summarize the specification, implementation, and verification strategies shared across both.

1.4.1 Guiding principles

This approach is motivated and guided by the following principles designed to bridge the gap between abstract security models and concrete hardware implementations.

1) Principled security specifications. Rather than engaging in a game of "whack-a-mole" against specific attack variants or protecting individual mechanisms, we focus on establishing trustworthy, readable specifications that rule out entire classes of interference and leakage. By adopting the principle of using an allowlist instead of a blocklist (enumerating allowed information flows and behaviours rather than ruling out known attacks or maintaining an incomplete list of safety properties), our specifications rule out sources of leakages not yet discovered. The specifications should stand alone—implementations should not be in the TCB—and should avoid overspecifying in order to support a family of secure implementations.

2) Modular verification for tractability. To handle the complexity of modern microarchitectures, we design abstraction boundaries to hide unnecessary microarchitectural complexity while exposing just enough information to reason about leakage, resulting in proofs covering a space of secure designs without overspecifying. These abstractions also enable architects to reason about early-stage designs, using higher-level abstractions suitable for design-space exploration. We decompose the verification challenge into per-component functional and security obligations. This decomposition allows hardware designers to verify submodules, such as branch predictors, caches, and arithmetic units, independently and prevents design modifications from triggering avalanches of proof breakages. We prioritize developing modular reasoning abstractions over immediate compatibility with existing HDLs.

3) Applicability to synthesizable hardware. Formal models are only useful if they reflect reality. We design our approach to apply to synthesizable, cycle-accurate register-transfer level (RTL) designs, bridging the gap between abstract security models and concrete hardware circuits.

4) Integration verification. A hardware-software contract is the shared, machine-checkable interface that lets a software proof and a hardware guarantee be composed into a single theorem about the whole system. We design our methodology to enable end-to-end verification from software to hardware, eliminating even intermediate specifications (including the hardware-software contract itself) from the TCB. For such integration verification that connects high-level software with RTL executions via compiler-correctness and processor-correctness proofs [25, 32], proof assistants (with their expressive logics) have thus far been the most compelling verification tool.

1.4.2 MTIsolation: proving hardware support for isolation

MTIsolation[5] focuses on restoring hardware support for isolation, tackling interdomain microarchitectural side channels arising from contention on shared resources. We propose

[5] For "Microarchitectural Timing Isolation". Pronounced "Mount Isolation", the last of my NH48. Alternatively, as a certain PhD supervisor put it, "Where PhD students go to finish writing their dissertations."

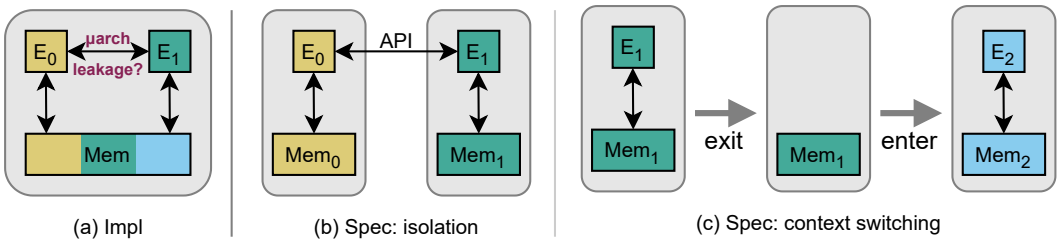


Figure 1-1: Preview of specifying strong isolation. Different colours denote different enclaves. (a) In the real system, enclaves on the same machine share microarchitectural resources and main memory. (b) In the specification, enclaves run on dedicated machines connected via a well-defined API. (c) Transitioning from one enclave to the next is modeled as "throwing away" the old machine and booting up a new machine.

to build confidence in mechanisms for eliminating microarchitectural leakages between security domains by formally ruling out, with machine-checked proof, interference across isolated processes or security domains, hereafter referred to as *enclaves*[6]. MTIsolation is a methodology, formalised in Rocq, for proving that an RTL design implements timing-sensitive strong isolation for enclaves. Informally:

Definition 1 (Strong isolation)**.** Programs colocated on the same machine observably behave as if they were running on separate machines, connected only by a dedicated API.

At a high level, an attacker on a different machine is only able to interact with the victim through a dedicated API, such as network calls. The attacker can observe the latency of API calls but should not be able to infer private information through shared microarchitectural resources such as caches or shared buffers. Thus, if we can prove a design implements strong isolation, then any private information that can be exfiltrated—via timing side channels or otherwise—by a local attacking process can also be exfiltrated by a remote process. This property restores the process-isolation abstraction to the enclave programmer, yet it is not complete: subtleties remain with enclave context switching and communication.

MTIsolation introduces a formalisation of strong isolation in the presence of context switching, previewed in Figure 1-1. We develop a cycle-level specification logically modelling enclaves as running on separate machines that are air-gapped apart from communication made explicit via an API allowed by the threat model. This formulation rules out interference arising from shared microarchitectural resources by definition: enclaves cannot be affected through e.g. shared cache lines or contention for DRAM controller bandwidth as they do not share cache lines or DRAM controllers in the specification. Context switching is modeled as discarding the old machine and booting a new one, with any information preserved across

[6]The term "enclaves" is associated with various meanings, especially in the context of TEEs. We adopt a more general meaning, as isolated processes or security domains, and expect our work to be relevant both for TEEs and, more broadly, for virtual machines and process isolation.

the switch defined *explicitly*—so leftover branch-predictor or cache state cannot carry secrets across protection domains. As the specification enumerates allowed flows (an allowlist) rather than forbidding known attacks, it rules out sources of interference not yet discovered. Crucially, our specification is parametric on non-security-critical functional and timing behaviour and does not overspecify: it suffices to show that the implementation behaves like air-gapped machines without constraining exactly what the machines are; these air-gapped machines are "black boxes" with access only to information accessible by the corresponding protection domain. This formalisation is readable, concise, and trustworthy—a specification-reader does not need to reason about design-specific, low-level details—and is general enough to capture a range of secure designs.

The specification imposes minimal restrictions on an implementation's functional correctness, setting the stage for a methodology for formal verification of isolation of RTL designs that sidesteps full functional-correctness proof. We show that securely implementing and proving isolation can be decomposed into two parts largely independent of functional correctness: (1) enforcing a simulation relation between running enclaves in the implementation (colocated on one machine) and specification (on separate machines) through spatially and temporally partitioning resources, and (2) reaching a state functionally equivalent to an appropriately initialised new machine when context switching by purging microarchitectural state. This decoupling allows non-security-critical design modifications to be made with minimal effects on the proof, alleviating a traditional problem with formal verification where relatively small changes trigger avalanches of proof breakages. For example, substituting a simple RISC-V processor with a more complex processor would only require modifications to the proof that pertain to the secure implementation of purge. In addition, we show how to decompose the proof modularly into per-component security obligations, allowing hardware designers and verifiers to implement and prove security properties independently for submodules such as the processor and memory hierarchy. Lastly, we demonstrate a hybrid Rocq-SMT strategy that partially automates reasoning about circuit-level designs. This hybrid approach reduces verification costs while bypassing the usual scalability challenges faced by SMT solvers and maintaining an expressive specification language for readability and composability with full-stack, hardware-software guarantees.

To demonstrate our methodology, we build and verify a multicore, pipelined RISC-V system implementing strong timing isolation, written in the Kôika HDL [16] and lowered to circuits by its verified compiler. We validate the prototype's functionality through executing a suite of RISC-V and C tests compiled with standard toolchains using Cuttlesim [65] and Verilator, and we study programming under the enclave abstraction with a toy password-manager application. To the best of our knowledge, this is the first machine-checked proof of strong isolation (or lack of timing side channels) for an enclave system. Part I gives the full development, and Section 1.4.4 describes the specification and verification strategy that MTIsolation shares with Granite.

1.4.3 Granite: proving hardware support for secret-independent timing

Whereas MTIsolation focuses on interdomain leakage and restoring hardware support for isolation, Granite focuses on intradomain leakage and restoring hardware support for confidentiality. For decades, cryptographers relied on constant-time programming to ensure that the wall-clock time when executing cryptographic software on hardware is independent of secrets. However, beginning with Spectre and Meltdown, there has been a surge of microarchitectural timing vulnerabilities invalidating traditional constant-time assumptions and exposing novel leakage vectors. Formal verification against leakage-aware architectural specifications offers a compelling approach, adopted by this thesis, for providing software with the tools to defend itself against timing side channels.

Granite is a methodology, formalised in Rocq, for end-to-end verification of both functional correctness and nonleakage of cycle-accurate RTL processors against leakage-aware ISA contracts—specifications that pair the functional ISA semantics with a declassification policy fixing exactly which architectural events (for the constant-time policy, the addresses of memory accesses, branch conditions, and arguments to variable-latency instructions) may influence observable timing. We show how to build modular proofs for processors and their components, with specifications that capture which information flows are allowed without prescribing specific implementation behaviours or requiring determinism. Proving that an RTL processor implementation satisfies such a contract yields a mathematical guarantee that hardware does not introduce unexpected timing channels beyond what is explicitly permitted by the contract.

Granite's specification approach, *leakage-aware refinement via determinism*, reduces both functional correctness and nonleakage to trace equivalence against a family of deterministic, cycle-accurate state machines. The central challenge is supporting nondeterminism without underspecifying (allowing unintended leakage) or overspecifying (ruling out secure implementations). Our strategy is to *existentially parameterise* each specification with a deterministic, public *driver*—depending only on public information (public inputs, the program, and declassified leakage), responsible for resolving secret-independent nondeterminism such as the timing of outputting an MMIO request or at which instruction boundary to take a precise interrupt—and a deterministic *witness* for nondeterminism that may legitimately depend on secrets. These parameters are untrusted.

We formalise nonleakage as the existence of a global *leakage transformer* mapping the trace of public inputs to adversary observations (e.g. timing), with the (static) choice of that function left intentionally unconstrained to allow for implementation variability. This property trivially implies the classical notion of noninterference. A simple example appears in Figure 1-2. This idea takes a variety of shapes when reconciling differences between specification and implementation; we show how to capture that:

- A zero-skip multiplier's (public) latency can depend on whether either operand is zero

```
Record req_t := { input_a : bv width; input_b: bv width }.
Record MulSt := { reqs: list req_t; hist : list ActionMethod }.
Definition leakage_of_AM (m: ActionMethod) : LeakEvent :=
  match m with
  | Enq req => LeakEnq ((req.(input_a) = 0) || (req.(input_b) = 0))
  | Deq => LeakDeq
  | Tick => LeakTick
  end.
Definition RespReadyOk impl spec := exists leakageTransformer,
  impl.(RespReady) = match spec.(reqs) with
                     | [] => false
                     | _ => leakageTransformer (map leakage_of_AM spec.(hist))
                     end.
```

Figure 1-2: Example fragment of a module specification: the public timing of a zero-skip multiplier is a fixed function of whether its operands are zero. Functional correctness requires a pending request for `RespReady` to be high.

but not on other aspects of input data.

- The response time of a memory subsystem may depend on addresses but not on values stored and loaded.
- While the instruction-set specification processes one instruction per step, a processor cycle may or may not commit a new instruction—but it must not decide based on secrets.
- Timing of external inputs and interrupts can influence processor progress, but values of inputs cannot in general be revealed.
- A processor can delay taking an interrupt (note that this decision can alter the sequence of instructions executed and thus the ISA-level observations!), but the decision of when to take the interrupt should not depend on secrets.

We show how to formalise an ISA leakage contract (covering I/O, exceptions, and interrupts) in this style by lowering the ISA specification—written in the style of RISCV-COQ [13, 25] and extended with a leakage model emitting declassified events (e.g. memory addresses, branch conditions)—into a cycle-level machine whose nondeterminism is constrained by existential parameters. Functional correctness becomes I/O-trace equivalence over all programs, data, and inputs; nonleakage becomes the existence of a leakage transformer from public inputs, public data, and the specification leakage trace to public outputs.

The verification is structured around both vertical and horizontal modularity. *Vertically* (cf. Figure 9-1), a layered approach decouples ISA-specific details from microarchitectural design and parameterises the proof over different ISA encodings and existential parameters; we show that connecting the layer to RTL amounts to instantiating the existential parameters

by running a public "shadow" copy of the implementation. *Horizontally*, we construct per-component specifications—with canonical representations capturing both correctness and leakage, plus sub-cycle invariants—extending prior one-method-at-a-time modular refinement [15, 23, 59] to cycle-accurate nonleakage, yielding theorems general enough to cover classes of designs and to verify early-stage designs and defenses.

As an end-to-end demonstration, we build a confidentiality-and-correctness proof for a pipelined RISC processor (as synthesizable RTL) featuring branch prediction, exceptions, interrupts, and memory-mapped input and output. We then demonstrate suitability of the instruction-set specification for integration verification by implementing and proving a static analysis for constant-time programming: any program accepted by the static analysis, run on a processor satisfying the specification, does not leak secrets at the RTL. Instantiated on a Salsa20 program compiled with a standard RISC-V toolchain, this proof architecture yields a single, foundational proof connecting the source-level constant-time discipline to RTL and eliminates intermediate specification layers (including the hardware-software contract) from the trusted computing base. Part II gives the full development, and Section 1.4.4 describes the specification and verification strategy that Granite shares with MTIsolation.

1.4.4 Specification, implementation, and verification strategies

Our work is grounded in the Rocq proof assistant. Its expressive logic accommodates auditable specifications, first-class modular abstractions and refinement, reasoning about metatheory, and direct connection to software proofs.

Specification strategy: leakage-aware refinement via determinism. Part I and Part II target different security properties (isolation by eliminating interdomain leakage and secret-independent timing by eliminating intradomain leakage respectively), but they share the same underlying specification strategy: leakage-aware refinement via determinism. This strategy formulates both functional correctness and nonleakage as observational equivalence or refinement to a cycle-accurate, deterministic, reference model. Specifically, we define a family of deterministic, cycle-accurate specification state machines that expose the same circuit-level interface as the implementation. These specification machines are *existentially parameterised* by untrusted abstract functions that explicitly define the dimensions along which designs may vary, capturing exactly which behaviours (such as timing variations depending on a subset of inputs) are admissible without overspecifying. These "black box" parameters enable explicit specification of both secret-independent nondeterminism and secret-dependent determinism by constraining information flows appropriately (only public information flows into black boxes representing secret-independent choices, and all information can flow into black boxes representing secret-dependent choices). We show that these existential parameters can be instantiated with shadow copies of the implementation—air-gapped machines in the

case of isolation, and machines without access to secrets in the case of secret-independent timing.

This specification style allows the specification to support safe, implementation-specific performance optimizations without trusting the implementation (the implementation stays outside of the TCB), rules out leakage without enumerating attacks while supporting evolving threat models using an allowlist approach (for instance, allowing explicit communication between the air-gapped machines or declassification of secrets for secret-independent timing), and sets the stage for a scalable verification strategy based on single-cycle preservation of a simulation relation. Furthermore, this refinement style supports a modular approach to verification, both vertically by composing layers of the stack and horizontally by admitting a substitution principle allowing complex submodules to be replaced by their abstract specifications during the verification of higher-level components. This approach extends decades of work on verifying functional-correctness of processors using proof assistants [15, 23, 32] to the setting of microarchitectural timing security.

Implementation strategy. At the synthesizable design level, we implement our case studies in HDLs (Kôika [16] and Quartz [2]) with cycle-accurate, sequential semantics embedded in the Rocq proof assistant. Kôika and Quartz enable modular reasoning about hardware designs as sequences of atomic transactions. Verified compilers lower the designs to circuit level. Section 15.1 discusses and contextualizes decisions made on this front.

Verification methodology. We reduce the proof to preserving a simulation relation across a single clock cycle. Single-cycle simulation relations sidestep the state-explosion problems common in verifying deep, speculative pipelines, and they are amenable both to interactive proof, as in Part II, and to automated, SMT-based discharge as in the hybrid toolchain of Part I. We show how to create modular, canonical specifications of components that capture both functional correctness and nonleakage. Part II demonstrates a modular and layered refinement methodology supporting early-stage designs.

1.5 Contributions

This thesis shows that timing-channel security can be established as a first-class, machine-checked property of synthesizable hardware, with robustness to attacks not yet discovered. It shows that hardware support for isolation and confidentiality can be proven down to the circuit level and in the presence of timing channels.

This thesis contributes:

- *Leakage-aware refinement via determinism*: a method for specifying timing-channel security based on trace equivalence against a family of deterministic, cycle-accurate specifications parameterised by untrusted existential parameters.

- To address **isolation**:
  - A formalisation of strong timing isolation based on "air-gapped machines" supporting dynamically evolving security domains.
  - A methodology and Rocq-SMT toolchain for modular proofs of isolation for RTL designs that sidesteps full functional-correctness proof.
  - A prototype and machine-checked proof of a multicore, pipelined RISC-V system implementing strong timing isolation.
- To address **secret-independent timing**:
  - A specification approach for functional correctness and nonleakage under nondeterminism, via trace equivalence with deterministic, cycle-accurate state machines.
  - A formal HW/SW leakage contract covering I/O, exceptions, and interrupts.
  - A methodology for modular proofs of functional correctness and nonleakage of RTL designs.
  - A machine-checked proof of a synthesizable, pipelined RISC processor (speculation, interrupts, I/O) against a constant-time ISA contract.
  - An end-to-end proof combining the above with verified-constant-time software into an RTL confidentiality theorem, stated without reference to the novel specs.

Ultimately, this thesis demonstrates that timing-aware contracts, supported by mechanised proof, offer a principled path toward hardware that provides meaningful and provable confidentiality guarantees in the presence of complex microarchitectural optimizations.

1.5.1 Limitations and nongoals

The techniques developed in this thesis have limitations, described in more detail in the relevant chapters. Globally, this thesis targets the digital abstraction within a single-clock domain from the ISA level down to RTL and does not handle physical side channels such as radiation, temperature, and power. The case-study processor designs were chosen to include features to demonstrate the methodology—focusing on in-order, pipelined cores—and do not aim for full ISA coverage or optimal performance. The case-study designs were expressed in HDLs with sequential semantics, and we do not directly verify designs written in industry-standard HDLs such as Verilog or VHDL (see Section 15.1.5 for more discussion on this front). We focus on safety properties and do not address liveness—a brick does not leak secrets. As with all verification projects, this work does not defend against attacks exploiting mistakes in the specification[7].

[7] But broadening proofs helps catch errors in specifications that do not appear in top-level system theorems!

1.5.2 Open-source software

We have open-sourced all software developed as part of this thesis:

- **MTIsolation:** https://github.com/mit-plv/isolation
- **Granite:** https://github.com/mit-plv/granite
- **Quartz:** https://github.com/mit-plv/quartz

1.6 Related publications

This dissertation incorporates and extends work covered in three papers:

> Stella Lau, Andres Erbsen, Adam Chlipala. Granite: A Modular Methodology for Foundational Verification of Hardware-Software Leakage Contracts. In submission; preprint available at https://arxiv.org/abs/2607.27480.
>
> Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. Specification and Verification of Strong Timing Isolation of Hardware Enclaves. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS 2024).*
>
> Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. Pensieve: Microarchitectural Modeling for Security Evaluation. In *Proceedings of the 50th International Symposium on Computer Architecture (ISCA 2023).*

Chapter 2

Background and Related Work

This chapter introduces shared background for both parts of this thesis. We first provide context for microarchitectural timing side channels (Section 2.1), distinguishing between interdomain and intradomain channels addressed by Part I and Part II respectively. We then introduce concepts that our specifications build upon: noninterference, declassification, and refinement in the presence of nondeterminism (Section 2.2). Next, we discuss verification approaches for secure hardware (Section 2.4). Finally, we describe the rule-based hardware-description languages in which our case studies are implemented (Section 2.5). Further background specific to verifying hardware-software leakage contracts appears in the introduction of Part II.

2.1 Context: microarchitectural timing side channels

The instruction-set architecture (ISA) is the contract between hardware and software. An ISA such as RISC-V or x86 defines an architectural model, consisting of registers and memory, and how each instruction updates architectural state, granting hardware designers freedom to optimize (e.g. via pipelining, branch prediction, and caching) so long as architectural state matches a sequential execution of instructions[1]. Traditionally, this contract has been purely functional, omitting microarchitectural state and abstracting over latency. However, because the ISA does not specify the timing (or power consumption) of instructions, hardware optimizations can leak information about secret data through these side channels.

We distinguish between two classes of timing channels based on the confidentiality properties they violate: *isolation-breaking* channels (interdomain) and *constant-time-breaking* channels (intradomain).

[1] Relaxed with weak memory, outside the scope of this thesis.

2.1.1 Isolation-breaking timing channels (interdomain)

In this setting, multiple protection domains (e.g. processes, VMs, cloud tenants, or enclaves) execute on a shared microarchitecture. Architectural isolation promises noninterference: an attacker domain should be unable to distinguish the secret states of a victim domain and also unable to interfere with the victim. However, to maximize hardware utilization, implementations physically share microarchitectural resources—such as caches, memory bandwidth, and branch predictors—across domains. Contention on these shared resources modulates the execution time of both the victim and attacker.

This resource contention creates a timing channel. For example, in a *Prime+Probe* [54] attack, an attacker fills a cache set with their own data (prime), allows the victim to execute, and then measures the latency of accessing that same data (probe). If the access is slow, the attacker infers that the victim accessed memory mapping to that specific cache set, potentially revealing cryptographic keys or access patterns. These attacks have been demonstrated on production cloud environments, where an attacker colocates with a victim [91] and then, for example, exploits contention on the shared, last-level cache to extract cryptographic secrets [92]. While defenses such as cache partitioning have been proposed, they are often applied ad hoc, and attacks continue to be discovered as these solutions are either unverified or too localized. Fundamentally, whenever there is contention on a shared resource, there is opportunity for leakage.

Constructing enclaves secure against timing side channels. Instead of localized defenses, academic designs like Sanctum [26] and MI6 [14] take a broad, principled approach to implement microarchitectural modifications with the aim of restoring strong timing isolation: spatially and temporally partition all resources to construct isolated processes, or enclaves. Resources are assigned to protection domains independently of their demand; for example, last-level caches and DRAMs are partitioned across protection domains, a domain can have at most, e.g., half the DRAM controller bandwidth irrespective of the memory intensity of colocated domains, and static arbiters (e.g. round-robin arbiters) resolve cycle-level contention. Resources left behind when context switching between domains can also be a source of leakage: MI6 introduced a *purge* instruction to erase program-dependent state when exiting an enclave—involving flushing in-flight instructions, purging branch predictors, and flushing caches— and Wistoff et al. [84] implement a temporal fence instruction for context switching analogous to MI6's `purge`, experimentally validated to have secure (history-independent) timing and low performance overhead. These approaches inspire work in this thesis but they do not tackle the verification challenges in formally proving isolation. Subramanyan et al. [73] verified Sanctum's security, but they verify models rather than synthesizable designs, and their spec is expressed as relatively complex invariants (in contrast with our high-level isolation spec). MTIsolation develops a specification of strong isolation that captures the dynamically evolving security domains present with context switching, formalising and adapting the parti-

tioning and purging mechanisms proposed in MI6 to achieve provable strong isolation at the RTL.

2.1.2 Constant-time-breaking channels (intradomain)

While isolation is a useful primitive that provides broad protection between security domains with minimal requirements on software, it does not generally suffice to achieve confidential hardware-software systems: even a program that runs in isolation may leak secrets if its execution time depends on secrets. Cryptography libraries have long included these timing side channels in their threat model and mitigated them by operating under a de-facto, constant-time leakage contract. A *leakage contract* closes this gap by specifying which architectural events may influence timing, acting as a declassification policy. Under the cryptographic constant-time policy, only control flow, addresses of memory accesses, and inputs to variable-latency instructions are leaked; it is then the software programmer's responsibility to keep secrets out of these channels. Numerous libraries hold all cryptographic code to this contract, either informally (BoringSSL, BearSSL) or formally (HACL* [93] and EverCrypt [66]).

Guarantees provided by these contracts are only as sound as the leakage model, and models routinely make assumptions that some real processors do not satisfy—details of hardware-software leakage contracts can be tricky to get right, and it is not clear that all corner cases have been considered. For example, HACL* assumes integer multiplication is constant-time, but integer multiplication can be variable-time on some ARM and i386 platforms; OpenSSL shipped cryptographic code deemed secure under a leakage model that did not account for time-variable arithmetic operations [6]; and hardware verification exposes still more: Granite's case study (Chapter 12) exposed a specification bug—writes to interrupt-related system registers must also be considered public (the processor will jump to these addresses upon interrupts).

Transient execution. The traditional constant-time discipline reasons about the sequential ISA semantics, but modern processors do not execute sequentially. To hide the latency of unresolved branches and pending loads, modern processors employ dynamic branch prediction to guess the direction and target of control flow and execute instructions transiently before the branch condition is resolved. During this "speculation window", the processor executes instructions based on a guess. If the guess is correct, the instructions retire; if incorrect, the architectural effects (register updates) are rolled back, and the pipeline is flushed. Crucially, however, the microarchitectural side effects of the misspeculation (such as the cache lines fetched or the predictor entries updated) are generally not rolled back. This gap allows secrets to be exfiltrated even from code that is architecturally dead or unreachable—breaking the hardware-software contract assumed by cryptographers—and is what Spectre [48] exploits.

In the bounds-check-bypass variant (Figure 2-1), the attacker first mistrains the branch predictor so that the guard is predicted taken, then supplies an out-of-bounds `x`; the processor

```
1 if (x < array1_size)              /* mistrained: predicted taken */
2     y = array2[array1[x] * 64];   /* transient secret-dependent load */
```

Figure 2-1: The Spectre v1 (bounds-check-bypass) gadget. With `x` out of bounds, the guarded load executes transiently and leaves a secret-dependent footprint in `array2`.

speculatively reads the secret byte `array1[x]` and uses it to index `array2`, imprinting a secret-dependent line into the cache that survives the squash and is recovered by a subsequent Prime+Probe or Flush+Reload over `array2`. The victim's source code is constant-time under the sequential model; the leak lives entirely on the transient path. Meltdown [53] exploits a related mechanism: a faulting load to a protected address forwards its data to dependent transient instructions before the permission fault is architecturally delivered, allowing those instructions to encode the protected value into the cache and thereby cross an architectural protection boundary.

Defending against transient-execution attacks. The computer-architecture community has proposed a variety of defenses. One set of defenses focuses on achieving "invisible speculation" [5, 42, 69, 85] or speculative noninterference [24, 88]. These defenses attempt to block speculative-execution attacks by hiding the side effects of speculative memory accesses on the memory hierarchy. However, hiding the cache footprint of speculation is not sufficient: speculative-interference attacks [10] show that misspeculated younger instructions can perturb the timing of older, bound-to-retire instructions via transient contention on shared ALUs and MSHRs, so that secrets leak through the timing of instructions that do commit. GhostMinion [4] was proposed to defend against speculative-interference attacks, stating that a processor only allows older instructions to affect the timing of younger instructions. While to date there have been no attacks on the *theoretical* defense, the devil is in the details—there have been no verified implementations[2], and it is unclear how GhostMinion interacts with e.g. hardware prefetchers. Timing leakage is an emergent property of the microarchitecture as a whole, motivating verification of entire designs rather than informal reasoning about localised fixes.

2.2 Specifying security: noninterference and refinement

In this thesis, timing-channel security is concerned with ensuring that the (adversary-observable) timing of higher-level constructs (such as the time it takes for a program to execute or the timing of I/O requests/responses) does not leak secrets. This top-level property can be stated as a form of *noninterference*: a high-level security policy requiring that an adversary's observations

[2] Indeed, Pensieve [86] discovers an attack variant based on GhostMinion's informal prose description underspecifying an aspect of the defense scheme.

be independent of secret data (Section 2.2.1); alternatively, this property can be stated as requiring that an adversary's observations be a function of only public data. However, specifying what data is considered public can be complex: it is generally not a straightforward labeling of input signals but instead involves *declassification* that may be based on higher-level abstractions (for example, in the setting of hardware-software leakage contracts, branch conditions under an instruction-level semantics are declassified whereas an adversary's observations are at the wire level). Furthermore, reconciling the mismatch between higher-level semantics (e.g. instruction level) with RTL semantics often requires functional-correctness arguments in addition to information flow.

We found constructing a state machine that specifies what information is declassified (considered public), and then quantifying over "functions of only public data" to enforce information flow, to be a general method for expressing noninterference with declassification (discussed further in Section 15.2). This specification strategy gives rise to a top-level theorem based on *refinement*—essentially, any implementation behaviour is also a behaviour of the specification (Section 2.2.2)—that builds upon decades of work on functional-correctness verification of hardware [15, 23, 32]. Refinement, however, does not naturally suffice for establishing information-flow security under nondeterminism [68]. This thesis shows how we can extend the refinement strategies developed for functional-correctness verification for timing-channel security through determinizing nondeterminism. The remainder of this section provides further background on noninterference and refinement.

2.2.1 Noninterference

Noninterference is a security policy requiring that an adversary's observations be independent of secrets: any two executions agreeing on adversary-visible "public" data must yield observations that the adversary cannot distinguish. As a relation over pairs of executions it is a hyperproperty rather than a property of any single run, and it underlies both security goals of this thesis.

Isolation. Isolation is the special case in which the secret is the entire state of a security domain: a design enforces isolation if the observations available to one domain are independent of the secrets of every mutually distrusting domain. Strict noninterference is too strong once domains must communicate, however, so the definition is relaxed with a declassification policy, which explicitly permits a controlled set of flows such as messages exchanged through a communication API.

Secret-independent timing. For secret-independent timing, the relevant secrets are a single domain's own data, and the adversary observes that domain through timing of, for example, external calls. Here, characterising what an implementation may leak has moved beyond ISA

functional correctness to leakage contracts [38] that name the architectural events—memory-access addresses, branch outcomes, operand-dependent latencies—that a threat model permits to influence observable behaviour. Such a contract acts as a declassification policy; the corresponding noninterference statement is *contractual noninterference* (discussed further in Part II): two executions whose declassified leakage traces agree (under the architectural model) must be indistinguishable to the adversary at the microarchitectural level. By ensuring that hardware satisfies this contract, the software programmer can reason at the architectural level, ensuring that its program does not leak secrets according to the ISA, without needing to reason about the microarchitecture.

Both goals are thus instances of noninterference modulo declassification, differing only in what counts as secret and in what the contract declassifies.

Specifying and verifying noninterference under declassification. There is a long line of work on formalising and verifying noninterference: the challenge is not with handling the baseline noninterference property (here, one can imagine a trivial information-flow type system) but handling the declassification needed to support realistic systems. Notable examples include work on verifying the seL4 kernel [47], mCertiKOS hypervisor [28], and Komodo security monitor [35]: these projects formalise noninterference and information-flow security at the ISA level, but they do not tackle RTL verification. At the hardware level, HyperFlow [36], SecVerilog [90], and SpecVerilog [89] use a language-based approach to enforce security through type systems tracking information flow. Although type checking and information-flow techniques provide a high degree of automation and impose low verification overhead, they face completeness limitations common in static-analysis approaches. For soundness, type systems are conservative and may reject secure designs because the type system cannot reason about the functional logic that prevents leakage. For expressivity, designers use declassification policies to express allowed leakages. However, when defined at a low level associated with a particular implementation's wires and registers, understanding the security guarantee requires detailed auditing and deep understanding of the system. In contrast, the specifications in this thesis both are auditable without reference to low-level implementation details and do not compromise on expressivity. For isolation, the specification captures noninterference under dynamically evolving security domains and explicit communication between enclaves; for secret-independent timing, verifying both functional correctness and nonleakage together yields a contract-level noninterference theorem that removes the implementation from the TCB and composes with software proofs.

2.2.2 Refinement and nondeterminism

One approach to processor verification, adopted by this work, is to use an interactive theorem prover to establish correctness via *refinement*. At a high level, refinement involves proving that every behaviour of the implementation state machine is allowed by the specification

state machine, typically formalised as trace inclusion or the existence of a simulation relation between states.

Refinement has been used to verify functional correctness of processors in Kami [23] and Fjfj [15], languages without cycle-accurate semantics. Refinement extends cleanly to nonleakage for deterministic specifications (as trace equivalence proved via bisimulation), but isolation and architectural specifications support various types of nondeterminism (see below) to allow for interaction with the external world and to allow hardware designers freedom to optimize. However, a naïve extension to nondeterministic specifications can be unsound as nondeterministic choices in the specification may implicitly encode secret-dependent behaviour, allowing a leaking implementation to be "explained" by an adversarial choice of specification nondeterminism [68].

Sources of nondeterminism in isolation specifications include:

- Execution time: the number of cycles it takes to execute an enclave;
- Input nondeterminism from the external world, such as from MMIO;
- The results of the computation: functional correctness, modulo aspects for architectural isolation of security domains, is not required under isolation.

Sources of nondeterminism under hardware-software contracts include:

- Execution time: the number of cycles it takes to execute instructions;
- Input nondeterminism from the external world, such as from MMIO and interrupts;
- Secret-independent nondeterminism, such as at which instruction boundary an asynchronous interrupt is handled; and
- Unconstrained output nondeterminism, such as the value on a `data` signal when the corresponding `valid` bit is low in a ready/valid handshaking interface.[3]

To avoid this pitfall (of nondeterminism leaking secrets) without overspecifying the implementation, this thesis uses *existentially parameterised* specifications: we quantify over deterministic "driver" state machines that receive only public information for those choices that must be secret-independent, and unconstrained "witness" deterministic state machines for choices that can depend on secrets. This approach yields a refinement notion that avoids overspecifying and remains expressive while enabling compositional nonleakage theorems.

We provide further discussion on the challenges of extending prior work on functional-correctness verification of processors to security in Part II, after introducing the definition of contractual noninterference.

[3] Here, we assume it is the consumer's responsibility to use the `data` signal only when the corresponding `valid` bit is high.

2.3 Verifying noninterference under nondeterminism

The software-verification community has developed various techniques to reason about security in the context of the nondeterminism of underspecified behaviours arising from e.g. concurrency and compiler choices. O'Neill et al. [63] use *refiners* to determinize nondeterministic semantics for an information-flow noninterference definition—analogous to our existential parameters, but their focus is on information-flow static analyses rather than specifications and proofs, and they do not address the challenge of declassification. Conoly et al. [25] use *predictors* to prove a compiler preserves constant time under allocation and I/O nondeterminism, also via leakage transformers, but do not tackle anything analogous to the underspecification of the mapping of instruction boundaries and their approach does not address nontermination (they require finite leakage traces). Pantomime [82] also derives the implementation leakage trace from a specification trace via a simulator, but its specification trace comes from a circuit-level description of complexity comparable to the implementation, whereas Granite relates RTL to an instruction-set-level leakage trace. Similar to our security goal, Knox [9] introduces information-preserving refinement (IPR) for hardware security modules but targets transaction-level calls rather than ISA leakage contracts; Parfait [8] proves IPR by collapsing hardware-software boundaries but for a specific program on a minimal core (PicoRV32).

2.4 Other verification approaches for secure hardware

2.4.1 Model checking

A significant body of work uses model checkers to search for inductive invariants to prove safety properties such as noninterference. While appealing for their push-button automation, these tools suffer from the state-explosion problem, limitations in their specification languages—safety properties are often expressed as difficult-to-audit invariants rather than high-level security definitions, with the metatheory justifying the correspondence to the desired high-level properties left to informal reasoning—and brittleness due to monolithic proofs where small design changes require reproving large parts of the system. Our work on Part I uses a hybrid Rocq-SMT approach to discharge low-level, sub-single-cycle properties of concrete designs—taking advantage of both the expressivity of Rocq and the automation of SMT solvers (at the cost of a larger TCB).

Model checking constant-time properties. Model checkers have been used extensively for checking designs against constant-time properties. Tools like IODINE [76], UPEC-DIT [31], and Xenon [77] can successfully identify constant-time violations in both in-order and out-of-order RTL, but they generally restrict their scope to ensuring individual instruction latency is

independent of operands, often ignoring the complex leakage vectors introduced by speculative execution. UPEC [34], LeaVe [80], and Contract Shadow Logic [74] use model checkers to check speculative constant-time contracts. However, there are challenges with aligning these approaches with our guiding principles from Section 1.4.1. First, these works assume functional correctness of the processor and therefore express their properties in a manner that requires auditing the implementation. Second, they rely on automated invariant generation that often does not scale to deeper or more complex pipelines. The challenge here is that speculative-execution defenses often rely on aspects of functional correctness, which automatic invariant-generation techniques struggle to infer. Third, they do not natively support the nondeterminism introduced by interrupts and I/O, and they typically assume no exceptions or interrupts are raised during execution. Fourth, the checked properties are constrained by the expressivity of SMT-based property languages: they are effective for bounded relational checks but less suited for high-level software specifications and compositional statements combining functional correctness, declassification, and nondeterminism. In contrast, Granite verifies both functional correctness and security of RTL designs against high-level hardware-software specifications encompassing nondeterminism, obtaining an easy-to-audit proof that composes with software proofs for an end-to-end guarantee.

2.4.2 Hardware-software verification

Work on end-to-end verification of hardware and software has largely focused on functional correctness proofs. The Bedrock2 lightbulb [32], garage door [33], and CakeML stack [55] verify functional correctness of hardware-software systems but do not rule out information leakage. Parfait [8] uses automatic verification tools to collapse abstraction layers symbolically, from software down to RTL, and prove functional and leakage properties about a specific program on relatively simple hardware. VRASED [60] verifies a hardware-software codesign for remote attestation. While Parfait and VRASED target verification of specific programs, this work proves properties about the microarchitecture *for all programs* and preserves modular abstractions for scalability.

2.4.3 Abstract microarchitectural models and mitigations

A complementary line of work reasons about leakage on abstract processor models rather than synthesizable RTL. Pensieve [86], Guarnieri et al. [38], and CheckMate [75] target early-stage architectural models: these models are valuable for design-space exploration of defenses, but they underspecify the cycle-level implementation details from which leakage arises, and they do not prove that any concrete implementation satisfies them. Pensieve introduces a specification style for submodules based on decomposing functional-correctness and leakage properties, which Granite extends for use in verification. Motivated by transient-execution attacks, richer leakage models [20, 57] with fuzzing-based validation [61, 62], processors

designed for secure speculation such as ProSpeCT [29], defenses such as GhostMinion [4], and compilers such as Serberus [58] all aim to close the leakage gap but remain unverified at the RTL. This thesis complements these efforts by using a modular approach that is suitable for early-stage designs (with abstract specifications for submodules) while also proving that synthesizable RTL adheres to a leakage-aware contract.

2.5 Rule-based hardware-description languages

The case studies in this thesis use languages and ideas based on rule-based hardware-description languages (HDLs) in the family of Bluespec [59]. Compared to traditional HDLs such as Verilog, rule-based HDLs support a more software-like semantics and enable modular, sequential reasoning. In rule-based HDLs, the design specifies stateful elements (registers) and describes the behaviour using atomic rules. Each rule defines a deterministic state transformation on registers, and it is guaranteed that rules *appear* to execute atomically, or "one-at-a-time". The atomic transactions allow sequential, high-level reasoning about designs. This semantic structure has been exploited in Rocq developments to make hardware proofs more tractable: Kami [23] and Fjfj [15] provide formal semantics for rule-based designs with nondeterministic rule scheduling and prove theorems that allow submodules to be reasoned about in terms of their specifications. However, the nondeterminism in rule scheduling abstracts away cycle-accurate timing behaviour, limiting their applicability to reasoning about timing channels.

Kôika [16] addresses this gap by providing a cycle-accurate rule-based semantics formalised in Rocq—with a verified compiler to circuits—but it forgoes the strong per-rule modularity that makes one-rule-at-a-time reasoning effective, and it does not directly support modular verification of hierarchical submodules[4]. We address these limitations with Quartz, a deterministic HDL mechanised in Rocq representing combinational circuits; inspired by rule-based design, it provides a "one-function-at-a-time" execution model that avoids rule-conflict reasoning and simplifies mechanised proofs. In Part I, we present our MTIsolation prototype implemented in Kôika. The hardware-software leakage-contract work in Part II is implemented in Quartz[5]. Both Kôika and Quartz and their formal, cycle-accurate semantics are mechanised in Rocq, along with verified compilers to circuits.

[4] Consequently, we developed a hybrid Rocq-SMT toolchain to automate aspects of the reasoning.

[5] Our intermediate representation is shallowly embedded and independent of the underlying HDL, and it is proven equivalent to the Quartz semantics.

Part I

Isolation

Chapter 3

Overview of MTIsolation

Introduced in Section 1.4.2, MTIsolation tackles the challenge of proving that RTL hardware supports strong timing isolation for security domains ("enclaves"). It is informally based on the observation that programs colocated on the same machine should behave *as if* they were running on separate, air-gapped machines and that context switching can be modeled as throwing away a machine and booting a new machine. However, isolation does not specify *how exactly* these air-gapped machines behave, allowing specifications to be parameterised by untrusted, black-box machines. This gives rise to a verification approach that sidesteps full functional-correctness proof of the processor and thus allows non-security-critical modifications to the processor without changes in the proof script. Of course, the execution time and external interactions of enclaves are leaked: Part II closes this gap with Granite.

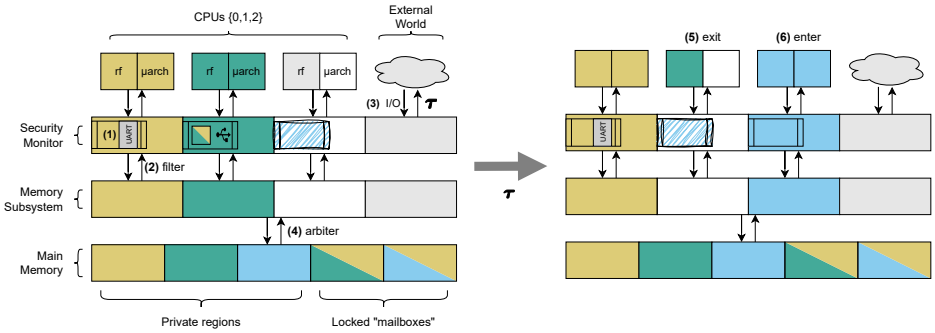
This chapter provides an overview of MTIsolation, discusses the threat model and security guarantee (Section 3.1), the implementation and specification systems (Figure 3-1), and the components needed to prove that the implementation implements strong timing isolation (Section 3.2). Figure 3-1a introduces the implementation or "real-world"[1] system and enclave programming model used as a motivating example throughout this part. Figure 3-1b shows a corresponding "ideal" system serving as a specification.

3.1 Threat model and security guarantee

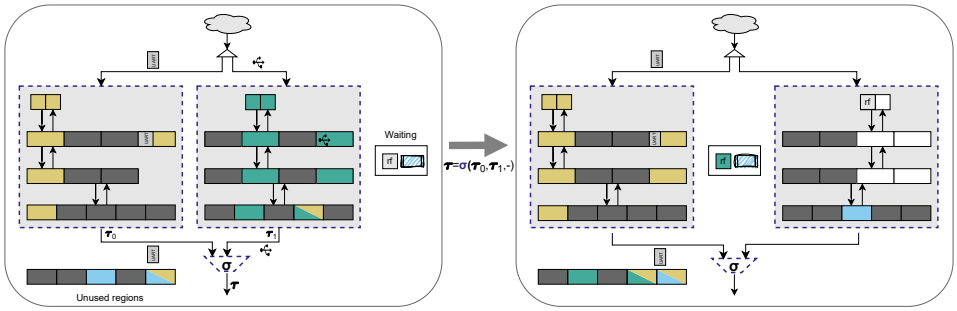
In our threat model, an attacker enclave attempts to extract secrets from or interfere with a victim enclave colocated on the same machine (either concurrently on a different core or after context switching on the same core). The attacker enclave can send memory and memory-mapped I/O (MMIO) requests; observe memory and MMIO responses, including response times; and yield the processor to another enclave.

We formally verify the strong-isolation property of Definition 1 for enclaves during execution (timelined in Figure 3-2), which guarantees that any information exfiltrated by a colocated attacker could be obtained by a remote attacker. As such, we do not protect against

[1] Terminology borrowed from the cryptography community.



(a) Implementation or "real-world system" with three cores, a security monitor (SM), memory subsystem, and main memory. Enclaves, denoted by different colours, have exclusive access to private memory regions and can obtain locks for "mailbox" and MMIO regions, held for the duration of execution. (1): The SM maintains an enclave configuration per core: `Core0` runs yellow enclave and has UART lock; `Core1` runs green enclave and has locks for USB and green-yellow mailbox; `Core2` waits for SM permission to enter blue enclave. (2): The SM enforces architectural isolation by filtering out-of-region memory accesses. (3): Secure communication with the external world via MMIO is enforced by SM. (4): Static, round-robin arbiter enforces memory noninterference. (5): `Core1` exits green enclave, purging microarchitectural (μarch) state but preserving architectural state (main-memory regions and register file, `rf`) and requests to switch to blue enclave. (6): `Core2` transitions from waiting state to running blue enclave.



(b) Specification or "ideal" system showing enclave execution (`Core0`), exit (`Core1`), and creation (`Core2`). Execution: enclaves run independently as "air-gapped" machines, modulo external communication. Exit: modeled as "throwing away" the old machine, preserving only architectural state and the next enclave configuration. Creation: modeled as initialising a "brand-new" machine with preserved architectural state and the enclave's memory regions. The specification is parameterised on non-security-critical implementation and timing details, denoted by dotted blue boxes. The σ parameter merges outputs from different enclaves as a function of enclave configs and cycle counter (elided in the figure).

Figure 3-1: A real-world system implementing, and corresponding ideal system specifying, strong isolation. An implementation is secure if there exists an ideal system such that, for any external-world oracle, the observations (per-cycle τs) are equivalent.

remote timing side channels such as NetSpectre [71]. Additionally, we prove that the system's execution aligns with our enclave programming model, which defines the memory and MMIO regions an enclave can access and the semantics and information-sharing policy of context

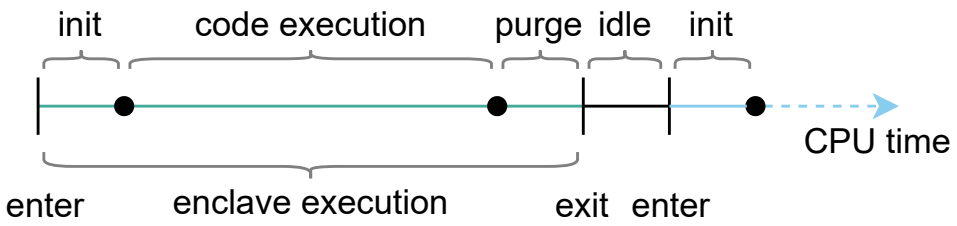


Figure 3-2: An example timeline of enclave execution on a core. We verify strong isolation holds throughout an enclave's lifetime or execution, inclusive of any initialisation, code execution, and purging. We specify behaviour on enclave context switching (exit and enter).

switching. For instance, our example model in Figure 3-1 specifies that the register file is preserved across enclave exits to facilitate argument passing. As enclave memory regions are exclusively owned, our model leaks information about which enclave is running (by blocking entry to running regions).

The programmer is responsible for ensuring that the program does not itself leak secrets with respect to a hardware-software contract [38], either directly through e.g. writing secrets to MMIO or indirectly by e.g. leaving secrets in the register file when (cooperatively) context switching or via enclave run time or tear-down time. Symmetrically, the hardware is responsible for adhering to the contract for programs run in isolation.

We do not consider availability or physical side channels.

3.2 Components to prove strong isolation

Here, we outline the components needed, and summarize our instantiations thereof, in a formal proof of strong isolation.

1) A formal, cycle-accurate model of our implementation. A hardware system includes synthesizable components (such as the processor) implemented in an HDL such as Verilog and non-synthesizable, "external" components such as SRAM traditionally modeled in simulation. This system must be designed to be secure with a clear separation of resources, whereas conventional architectures are not secure. We implement our system in Kôika and model external components in Rocq. We use a hardware security monitor to enforce architectural noninterference by controlling access to memory and MMIO regions. Microarchitectural noninterference is achieved by partitioning resources spatially or temporally. Upon context switching, we purge or flush microarchitectural state like scoreboards, outstanding memory requests, and caches to erase any program-dependent, non-preserved state that could affect future executions.

2) A formal, cycle-accurate model of our ideal system. The spec is part of our trusted computing base (TCB), and we must trust that it corresponds to our desired notion of strong isolation: we strive to make the model easy to audit. The spec must be cycle-accurate, but it need not be

synthesizable. As shown in Figure 3-1b, enclaves are modeled as "air-gapped" machines during execution (apart from explicit external communication) and context switching as "throwing away a machine and getting a new one" (apart from explicitly preserved state when exiting and explicit arguments when initialising).

To ensure that the spec is expressive enough to capture a wide range of secure implementations, we express the spec as a family of deterministic, cycle-accurate state machines parameterised on non-security-critical implementation and timing details. Intuitively, one must exhibit *an* air-gapped machine with the same timing as the implementation, but we do not constrain that timing further (beyond constraints imposed by the enclave programming model). With respect to security, these parameters are not in the TCB and hence do not need to be audited.

3) An observation function corresponding to the threat model. The observation function formalises what an attacker can observe and influence and relates the implementation to the specification. In our example, the observations τs are defined as the *cycle-accurate* interactions with the external environment (MMIO requests/responses)[2].

An implementation is secure if it has the same observable behaviour as the spec, for any external environment.

4) A machine-checked proof that the implementation is secure. We avoid the need to prove functional correctness according to an ISA semantics. In our verification methodology, we emphasize *modularity*, to allow independent design and verification of hardware components; and *automation*, via static analyses and selective usage of SMT solvers to reduce verification cost.

[2]Traditional approaches often describe observations as memory requests and responses. Our definition captures the intuition that it is not a security violation if an attacker obtains a victim's secret if the attacker cannot exfiltrate it (e.g. by communicating it to the outside world).

Chapter 4

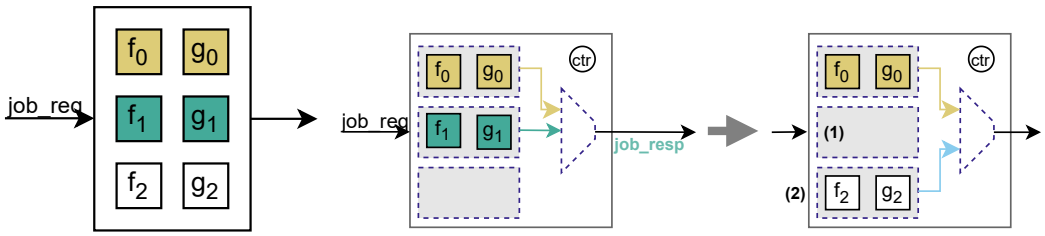
Specifying and Enforcing Isolation

In this section, we first illustrate the key points of specifying and achieving strong isolation with a toy example and then apply these principles in the enclave-isolation setting.

4.1 Toy example: resource isolation

Consider a machine taking jobs $(m, n, x) \in \mathbb{N}^3$ and computing $g^n(f^m(x))$ (for some combinational functions f and g) using three[1] f and g boxes, as shown in Figure 4-1a. The machine guarantees independence of job-completion time (and result) from any other concurrently executing job. An implementation can meet the requirement by running at most three jobs at a time—eagerly reserving or *spatially partitioning* an f and g box per job—and *temporally partitioning* the output port `job_resp` with a round-robin arbiter. Potential sources of security violations in insecure implementations include:

[1] Our verified implementation uses two sets of boxes—we use three here to illustrate different state transitions. It also assigns tags to jobs, elided here for simplicity, that can be used to demultiplex responses by consumers of the output.



(a) Implementation (b) Specification step transition

Figure 4-1: (a): Implementation system with single job-request input port and single job-result output port. (b): Step transition of the specification system running two jobs on air-gapped machines. (1): Finishing a job modeled as throwing away old machine. (2): A new job starts from fresh state.

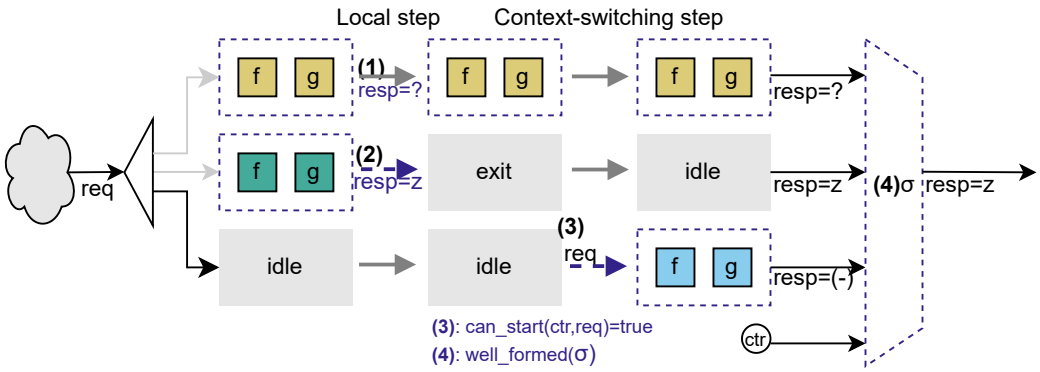


Figure 4-2: Resource-isolation specification's state transition, split into local and context-switching steps. (1): When running, subsidiary machines behave independently, with no inputs. (2): When finishing, no local machine state is preserved. (3): If a job `can_start`, a brand-new machine is initialised with only the new job. (4): σ combines the outputs from subsidiary machines. I/O arrows abbreviated for intermediate steps.

1. Backpressure when trying to run more than three jobs simultaneously and all g boxes are occupied (i.e. when a job has completed its f phase on f_0 and is only using g_0, it would be faster to begin another job on f_0). This issue is analogous to backpressure caused by contention on finite cache bandwidths.

2. Output-port contention in the absence of time partitioning: if two jobs finish simultaneously and output-port contention delays the output (and thus, completion time) of a job, then isolation is broken.

3. Performance optimizations leading to results being forwarded between e.g. f boxes, or caching intermediate results (e.g. if the machine expects to receive many jobs with the same x value).

4.1.1 Specification

Figure 4-1b summarizes a spec system ruling out the above violations. The state consists of three subsidiary air-gapped machines, each either `Running` or `Waiting`. Figure 4-2 details the state transitions, separated into two phases: 1) a local step where jobs run in isolation and 2) a context-switching step (that is, a step where hardware units switch to servicing different client requests) specifying job exit (no state is preserved) and job start (initialised only with the new job). The spec is parameterised on f and g boxes, a `can_start` function, and a *well-formed* σ function combining outputs from subsidiary machines. This definition guarantees that independently of the implementation of the f and g boxes, concurrent executions cannot affect or observe the runtime or result of another job.

4.1.2 Spec expressivity and audit: capturing a range of secure implementations using existentially quantified parameters

From an isolation perspective, the functional behaviour of the implementation and thus of subsidiary machines is unimportant (i.e. machines do not have to compute $g^n(f^m(x))$ or have f or g boxes). The implementation is secure if there *exist* subsidiary machines such that the specification is observably indistinguishable from the implementation, for all external-world behaviours (inputs to `job_req`).

In other words, the spec is *existentially parameterised* on deterministic state machines $(S, s_0, \delta, \omega)_i$, for $i \in \{0, 1, 2\}$, where S_i is the set of states, $s_{0_i} : \texttt{job\_req} \to S_i$ is the initial state as a function of job request, $\delta_i : S_i \to S_i$ is the transition function, and $\omega_i : S_i \to \{0, 1\} \times \texttt{job\_resp}$ is the output function with a bit indicating job completion. Then, n cycles after a $\texttt{req} \in \texttt{job\_req}$ is accepted, the machine i has state $\delta_i^n(s_{0_i}(\texttt{req}))$. The job-completion time is $t_{e_i} := \min\{t \in \mathbb{N} \mid \pi_1(\omega_i(\delta_i^t(s_{0_i}(\texttt{req})))) = 1\}$, and the job response is $\pi_2(\omega_i(\delta^{t_{e_i}}(s_{0_i}(\texttt{req}))))$. Clearly, completion time and response are independent of concurrently executing jobs. This property is extended to the trace through a *well-formedness* predicate on σ (the parameterised output-merging function) specifying that outputs at any given cycle must come exclusively from one statically predetermined machine. The parameterisation of σ allows different static schedules, the choice of which is implementation-specific and non-security-relevant.

This *family of specifications existentially parameterised on non-security-relevant design choices* style matches intuitive notions of security, decoupling security properties from functional-correctness properties, thereby enabling the spec-reader to avoid reading low-level design details. The top-level spec remains concise, because the (possibly quite complex) state machines and logical rules that get substituted for the parameters are not in the TCB. This approach also allows one compact spec to support a wide range of implementations.

4.1.3 Instantiating the spec

The subsidiary machines are instantiated according to the implementation with single f and g boxes and arbitration logic. The `can_start` function is implementation-specific and not security-critical and hence is existentially parameterised, supporting a range of job-assignment implementations. For example, jobs can be assigned to unoccupied machines in priority order or to certain machines based on the request (e.g. to take advantage of an f accelerator or when $m = 0$). σ is instantiated based on the implementation of arbitration, e.g. by returning the response from machine $n \bmod 3$ at cycle n for a round-robin arbiter.

4.2 Static strong isolation

We extend the techniques from Section 4.1 in the context of processors and a memory system running hardware enclaves, with communication defined by the enclave programming model (e.g. MMIO), as in Figure 3-1. As in Section 4.1, we decompose the spec into two parts, pertaining to the semantics of 1) enclave execution while running and 2) enclave context switching. We are guided by the principle of using allowlists instead of blocklists: we use "air-gapped" machines as a starting point, explicitly add I/O as defined by the enclave programming model, and then define the semantics of context switching on top of the baseline "throwing away" the old machine and starting a "new" machine. We discuss and formalise 1) in this section and 2) in Section 4.3.

Isolation without context switching. Consider a machine M with m processors running m enclaves initialised with secrets (e.g. enclave memory contents) sec_k for $1 \leq k \leq m$. Intuitively, in the absence of context switching, the machine is secure if it can be expressed as an observationally indistinguishable *product machine* with m submachines M_k, each initialised with corresponding sec_k. More formally:

Definition 2 (Static strong isolation with fixed inputs)**.** Suppose we have an implementation machine[2] $M := (S, s_0, I, O, \delta, out)$ where S is the set of machine states, $s_0 : \{Sec\}_m \to S$ is the initial state as a function of m enclave secrets, I and O are the input and output types, $\delta : S \times I \to S$ is the (cycle-accurate) transition function, and $out : S \times I \to O$ is the output function. Given inputs $\iota_1, \ldots \iota_n$ and m enclave secrets $\{sec\}_m$, the machine steps as follows after n cycles:

$$s_0[\{sec\}_m] \xrightarrow[o_1]{\iota_1} s_1 \xrightarrow[o_2]{\iota_2} \cdots \xrightarrow[o_n]{\iota_n} s_n$$

The trace is defined as the (cycle-accurate) sequence of outputs $o_1, \ldots, o_n$. M securely implements isolation without context switching if there exists, for a given input-partitioning function π, a product machine $\Pi := (\{M\}_k)$, and well-formed[3] output-joining function σ such that, for each M_k and enclave secret sec_k,

$$s_{0_k}[sec_k] \xrightarrow[o_{1_k}]{\pi(k,\iota_1)} s_1 \xrightarrow[o_{2_k}]{\pi(k,\iota_2)} \cdots \xrightarrow[o_{n_k}]{\pi(k,\iota_n)} s_{n_k}$$

and Π's output $o_{\Pi_i} := \sigma(o_{i_1}, \ldots, o_{i_m})$ is equal to M's output o_i at every cycle i. π and σ denote functions splitting inputs and combining outputs respectively.

Trace equivalence here expresses that the implementation's outputs can be constructed from separate machines initialised with separate secrets. We detail the modeling of inputs

[2] A finite Mealy machine.

[3] As in Section 4.1, a well-formedness property captures isolation constraints on the implementation's output such as "Outputs for the UART must come from the enclave currently owning the lock to the UART".

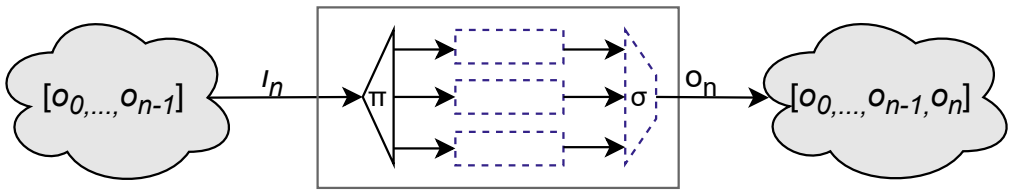


Figure 4-3: External world as a function over output history.

and outputs next and then provide intuition for sources of leakage ruled out by Definition 2.

Nondeterministic I/O. A challenge with I/O is handling nondeterminism: secrets can be leaked through nondeterminism in the specification. Our strategy for I/O is to model the external world as a deterministic "oracle" state machine Ω, ensuring the resulting implementation and specification systems remain deterministic, and prove trace equivalence for all possible Ωs. Without loss of generality, the external world's state can be defined to be the (cycle-accurate) history of external observations (outputs) of the implementation or specification machines as shown in Figure 4-3. Then, at each cycle, the external world generates an output (used as input to the enclave machine) as a function out_ω of its state.

In other words, $\Omega_{out_\omega} := (S, s_0, O, I, \delta, out_\omega)$[4] with

$$s_{0_\omega} \xrightarrow[\iota_1]{o_1} s_{1_\omega} \xrightarrow[\iota_2]{o_2} \cdots \xrightarrow[\iota_n]{o_n} s_{n_\omega}$$

$s_0 := []$ the initial state with no history, $\delta(s, o) := s{+}{+}[o]$ updating the external world with the enclave machine's output, and $\iota_k := out_\omega(s_{k-1_\omega})$ the enclave machine's input at cycle k. These enclave inputs (oracle outputs) can be censored or partitioned in some way, via the π function, and there can be additional properties on the out_ω depending on the threat model. For example, we might enforce that USB responses are dependent only on previous USB requests. The oracle function is incorporated into the state-transition systems of the implementation and specification, which remain deterministic. In other words, (M, Ω_{out}) and (Π, Ω_{out}) are deterministic state machines.

Definition 3 (Strong isolation without context switching)**.** M is secure for a given input-partitioning function π and output-joining function σ if $\exists \Pi.\ \forall out_\omega.\ (M, \Omega_{out_\omega}) \equiv (\Pi_{\pi,\sigma}, \Omega_{out_\omega})$ where $\equiv$ denotes trace equivalence.

[4] A Moore machine.

4.2.1 Security audit and enforcing isolation with spatial and temporal partitioning

Modern microarchitectures generally do not satisfy Definition 2 and Definition 3. Consider enclaves Enc_i and Enc_j colocated on the same machine. To satisfy Definition 3 and view Enc_i and Enc_j as running on separate machines M_i and M_j, we must eliminate contention through spatial and temporal partitioning. We provide examples of leakages, discuss how they are ruled out by the specification, and provide example enforcement techniques. Thanks to the allowlist approach, the spec does not explicitly enumerate attacks of leakage sources such as caches, allowing it to rule out attacks not yet discovered.

Enc_i and Enc_j try to access the same address. By assigning Enc_i's main-memory region exclusively to sec_i and Enc_j's to sec_j, Definition 3 rules out such architectural leakages as M_i does not have access to sec_j. To enforce this architectural isolation, we assign Enc_i and Enc_j separate address spaces by, e.g., partitioning the main memory into disjoint regions and using a security monitor (SM) to filter out-of-region requests.

Dynamic contention for L2 cache sets. If Enc_i and Enc_j access physical addresses in the same L2 cache set, then accesses from Enc_i can evict entries from Enc_j, which allows Enc_j to observe a timing leakage from a cache miss versus a hit. Definition 3 rules out such leakages as M_i and M_j simply do not share a cache and run as independent state machines. The MI6 fix is to partition the L2 cache spatially through set partitioning, such that the main-memory regions map to disjoint cache sets, and each set is exclusively owned by a single enclave. The memory subsystem must ensure that if memory requests from different cores are from different regions, then there should be no interference.

Port contention for cache-access pipeline or DRAM. Messages from different cores arriving at the single entry of an L2 cache may block each other for a cycle, leading to timing leakage. Definition 3 rules out such leakages, as enclaves do not share ports. A solution is static temporal partitioning of access to the port using, e.g., a round-robin arbiter enforcing that Enc_i can only access the port on even cycles and Enc_j on odd cycles. This pattern of having a multiplexer (mux) choosing between n different inputs is common: a similar round-robin technique must be applied whenever there is a mux of inputs from different protection domains.

4.3 Dynamic enclave isolation

There are subtleties with defining secure context switch: what information is preserved?; what is the initial state of a new enclave?; what if two cores want the same enclave? To implement secure context switching, we erase program-dependent, non-preserved state that could affect future executions. We model context switching as "throwing away the old machine and starting a new one," initialised based on the programming model. We use an allowlist instead of a blocklist to require *explicit* authorization of information preserved upon exit and used in entry. Figure 5-1 contains an example from our case study, sketched out below.

Exit. We extend the spec with state preserved $S_{preserved}$ and (existentially parameterised) functions $\texttt{extract\_preserved}_k : S_k \to S_{preserved}$ and $exit : S_k \times I \to \{0, 1\}$ (analogous to the job-finished bit of `job_resp` from Section 4.1). E.g.: `extract_preserved` returns the register file and DRAM, and `exit` returns 1 when the SM releases enclave resources. We illustrate informally below, eliding I/O, in a single-core setting with preserved state s_p:

$$(s_0, s_p) \xrightarrow[\texttt{!exit}]{} (s_1, s_p) \xrightarrow[\texttt{!exit}]{} \cdots \xrightarrow[\texttt{exit}]{} (s_n, s_p) \xrightarrow{\texttt{extract}} ((), s_{p'})$$

Start. Defining start involves extending the specification with an existentially quantified `can_start` function (with arguments specifying what information can be used to determine whether an enclave can start) and `init` function determining what information is used to initialise an enclave (e.g. the register file and enclave memory regions). A *well-formedness predicate* can be imposed on `can_start`, for example to ensure that enclaves access disjoint regions or to enforce a policy on which enclaves a given enclave can switch to. Diagrammatically:

$$((), s_{p_0}) \xrightarrow[\texttt{!can\_start}]{} ((), s_{p_1}) \xrightarrow[\texttt{!can\_start}]{} \cdots \xrightarrow[\texttt{can\_start}]{\texttt{init}} (s_0, s_{p'})$$

Definition 4 (Strong isolation with context switching)**.** Let $\texttt{Spec}(\Pi_{\pi,\sigma}, \texttt{extract}, \texttt{exit}, \texttt{can\_start}, \texttt{init})$ be the deterministic state machine defined by its existentially quantified *parameters,* with enclave k transitioning according to Π_k when running, exiting according to `exit` with state preserved defined by `extract`, and starting according to `can_start` with initial state defined by `init`. M is secure for a given input-partitioning function π and output-joining function σ if there exists `params` such that $\forall out_\omega.(M, \Omega_{out_\omega}) \equiv (\texttt{Spec}(\texttt{params}), \Omega_{out_\omega})$ where $\equiv$ denotes trace equivalence.

Chapter 5

Case Study: Multicore RISC-V Processor

As a case study, we specify, implement, and verify strong isolation for a two-core enclave system (shown in Figure 3-1[1]) for the following enclave programming model.

Enclave programming model. The address space is statically partitioned into regions exclusively owned by enclaves and "locked mailbox" regions shared by pairs of enclaves (but

[1] Our implementation has two cores. The figure has three cores to show different state transitions.

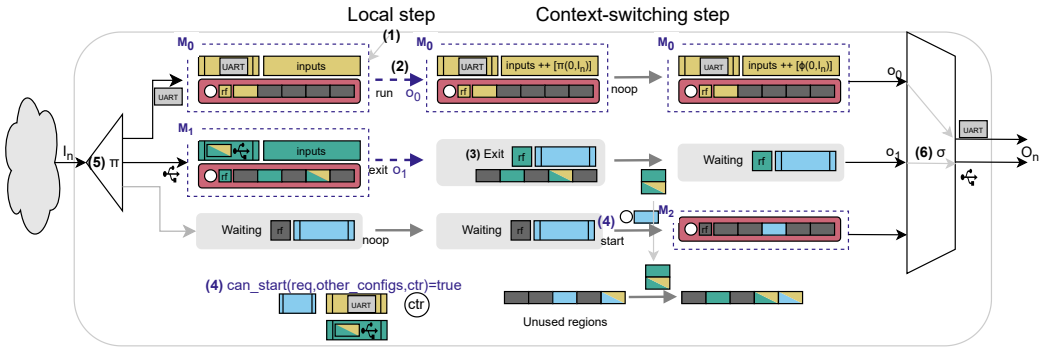


Figure 5-1: Spec system's transition corresponding to Figure 3-1, decomposed into local and context-switching steps. (1): `Enclave0` runs yellow enclave with UART. At cycle n, `Enclave0`'s state is a (parameterised) function of its initial state (denoted by the red box, containing cycle counter `ctr`, register file `rf`, and memory regions) and n UART inputs received thus far. (2): `Enclave0` independently steps, outputting o_0, and is now a function of its initial state and the $n + 1$ inputs. (3): When exiting, the `rf`, enclave memory regions, and requested config (blue) are preserved. The exiting enclave's memory regions are returned to the spec's map of unused regions. (4): `can_start` is a function of the requested config, configs in use, and public state (i.e. `ctr`). Its *well-formedness condition* enforces config disjointness. If a machine `can_start`, it is initialised with `ctr`, previous `rf`, and memory regions. Else, it remains idle. (5): The external world (MMIO) input I_n is split with π based on enclave config, with UART inputs passed to `Enclave0` and USB inputs to `Enclave1`. (6): σ combines enclave outputs to generate the spec's output O_n, with UART outputs from `Enclave0` and USB outputs from `Enclave1`.

exclusively owned during execution). An enclave runs on a single core and is configured with a set of mailbox regions and locked MMIO regions. Enclaves can send memory and MMIO requests and receive responses and measure their latencies. An enclave has exclusive ownership over its regions, cannot access other regions, and runs in isolation.

Enclaves communicate via the external world through MMIO or via the register file (rf) or mailboxes after context switching. Preserving the rf enables more efficient implementation of enclave calls (i.e. `Enclave0` calls a function from `Enclave1` with arguments in standard ISA argument registers, and `Enclave1` returns control to `Enclave0` with the result). The programmer is responsible for removing secrets from the rf and saving e.g. stack and frame pointers to resume execution. Alternatively, mailboxes allow exchanging more data. A mailbox shared by `Enclave0` and `Enclave1` can be written to by `Enclave0` and later read by `Enclave1`, after `Enclave0` exits.

Enclaves cooperatively yield to other enclaves, by requesting to switch to new configurations. When exiting, enclaves preserve their memory regions and register files. A new enclave can only be created with a config if no other enclave's config conflicts (this check leaks information about enclave configs and runtime). A new enclave is initialised as a fresh machine with assigned memory regions and register file.

5.1 Specification

Figure 3-1b summarizes a spec for our programming model. Figure 5-1 details the spec's state-transition function, showing information preserved upon exiting and used in initialisation. Figure 5-1 illustrates that after running for n cycles, an enclave can be viewed as an (existentially parameterised) function exclusively of its initial state (register file, memory regions, and cycle counter) and the n MMIO inputs from the external world. π splits the MMIO inputs according to enclave config such that, e.g., `Enclave0` receives inputs for UART addresses and `Enclave1` for USB addresses. σ combines the enclaves' outputs according to the configs (e.g. UART output is equal to the output from the enclave owning the UART). This spec allows a wide range of implementations by imposing minimal restrictions on functional correctness, while ruling out any interference not via MMIO while enclaves are running. We mechanise this spec in Rocq, shown in Figure 5-2.

5.2 Implementation

We instantiated our approach with two pipelined RISC-V cores, a security monitor, and a memory subsystem arbitrating access to a single-port BRAM (extended with caches in Section 7.1). The SM enforces architectural isolation and the enclave programming model. The processor and memory are responsible for purging microarchitectural state upon context switching, reaching a state functionally equivalent to a new machine initialised only with state

```
Inductive core_state_t :=
| CS_Running (st: machine_st) (config: enclave_config)
| CS_Waiting (new: enclave_config) (rf: reg_file_t) (exit_cycle: nat).
Definition ctx_switch_data:= enclave_config * dram_t * reg_file_t.
Inductive transition_state_t :=
| TS_Exit (new: enclave_config) (ctx: ctx_switch_data) (obs: local_obs_t)
| TS_Core (st: core_state_t) (obs: local_obs_t).
Record state_t :=
  { core_st: core_id → core_state_t; mem_regions: mem_region → dram_t; cycles: nat }.
Definition step_local (core: core_id) core_state input :=
  match core_state with
  | CS_Running mst config ⇒
     let (mst',obs) := step_running core mst input in
     match obs.obs_exit_enclave core with
     | Some config' ⇒ TS_Exit config' (config, extract_dram mst', extract_rf mst' core) obs
     | None ⇒ TS_Core (CS_Running mst' config) obs
  | CS_Waiting new rf exit_cycle ⇒ TS_Core core_state empty_obs.
Definition step_enter core ts_state (other_config: option enclave_config)
         cycles mem_regions :=
  match ts_state with
  | TS_Exit _ _ _ ⇒ ts_state
  | TS_Core (CS_Running _ _) _ ⇒ ts_state
  | TS_Core (CS_Waiting config' rf t_exit) obs ⇒
    if does_not_conflict new other_config
       && can_start core t_exit cycles config' other_config then
       let machine := spin_up_machine core (cycles+1) new
                      (get_dram enclave_params mem_regions config') rf in
       TS_Core (CS_Running machine config')
    else ts_state.
Hypothesis wf_can_start: forall t_exit0 t_exit1 cycles new0 new1,
              conflicts new0 new1 →
              can_start Core0 t_exit0 cycles new0 None = true →
              can_start Core1 t_exit1 cycles new1 None = true → False.
Definition step_exit ts_state mem_regions cycles :=
  match state with
  | TS_Exit new ctx obs ⇒
    let regions' := update_regions enclave_params ctx.config ctx.dram mem_regions in
    (CS_Waiting new ctx.rf cycles, obs, regions')
  | TS_Core st obs ⇒ (st,obs,mem_regions).
Definition spec_step (st: state_t) input : state_t * output_t :=
  let ts0 := step_local Core0 (st.core_st Core0)
                      (filter_input (get_config (st.core_st Core0)) input) in
  let ts1 := step_local Core1 (st.core_st Core1)
                      (filter_input (get_config (st.core_st Core1)) input) in
  let ts0' := step_enter Core0 ts0 (get_config (st.core_st Core1))
                       st.cycles st.mem_regions in
  let ts1' := step_enter Core1 ts1 (get_config (st.core_st Core0))
                       st.cycles st.mem_regions in
  let (cst0', obs0, mem0) := step_exit ts0' st.mem_regions st.cycles in
  let (cst1', obs1, mem1) := step_exit ts1' mem0 st.cycles in
  ({| core_st := fun c ⇒ match c with | Core0 ⇒ cst0' | Core1 ⇒ cst1' end;
       mem_regions := mem1; cycles := st.cycles +1 |},
   (merge_external_observations obs0 obs1)).
Definition step input_machine (st: state_t) (ext_world: ext_world_state_t)
              : state_t * ext_world_state_t * output_t :=
  let input := input_machine.get_input ext_world in
  let (st', output) := spec_step st input in
  (st', input_machine.ext_step ext_world output, output).
```

Figure 5-2: Specification of strong isolation in Rocq. The inputs and outputs are bitvectors containing MMIO requests and responses. The specification is parameterised on the text in red. Each core running an enclave takes a local step `step_running` independently of other cores. A waiting core begins executing an enclave if it `does_not_conflict` with other enclaves and `can_start`, initialised with the register file, memory regions, and cycle counter.

in the allowlist. The memory is also responsible for guaranteeing that, from a purged state, if requests from different cores are from disjoint configs then it should behave for each core as if it were two separate subsystems.

Purge state machine. Upon receiving a switch request, the SM instructs the core and memory to purge microarchitectural state. The purge state machine has three states: i) `Ready`: the CPU/memory are running an enclave, ii) `Purging`: the CPU/memory are requested to purge microarchitectural state, and iii) `Purged`: the CPU/memory have purged and await SM instruction to start a new enclave execution.

Processor. The processor is based on the four-stage RISC-V core from [16]. The processor tracks in-flight instructions via queues, bookkeeping state, a scoreboard for detecting hazards, and state to track speculation/branches (extended with a BTB and BHT for prediction and an epoch mechanism to correct misprediction in Section 7.1). The processor is connected to the SM via pairs of FIFOs (instruction memory, data memory, and MMIO) and uses a custom RISC-V instruction to switch enclaves. Upon executing a switch request, the processor drains its pipeline and purges microarchitectural state.

Security monitor. The hardware SM stores an enclave config per core, consisting of enclave ID, any locked mailboxes, and reserved MMIO addresses. The SM filters out-of-enclave address requests and arbitrates access to the shared MMIO bus. Upon receiving a switch request, it instructs the core and memory to start `Purging` and waits until they are `Purged` to exit. The SM waits until no other enclave owns requested regions before allowing enclave entry.

Memory subsystem. We implement two memory subsystems: a simple, round-robin arbiter (described below) and a system with L1 caches (see Section 7.1). Both are connected to the SM via four pairs of FIFOs (instruction and data memory per core) and interface with a single-port, single-cycle SRAM shared by the cores. The subsystem maintains queues per-core and shares "status-holding request" registers tracking requests in the shared SRAM queue. A round-robin arbiter ensures requests do not contend for SRAM bandwidth. To avoid backpressure, the system ensures that there is enough space in the FIFOs to reply to the SM, before sending a request to SRAM. Finally, it executes a purge state machine (flushing caches and draining FIFOs) after ensuring there are no in-flight requests per-core.

Chapter 6

Verifying Strong Isolation

Our approach enables a *modular proof*, allowing architects and verifiers to implement and prove security properties of submodules independently. We found it valuable to state modular security obligations independently of functional correctness, allowing verifiers to prove security without proving functional correctness. We feature a hybrid Rocq-SMT approach, using Rocq's expressivity for clear specs and to reduce security to single-cycle, circuit-level properties amenable to automatic verification with SMT solvers. We summarize the proof architecture in Section 6.1, discuss instantiation of existentially quantified parameters in Section 6.2, describe per-component obligations in Section 6.2.1, and detail MTIsolation (the Rocq-SMT toolchain) in Section 6.3.

6.1 Proof architecture

Figure 6-1 summarizes our verification framework. The spec is written in Gallina (Rocq's specification language) and the implementation in Kôika (for synthesizable components) and Gallina (for models of SRAM). To capture a range of implementations, the spec is parameterised on non-security-critical behaviours. We leverage Kôika's verified compiler to circuits to generate semantically equivalent circuits.

We use an (unverified) translation from Kôika to δKôika, a derivative of Kôika that is syntactically and semantically nearly identical to Kôika's reference implementation [1]. δKôika was designed with a term representation to support verification and crucially extends Kôika's semantics with support for reasoning about external functions.

Modular decomposition. Kôika has non-local interactions between rules and no first-class support for modules: a rule may fail to execute based on "conflicts" with what happened previously (dynamically) in a cycle[1]. To restore modularity, we use static analyses (verified in

[1] A register can only be written to once in a cycle. In Kôika, to enable data forwarding or pipelining in the same cycle between rules, reads and writes to registers are associated with ports P_0 and P_1, and certain orderings such as $\texttt{write}_1$ followed by $\texttt{read}_0$ lead to conflicts and a rule failing.

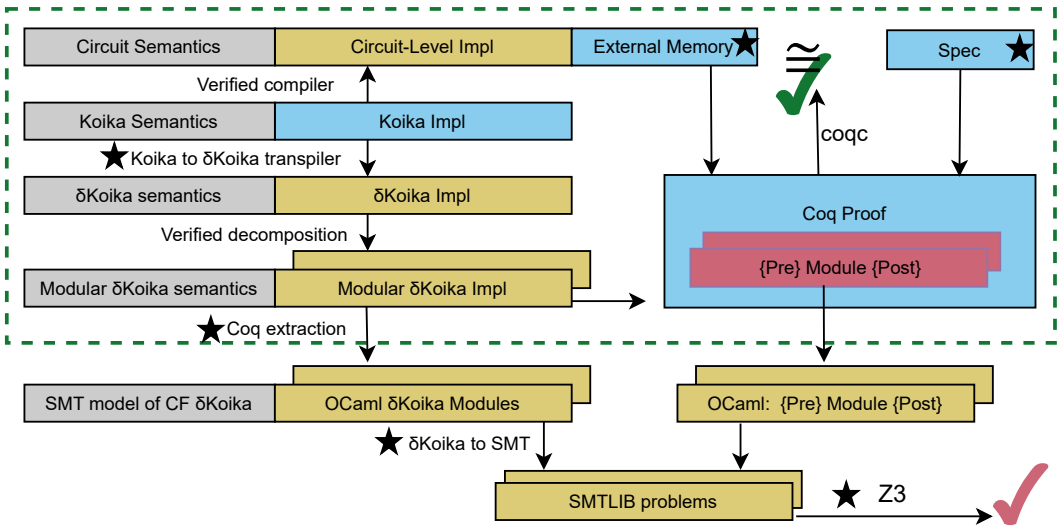


Figure 6-1: Architecture for proving that a circuit-level implementation satisfies a specification. Grey boxes are provided by MTIsolation. Blue boxes are implemented by the developer. Yellow boxes are intermediate steps generated by MTIsolation. Red boxes denote assumptions within Rocq, validated using Z3. The green-dotted box indicates parts within Rocq. The star indicates components in the TCB.

Rocq) to decompose the design into per-component semantics by checking that rules do not conflict across modular boundaries. We implement our designs in a style such that we can guarantee statically that rules do not conflict across component boundaries, enabling a "one-component-at-a-time" semantics. We expect modular reasoning to be more straightforward with HDLs with good support for modularity.

Proving trace equivalence using state-machine refinement. After instantiating the Rocq spec based on the implementation (see Section 6.2), the proof developer states (in Rocq) a simulation relation between the spec and implementation along with a single-cycle, circuit postcondition (e.g. MMIO calls are adequately related). The developer proves (in Rocq) that strong isolation (for an unbounded number of cycles) can be reduced to the single-cycle preservation of the simulation relation and the postcondition implying trace equivalence.

Hybrid Rocq-SMT approach. The simulation relation and postcondition are composed of per-component obligations (preconditions and postconditions) expressed in MTIsolation's custom symbolic assertion language. MTIsolation extracts the assertions into OCaml, translates the assertions into SMT-LIB via a Kôika-to-SMT encoding (Section 6.3), and checks the generated SMT formulae using Z3 [30]. The developer strengthens the invariant as needed, if MTIsolation finds a counterexample. After this process, the developer obtains a proof that the circuit-level design implements strong isolation.

6.2 Instantiating the specification

The developer must instantiate existentially quantified parameters in the specification by, e.g., providing a state-machine definition for an enclave running in isolation. We found that a convenient template to instantiate the enclave's state machine is to take the underlying implementation and essentially erase or prove unused circuitry for other cores. As an example, one could take the implementation and disable fetch for other cores or delete other CPUs.

With an implementation expressed in a rule-based language designed with a separation of resources between security domains, we simply erased rules corresponding to other cores. For example, to instantiate the state machine for `Core0`, we removed the processor corresponding to `Core1` and memory/SM rules exclusively performing computation for `Core1`. For rules performing computation for both cores, we could either rewrite the rules to remove `Core1`'s computation or provably maintain the invariant that `Core1`'s circuitry does not affect `Core0`'s computation. For simplicity and to avoid rewriting rules, we chose the latter. This instantiation style actually allows us to prove a stronger property about the architectural behaviour of an enclave (see Section 7.4).

6.2.1 Per-component security obligations

The proof of strong isolation can be decomposed into two parts: i) enforcing a simulation relation between running enclaves in the implementation and specification through spatial and temporal partitioning, and ii) reaching and maintaining a state functionally equivalent to a new machine when context switching through purging. These two properties are amenable to a modular proof with per-component obligations of a similar form: each individual component must i) spatially and temporally partition resources and ii) purge program-dependent state when context switching. The modular decomposition requires stating additional per-component guarantees and assumptions (e.g. the SM guarantees that addresses from disjoint cores are from disjoint enclave regions, and the memory subsystem guarantees that it behaves like two separate memory subsystems assuming addresses from disjoint cores are from disjoint regions). Establishing these per-component specifications is useful from a design standpoint, clarifying guarantees and assumptions. Note that this decomposition is not limited to Kôika and is fundamental: e.g. purging operations are mostly local to individual components.

Processor. There are minimal constraints on the processor when running (as it is owned by a single enclave), but it must guarantee that it reaches a state functionally equivalent to a new machine when purging. Purging does not require *resetting* all microarchitectural state: there are multiple states equivalent to an empty processor pipeline (e.g. if implementing a queue with a circular buffer, any configuration where head and tail pointers are equal maps to an empty state and indeed, our implementation empties FIFOs without purging all data registers), and any such configuration is indistinguishable. The simulation relation

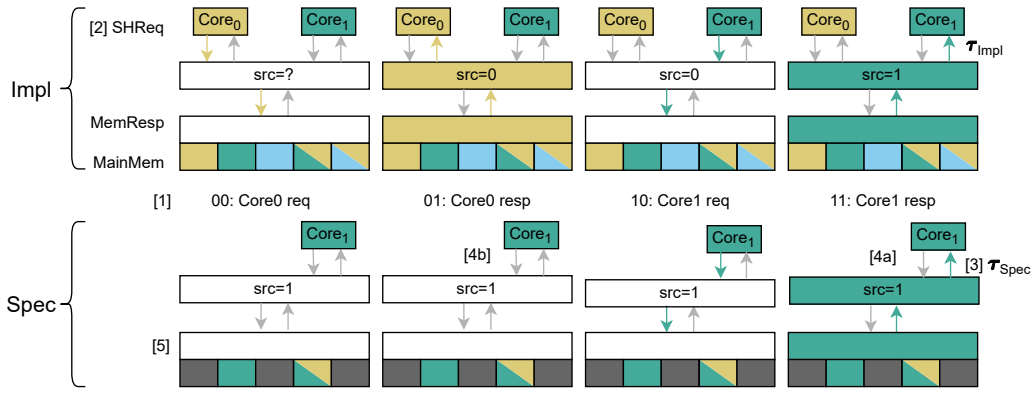


Figure 6-2: Memory arbiter for single-cycle, single-port memory. [1] Access to memory is time-partitioned: on cycles `00` and `01`, `Core0` and `Core1` respectively can exclusively send requests. [2] Status holding request (SHReq) register tracks the source of the outstanding request. [3] Specification must guarantee $\tau_{Impl} = \tau_{Spec}$, assuming memory requests from different cores access disjoint regions. [4a] If memory has an outstanding response, then $\texttt{SHReq}_{\texttt{Impl}} = \texttt{SHReq}_{\texttt{Spec}}$. [4b] Else, they are not necessarily related (e.g. at cycle `10`, the SHReqs hold different values). [5] Specification maintains invariant that there are only responses at cycle 11. So, the memory is self-purging. At cycle e.g. `10`, `Core0` is guaranteed not to have outstanding memory responses and can enter the `Purged` state.

captures this notion of equivalence. Note that the processor's proof obligation is independent of functional correctness and does not mention ISA semantics.

Security monitor. The SM's guarantees include that enclaves start only when both memory and core have purged and the core-to-memory pipeline is flushed, configs are disjoint, and requests forwarded to memory are within allowed memory regions. As a result, the memory may assume that, from a purged state, requests from cores will come from disjoint regions. The SM also arbitrates access to MMIO, with the simulation relation stating that MMIO requests from the implementation and spec are related based on enclave configuration, and an invariant maintains that enclaves do not have outstanding requests to unowned MMIO regions.

Memory subsystem. The memory subsystem implements a purge state machine similar to the processor's, but it must further act as two independent state machines *assuming* that requests from different cores are from disjoint enclave regions from a purged state. The memory subsystem must spatially and temporally partition resources across cores, maintaining invariants about requests made to caches or main memory. For example, the memory subsystem guarantees requests to main memory must previously have been received from the SM. The memory subsystem temporally partitions access to a single-port main memory with a two-bit arbiter, shown in Figure 6-2 with example invariants. We discuss invariants

```
Inductive mid_t := MachineImpl | MachineSpec.
Inductive sval :=
| SConst:list bool→ sval | SGetField:mid_t→ field_t→ sval→ sval
| SBits1:bits1→ sval→ sval | SBits2: bits1→ sval→ sval→ sval
| SExtCall:mid_t→ id_t→ sval →sval | SBound: string→ sval
| S{Init,Mid,Final}Reg: mid_t→ reg_t→ sval
| S{Mid,Final,Committed}Ext: mid_t→ fn_t→ sval
Inductive spred :=
| PConst: bool→ spred | PNot: spred→ spred
| PAnd : spred→ spred→ spred | POr : spred→ spred→ spred
| PImpl: spred→ spred→ spred | PExtFnEq: fn_t
| PEq: sval→ sval→ spred | PNEq: sval→ sval→ spred
| PForallArg1: (string*nat)→ spred→ spred
| PForallArg2: (string*nat)→ (string*nat)→ spred→ spred
| PFancy: fancy_spred→ spred
with fancy_spred :=
| PBase: spred→ fancy_spred
| PForallRegs: (reg_t→ fancy_spred)→ list reg_t→ fancy_spred.
```

Figure 6-3: Symbolic assertion language deeply embedded in Rocq

pertaining to caches in Section 7.1.

6.3 Kôika-to-SMT toolchain

The proof developer expresses pre- and postconditions (over a sequence of rules, per module or otherwise) in a symbolic language deeply embedded in Rocq provided by MTIsolation, shown in Figure 6-3. These symbolic predicates express cycle-granularity conditions over registers and external function calls before and after executing a sequence of rules and are associated with a symbolic interpretation that *reflects* the predicates back into Rocq propositions for use in proving strong isolation for an unbounded number of cycles. We additionally chain together the module specifications, asserting that postconditions of one module imply the precondition of the next. MTIsolation's SMT encoding of Kôika discharges these conditions to Z3. This approach soundly leverages the automation of SMT solvers while being largely immune to their scalability challenges and unpredictable performance: we use SMT solvers for single-cycle verification (comparable to assertion-based verification in traditional hardware verification) while proving the soundness of the reduction and metatheory in Rocq. MTIsolation supports the developer in finding design bugs as well as iterating on invariants: if a counterexample is found, MTIsolation outputs violating assertions and provides a query interface to inspect register assignments of the counterexample at rule boundaries.

Symbolic assertions. Figure 6-4 shows examples of assertions in MTIsolation's symbolic language in Rocq. Symbolic assertions for our design could be expressed in a fragment of

```
Example core_regs_purged core : fancy_spred :=
  {{{ impl1.[reg_purge core] = #(enum purge_state "Purged") →
      ∀ x in (regs_to_reset core), impl1.[x] = #(zeroes (reg_type x)) }}}.
Example assume_uart_sim core : fancy_spred :=
  {{{ forall1 "arg" of (fn_arg_type ext_uart_write),
      sget_field enc_data_sig "ext_uart_write" spec0.[reg_enc_data core] = 0b~1 →
      iapp ext_uart_write "arg" = sapp ext_uart_write "arg" }}}.
```

Figure 6-4: `core_regs_purged` shows the processor's purge invariant. `assume_uart_sim` relates the behaviours of external function `ext_uart_write`: if an enclave owns the UART, then `ext_uart_write` behaves the same in both implementation and specification for all inputs.

first-order logic without existential quantifiers and with limited usage of universal quantifiers (which can trigger issues with SMT). Universal quantifiers are only used for arguments to external functions, modeled as uninterpreted functions constrained using assertions such as `assume_uart_sim`. The ∀ quantifier in `core_regs_purged` is part of a sugared syntax that is translated to a conjunction of assertions in a verified desugaring phase in Rocq before translation to SMT. This sugaring allows convenient reflection with Gallina's `forall` quantifier for concise term representation and usage in Rocq proofs.

Chapter 7

Evaluation and Discussion

We showed that our method can be applied to a synthesizable machine. We additionally discuss the following:

- What changes are needed when making design modifications to the processor? And to the memory system?
- What is the developer's proof effort and process?
- What is the process of running applications on our enclave hardware?
- Does the spec style extend to other enclave programming models and compose with other properties?

7.1 Design modifications

Adding a branch predictor. We improve the branch predictor (a `PC + 4` predictor) with a Branch Target Buffer recording target addresses of branches and jumps and a Branch History Table predicting whether branches are taken or not. This extension required no changes to the spec and Rocq proof, taking about a day and showing that non-security-critical changes can be made without triggering an avalanche of proof breakages.

Adding caches. Our framework was designed to work with caches. The basic idea is to flush caches upon enclave switching and prove cache-coherence protocols are not needed as enclaves access disjoint regions. As an example, we implement a memory subsystem with four L1 caches (two per core, for instruction and data memory) without a cache-coherence protocol (which would not be triggered, anyway). The metadata and cache lines are stored in external SRAM and are flushed with a multicycle state machine. Implementing and proving the cache took 3-4 weeks. The modular proof structure meant that the proofs of the core and SM were unchanged, and the bulk of modifications were to strengthen memory invariants (see Figure 7-1). For example:

State	Example invariant
Ready	No metadata/cache/mainMem resps
ProcessRequest	Valid meta resp $\rightarrow$ valid `(tag++index++00)`
SendFillReq	Addresses of reqs to mainMem are in config
WaitFillResp	Impl-spec reqs/resps to/from mainMem are equal
FlushLineReady idx	Metadata lines $< idx$ are invalid; no mem resps
FlushLineProcess idx	Valid meta resp $\rightarrow$ impl-spec cache data equal
FlushPrivateData	All metadata lines are invalid; no mem resps
Flushed	All μarch state is invalid; no mem resps

Figure 7-1: Examples of cache invariants proved using SMT

All valid addresses in a core's slice of the memory subsystem are in the enclave configuration. Valid addresses include addresses in requests sent from the SM, addresses of requests that the cache protocol sent to main memory, and interestingly, addresses corresponding to valid cache lines (as defined by the metadata). With cache requests, stating the invariant involved reconstructing the address from the tag and index from the metadata and outstanding request from the SM.

The implementation's and specification's caches and memory return the same data for any valid cache line or address. We prove that the implementation and specification send the same writes to caches when the metadata entry is valid. Requests to invalid cache lines can return different data between the implementation and specification as our implementation of purge invalidates metadata but does not zero the cache (we prove invalid cache lines do not affect the memory's behaviour).

On flushing caches. The verified prototype paves the way for experimenting with design optimizations. For example, our implementation has a write-back cache and invalidates one line per cycle (independently of whether the line is dirty), which increases the cost of context switching. A design with a write-through cache could invalidate all clean cache lines in a single cycle by storing an epoch counter with the metadata (such that a line is valid if the metadata is valid and the counter matches the current epoch) and incrementing the epoch counter upon context switch. This scheme is insecure: epoch bits can overflow (this bug might not show up with testing or bounded model checking but is ruled out by our spec). Isolation can be restored by resetting all metadata valid bits before the counter overflows. This change alters enclave execution time. Thus, in our framework, a new machine must be initialised with the epoch counter (the number of context switches on that core), and this design would only be secure in a threat model allowing leaking the number of context switches.

	SLOC					Time(s)	
Design	Kôika	Csim[4]	Vlog[5]	Spec	Pf	Rocq[6]	SMT[7]
Resource	[1]630	-	-	160	6600	130	-
Enc:pf-mono[2]	2410	12780	1870	280	12550	540	720,360,1
Enc:pf-mod[3]	"	"	"	"	13050	780	200,95,1
Enc+BTB	2750	14760	2080	"	"	"	340,160,1
Enc+Caches	2850	17710	2320	"	17750	1340	380,95,1

Figure 7-2: Our examples. [1]Written in δKôika. [2]Monolithic simulation proof. [3]Modular simulation proof. [4]Cuttlesim's generated `C++`. [5]Kôika's generated Verilog. [6]Inclusive of extraction time. [7]Total, max, median (per-use-of-SMT) time.

7.2 Verification cost

Figure 7-2 shows the lines of code for the implementation, specification, and proof for various examples, and the verification time in Rocq and Z3. We proved the resource-isolation example from Section 4.1 entirely in Rocq, with per-rule specifications and a postcondition semantics for δKôika, and used a hybrid Rocq-SMT approach for the other examples.

As proof writing, debugging, and iterating on invariants are interactive processes, it is preferable for interaction cycles to be short (i.e. waiting three minutes for every interaction with the SMT solver hampers the proof experience). Most of our SMT queries are checked in $<$`1` second. Modularity reduces the size of the SMT formulae and verification time, from a maximum of `360` seconds to a maximum of `95` seconds. The overhead of modularity, in this instance, is the need to state intermediate invariants and prove module boundaries align (postconditions of one imply preconditions of another).

7.3 Running an application

We implement a basic programming toolchain to compile and link enclave programs (placing them at appropriate addresses) and run a toy password-manager application using Cuttlesim to validate our design's functionality. The password-manager enclave takes requests `get_pswd(id, key)` and `add_pswd(id, pswd, key)`. A calling enclave saves stack and frame pointers on its stack, flushes secrets from registers, then requests to switch to the password-manager enclave, passing a function identifier, arguments, and a requested return config in registers. Upon context switch, the SM jumps to the bootloader address for the enclave, which processes the function-call request and returns execution to the calling enclave. One limitation is the lack of shared read-only memory, resulting in duplicating libraries across enclaves.

7.4 Extensions and future work

Alternative enclave programming models. Our examples used a cooperative enclave model, which allows a malicious enclave to run indefinitely. Instead, we can introduce a timeout and allow the SM to preempt an enclave after some number of cycles. This extension is straightforward: we could track the number of cycles elapsed in the spec, and the SM would force the enclave to exit. Another model could implement privilege levels by specifying that unprivileged enclaves switch back to a privileged enclave. This extension simply involves constraining the `can_start` transition appropriately.

Our case studies demonstrated support for configurable static allocation (we parameterise over enclave configurations with disjoint regions) and dynamic allocation of resources through the "locked mailbox" and MMIO regions (a similar technique could be used to support requests for cache lines by "creating a new machine with the requested resources"). To support security domains dynamically evolving during enclave execution (i.e. adding and removing pages), we can allow running enclaves to output requests such as `add_page(id)` to a specification API handler and have the handler return the page if allowed.

Composability. Composing strong isolation with traditional constant-time guarantees is insufficient to avoid leaking secrets: purge time can depend on microarchitectural state and thus be used as a side channel. For example, the time to flush the L1 may depend on which cache lines are dirty. While it is straightforward to specify constant-time purge in this framework (each processor must prove there exists a constant number of cycles after which it has been purged, and the SM pads the time), the performance cost may be unacceptable.

In addition, to reason about the security of enclave software, we need functional correctness of processors. By instantiating the specification with subsidiary machines that behave architecturally equivalently to the implementation machine per enclave, we prove a property stronger than strict strong isolation that can be used in future work to reason about the architectural behaviour of an enclave.

Verifying designs in other HDLs. We implemented our design in Kôika for composability with future work on hardware-software guarantees, but we expect our presented techniques for isolation verification to be applicable to other HDLs including Verilog. Indeed, the spec style is largely independent of the implementation's HDL, we expect the modular decomposition to work even better in languages with first-class support for modularity, and the single-cycle, SMT verification style should be even more effective with languages closer to RTL.

7.5 Limitations and nongoals

Our focus is on security problems emerging from sharing a computer among mutually distrusting applications, leaving integration with application security and proofs of specific software programs to future and existing related work [38, 57]. We focus on decoupling security from functional correctness: verifying functional correctness was a nongoal. Part II discusses complementary work tackling functional correctness and software-level leakage. We chose our case-study processor and enclave model to include just enough features to make the problem interesting, not to be fully representative of complications found in commodity systems. We do not handle physical attacks.

Part II

Secret-Independent Timing

Chapter 8

Granite: Context and Guiding Principles

Part I described a methodology of constructing isolated enclaves, but sidestepped the challenge of verifying functional correctness and also of ruling out leakages due to enclave runtime. This part describes Granite, a methodology that tackles both these challenges.

Granite, introduced in Section 1.4.3, is a foundational methodology for modular verification of both functional correctness and nonleakage of RTL processors against leakage-aware ISA contracts. Ever since Spectre and Meltdown, there have been numerous approaches tackling the challenge of verifying processors against leakage-aware ISA contracts. Prior to presenting details of Granite, this chapter first provides additional background and context surrounding Granite and verifying hardware-software leakage contracts: motivation for verifying hardware-software contracts (Section 8.1), the guiding principles behind design choices of Granite (Section 8.2), Granite's security goal of contractual noninterference (Section 8.3), challenges related to verifying hardware-software leakage contracts with nondeterminism (Section 8.4), and a preview of Granite's strategy for modular specification of information flow (Section 8.5).

8.1 Motivation: confidential hardware-software systems

While isolation (provided by Part I) is a useful primitive that provides broad protection between security domains with minimal requirements on software, it does not generally suffice to achieve confidential hardware-software systems: even a program that runs in isolation and that computes responses perfectly independent of secrets may leak secrets to an adversary that can observe the timing of these responses. Numerous unexpected information flows from application data to observable timing have been discovered over the years: timing variability between subroutines [49], cache occupancy [17], operand-dependent timing of complex instructions [6], compiler passes introducing the former patterns to code that did not contain them [12], speculative execution triggering unreachable code paths with these patterns [48], speculative execution allowing data flows that would be disallowed by privilege checks in executions under the ISA semantics [53], interrupt latency revealing instruction-

granular execution state of hardware-enforced enclaves [78].

This list could be continued further, but instead we note commonalities between these risks:

- While easy to explain in hindsight, the fundamental information-flow mechanisms behind these attacks were unknown to dedicated, careful implementers before the cited reports.
- Whether an information flow exists depends on the *combination* of the program and the processor microarchitecture: successfully avoiding attacks against a particular program on one processor design is not sufficient to guarantee confidentiality on another processor design, or vice versa.
- Cross-virtual-machine and across-network attacks exploit the same underlying information flows [17, 71]: **isolation and keeping untrusted code off the machine is not enough to rule out timing side channels**.
- Exploitation is unusually engineering-intensive, so published attacks may not be representative of the inventory of exploit brokers.

These factors together have painted a daunting picture: it is unclear what is needed to build a hardware-software system that can be relied upon to keep a secret, even from a network attacker, even for security-critical software, and even on simple processors. While a couple of purpose-built research systems [7, 9] have been checked safe from timing attacks as a unit, the approaches used do not yield a blueprint for modular engineering and verification of their software and hardware components. Granite addresses this challenge: we show that a particular combination of specification and proof techniques allows timing-aware confidentiality guarantees to be achieved in a modular fashion, for the first time composing timing-security proofs of digital-logic modules all the way up to the instruction-set-level specification (and then with software proofs).

8.2 Guiding principles

Formal verification against hardware-software leakage contracts is a compelling approach for ruling out timing side channels. For such a formal-verification approach to be trustworthy as the primary assurance mechanism and applicable to practical designs, we take the stance that the methodology should satisfy four requirements: 1) it is auditable, with a specification simple enough to inspect and with the RTL implementation removed from the trusted computing base (TCB); 2) it composes with software proofs for end-to-end, hardware-software guarantees that eliminate the hardware-software contract itself from the TCB; 3) it is expressive enough for real hardware, supporting the nondeterminism arising from asynchronous interrupts, inputs, and unspecified behaviour; and 4) it is modular, both to tame the state-explosion

problem encountered in hardware verification and to let designers reason about early-stage designs. No prior methodology for verifying processors against hardware-software leakage contracts achieved all four. Granite addresses this gap.

Existing approaches. Various works propose sophisticated leakage models [20, 38, 57] and defenses [4, 24, 29] but are either unverified or do not provide formal guarantees for RTL implementations. A line of work [31, 74, 80] uses model checkers to verify noninterference of RTL designs against ISA leakage models. However, these methods assume functional correctness (and therefore require auditing the implementation), use a limited specification language of model checkers that does not smoothly compose with software proofs for end-to-end hardware-software guarantees, and lack the modular decomposition needed to tackle the state-explosion problem and scale to larger designs. Another line of work [15, 23, 32] uses proof assistants to prove functional correctness of processors as refinement against one-instruction-at-a-time (OIAAT) specifications, but refinement does not generally suffice for confidentiality as nondeterminism leaks secrets [63]. Lastly, Notary [7] and Parfait [8] collapse the hardware-software stack to prove functional correctness and confidentiality for specific programs on specific processors, but the approaches used do not yield a blueprint for modular engineering and verification of their software and hardware components.

8.3 Security guarantee: contractual noninterference

Granite's formalisation of nonleakage is based on *speculative noninterference* [20, 24, 74, 86, 88], which (informally) captures the notion that if a program does not leak information under the ISA model (e.g. it satisfies the constant-time contract), then running on hardware does not leak information. As leakage can arise without speculation and contracts generalize beyond it, we adopt the more general term contractual noninterference. We first state the property in the deterministic setting (consistent with definitions from prior work on speculative noninterference), where the architectural specification and implementation can both be modeled as deterministic state machines; Section 8.4.1 discusses challenges with generalizing to nondeterminism; and Section 9.3 presents our approach to generalize to nondeterministic specs.

Definition 1 (Contractual noninterference without nondeterminism)**.** Let Pub represent public initial state (such as program instructions and public data), and Sec^1 and Sec^2 represent secret data. Let $O^n_{ISA}(Pub, Sec)$ denote the specification leakage trace generated by running the ISA machine for n steps and $O^m_{\mu}(Pub, Sec)$ denote the trace of cycle-accurate observations generated by running the RTL implementation for m clock cycles. The implementation

satisfies contractual noninterference without nondeterminism if $\forall Pub$:

$$\left(\forall Sec^1, Sec^2, n.\, O^n_{ISA}(Pub, Sec^1) = O^n_{ISA}(Pub, Sec^2)\right) \rightarrow$$
$$\left(\forall Sec^1, Sec^2, m.\, O^m_{\mu}(Pub, Sec^1) = O^m_{\mu}(Pub, Sec^2)\right)$$

Leakage transformers. The above definition is a 4-copy property, stating that if the ISA leakage trace is independent of secrets (equivalently, the ISA leakage trace consists of public information), then the microarchitectural leakage trace is also independent of secrets (and therefore, there exists a function such that the microarchitectural trace is a function of public information). The latter function is exactly the "leakage transformer" we use to express nonleakage:

$$(\exists g.\, \forall n, Sec.\, O^n_{ISA}(Pub, Sec) = g(n, Pub)) \rightarrow$$
$$\left(\exists f.\, \forall m, Sec.\, O^m_{\mu}(Pub, Sec) = f(m, Pub)\right).$$

Challenges with proving contractual noninterference. Exploiting the assumption that the ISA leakage trace is public requires relating microarchitectural state to instruction-set-level state (which instructions have retired and *will* retire), a challenging task akin to proving functional correctness. An important simplification in model-checking-based work [31, 74, 80] assumes functional correctness of the processor, using the retire stage of the processor in place of the state of the ISA execution. While appealing due to avoiding the need to model and reason about the instruction-set specification, this approach runs into state-space-explosion challenges (due to needing to show that an instruction will retire). Furthermore, we are not aware of any work that establishes an integrated correctness-and-confidentiality theorem based on separate functional verification and constant-time-assuming-functional verification of a processor. Doing so seems challenging given leading approaches for comprehensive functional verification allow a rather flexible "flushing relation" [15, 72] between the retire-stage state and instruction-set-level state instead of establishing strict equality. As a goal of this work is a combined hardware-software result in which the hardware-software contract is only an intermediate specification, we seek a different approach.

8.4 Refinement and nondeterminism

Our approach is to extend—to the nonleakage setting—an approach to processor-functional-correctness verification [15, 23] based on using proof assistants to establish correctness via refinement: i.e., every behaviour of the implementation is a possible behaviour of the specification.

8.4.1 Specification challenges arising from inputs and nondeterminism

However, as discussed in Section 2.2.2, nondeterminism gives rise to challenges with straightforward application of refinement techniques to Definition 1:

Challenge 1: Nondeterminism leaks secrets: the need for secret-independent nondeterminism. Classical refinement becomes unsound for nonleakage: when a specification nondeterministically permits several behaviours, an implementation may resolve the choice on secrets—e.g., taking an interrupt immediately if and only if a secret bit is set—and the leak is then "explained away" by an adversarial choice of specification nondeterminism. To preserve microarchitectural flexibility while preventing side channels, specification frameworks must support secret-independent nondeterminism to ensure that while an implementation retains the freedom to resolve design choices (such as at which instruction boundary to take an interrupt), the choice is independent of secrets.

Challenge 2: Nondeterminism from inputs and interrupts affects the specification leakage trace. If execution time were the sole source of nondeterminism and the software leakage trace were independent of execution time (as is standard in idealized constant-time models [25]), then Definition 1 would be sufficient, as the specification leakage trace would be invariant under implementation design choices[1]. Real-world hardware, however, introduces dynamic sources of nondeterminism that directly influence the specification's leakage trace. For example, the values returned by MMIO requests can be affected by previous interactions with the external world, which has downstream effects on leakage. Additionally, asynchronous interrupts are scheduled nondeterministically: while an ISA may mandate that precise interrupts occur at instruction boundaries [81], it often permits flexibility regarding which specific instruction boundary traps the event. This hardware-driven scheduling decision actively diverges program control flow to an interrupt handler, altering the resulting sequence of specification leakage events.

Challenge 3: Inputs and different step sizes. The mismatch in step sizes between instruction-level software specifications and cycle-level hardware implementations not only is a key cause of the underspecification leading to microarchitectural timing side channels, but it also complicates interactions with the external world[2]. The external world can be modeled as a function of the cycle-level outputs of the microarchitectural machine, but high-level ISA specifications typically lack structural notions of clock cycles. Thus, Granite lowers the ISA machine into a cycle-level state-transition system so both layers interact with a synchronized model of the external world, enabling a reduction of functional correctness and nonleakage to trace equivalence provable by standard single-cycle induction.

[1] This definition is used in prior work [25], which does not address input nondeterminism.

[2] This step-size mismatch is also a key challenge in verifying functional correctness, discussed in Section 10.5.

8.5 Specifying information flow in combinational methods

We augment the specifications used for modular functional-correctness verification of the processor's submodules with timing specifications. The intuition is that any functional-correctness specifications suitable for modular verification of the processor must already capture some relation between the inputs and outputs of each module that is sufficiently precise to eventually relate these values to instruction-set-level state. Augmenting these specifications with constraints on what the timing of the output may depend on *in terms of the module's inputs* allows confidentiality verification to benefit from the modular structure of functional-correctness proofs. While functional-correctness proofs are nontrivial, the methodology for crafting them is established and reasonably consistent across different hardware-description languages [15, 23**?**].

As noted (Section 8.4.1), the nondeterminism and refinement that functional-correctness frameworks embrace do not suffice for confidentiality. Our approach, therefore, is to show equivalence (degenerately, refinement of a deterministic specification) after resolving all nondeterministic choices preemptively by instantiating the existential parameters.

8.5.1 Quartz: A deterministic HDL enabling one-method-at-a-time semantics

This strategy in turn leads us to choose a deterministic language, Quartz, for expressing our hardware designs. We work with the intersection of Rocq's native language Gallina and SystemVerilog functions operating on pairs, sums, records, and arithmetic modulo powers of 2. All concurrency is implicit—independent subexpressions in a functional program can be evaluated in parallel, and hardware-synthesis tools naturally produce implementations that do so. Strictly speaking, all operations syntactically representable in our fragment are combinational. As RTL model-checking tools can trivially prove equivalence of encodings of the same circuit as different RTL programs, and our fragment can express any circuit, this encoding choice is not a capability limitation.

Following the tradition of Bluespec and hardware-verification frameworks it inspired [15, 16, 23], we specify modules in terms of atomic *methods* that deterministically produce updated state and output given a starting state and input, perhaps calling other methods in the process. Each clock cycle involves several methods firing, in our case through deterministic, purely functional calls from the top-level `tick` method. A module implementation can then be verified independently as trace equivalence against a deterministic spec, existentially instantiated.

Chapter 9

Overview of Granite

Granite establishes an end-to-end guarantee—a constant-time program running on synthesizable RTL leaks nothing through timing—by composing a chain of proofs, shown in Figure 9-1. This section highlights aspects of the proof chain and introduces an example, a zero-skip multiplier, that exposes the specification challenge of this thesis: a specification must admit a whole family of functionally correct designs with legitimately varying timing yet forbid any timing that depends on secrets.

9.1 Example: zero-skip multiplier

Before the full ISA contract, we introduce the specification style on two toy examples. We provide intuition for how a family of deterministic specifications can admit a space of functionally correct designs with secure timing variations.

Warm-up: a combinational circuit. The smallest interesting case has no state. A combinational circuit σ maps a public input and a secret input to a public output and a secret output, $(out_{pub}, out_{sec}) = \sigma(in_{pub}, in_{sec})$. Confidentiality requires only that the public output not depend on the secret input: that there exist a function f with $out_{pub} = f(in_{pub})$, while out_{sec} is free to be any function of everything. We write this requirement as

$$\exists f, g.\ \forall in_{pub}, in_{sec}.\ \sigma(in_{pub}, in_{sec}) = \big(f(in_{pub}),\ g(in_{pub}, in_{sec})\big).$$

The exact value of out_{pub} is left unspecified, but the existential f forces it to be secret-*independent*: this device is our basic one for capturing secret-independent nondeterminism without overconstraining σ. The unconstrained g captures behaviour that is permitted to depend on secrets.

Adding state and time: the zero-skip multiplier. The combinational case fixed *which* outputs may depend on secrets. State and clock cycles raise a second question: *when* does an output appear, and what may that timing depend on? We answer both with the same strategy

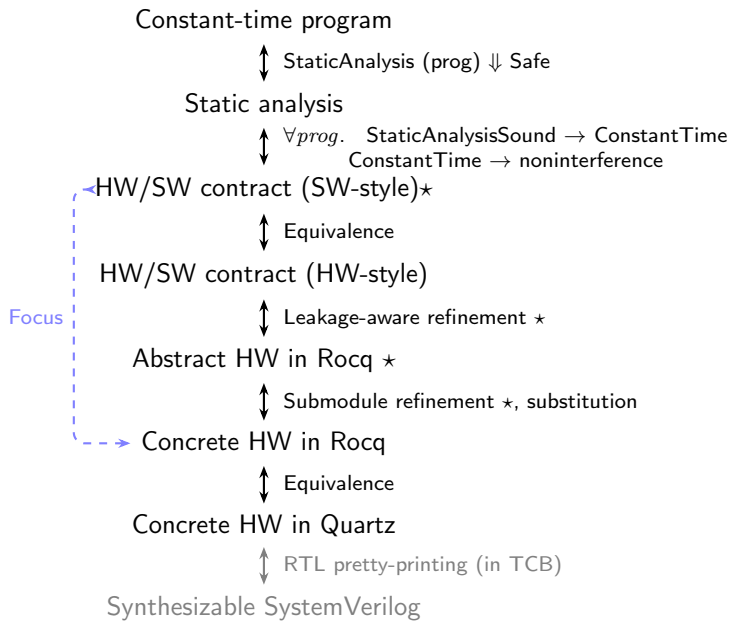


Figure 9-1: Layered, end-to-end proof structure. A program is deemed constant-time by a static analysis proven sound against a software-style ISA contract. Observations are the timing of externally observable behaviours (e.g. MMIO output). An equivalence proof lowers the software-style ISA contract to a hardware-style ISA contract that uses hardware-optimized, combinational decode and execute logic shared across instructions. A processor implementation with abstract submodules, specified using leakage contracts, is proven to refine the hardware-style ISA contract. After proving concrete submodules individually satisfy their specifications, the resulting processor is proven equivalent to an implementation in Quartz and pretty-printed to SystemVerilog.

as before but applied over traces rather than single values: every public output, now including the cycle when it becomes available, must be a fixed function of the module's public inputs trace.

Consider a zero-skip multiplier that computes $a * b$, returning in one cycle when either operand is zero and in n cycles otherwise. We intend to place it inside a processor whose multiply instruction declassifies only *whether* an operand is zero; the multiplier's contract must therefore permit the zero-skip optimization while forbidding any other data-dependent timing. We model implementation and specification as Mealy machines[1] over the method calls `Enq`, `Deq`, `Tick`, `Full`, `RespReady`, and `Peek`, with correctness defined as output-trace equivalence, summarized in Figure 9-2.

A specification should admit a whole family of secure implementations, so it leaves several behaviours nondeterministic—of the two kinds from the warm-up. A secret-independent

[1] A Mealy machine is a *deterministic* finite-state machine whose output values are determined both by its current state and the current input.

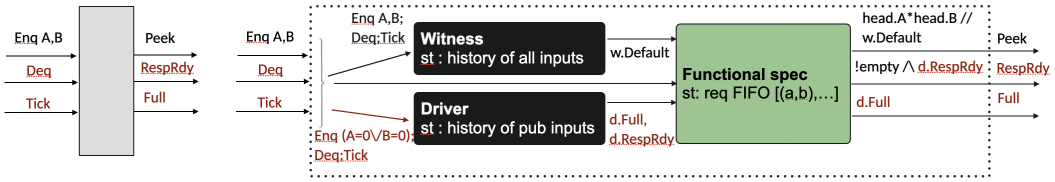


Figure 9-2: Implementation (left) and specification (right) of a pipelined, zero-skip multiplier. Functional correctness is specified based on a canonical representation of state as a FIFO of requests. Leakage-aware nondeterminism is specified using witness and driver components, with canonical representation of state as the history of their inputs. An implementation is correct and secure if there exist a driver (a function of the history of public inputs; deciding latency) and witness (a function of all inputs; deciding the value of `Peek`-when-not-ready) such that it is equivalent to the specification instantiated with the parameters.

driver (here `d.RespReady`/`d.Full`, a function of only the public leakage trace) fixes the timing, seeing the zero bit but nothing else, playing the role of f. An unconstrained *witness* (here the `Peek`-while-invalid value, which may reflect secrets) plays the role of g. These parameters are untrusted: an implementation is functionally correct and secure exactly when *there exist* driver and witness instantiations under which its trace equals the specification's. Section 11.2.1 gives the full contract in Rocq and proof method—we show that the witness is trivially instantiated with the implementation, and the driver is instantiated with a "shadow copy" of the implementation with inputs reconstructed from public inputs.

9.2 Security goal and threat model

We consider HW/SW contracts based on three components: a software or ISA specification with leakage semantics, a hardware or microarchitectural implementation, and an adversary model specifying what aspects of the microarchitecture are observable.

Adversary capability. Granite considers adversaries with direct access to the microarchitecture's digital I/O boundary, capable of driving input signals and observing output signals at every clock cycle. We formalise this capability by defining a public observation function O_μ over execution traces of the microarchitectural state machine. For example, our case study (Chapter 12) defines observations as the cycle-by-cycle ready/valid handshake wires of the external MMIO bus. Fixing observations at this system boundary means internal microarchitectural transitions need not be audited directly.[2] The threat model includes digital timing side channels but excludes arbitrary side channels such as power or EM radiation.

[2] Our intermediate theorems also prove secret independence of other commonly used observations, such as instruction completion time or the timing and addresses of all memory requests.

Security goal. Granite proves the implementation *satisfies its leakage contract*—the contractual-noninterference property of Definition 1, so observations are a function of what the contract declassifies or designates as public—and additionally proves *functional correctness* (refinement of the ISA's I/O behaviour, universally over all programs, data, and inputs). Section 9.3 generalizes contractual noninterference to nondeterminism and I/O over infinite traces.

ISA specification with leakage semantics. Granite's methodology is parameterised over an ISA specification defined as a state machine, consisting of architectural states and a transition function, together with a *leakage function* that emits the information the contract declassifies at each step. We focus on ISAs with sequential, one-instruction-at-a-time semantics on a single core.[3] As a concrete example, our case study considers a representative subset of RISC-V under the constant-time policy. Architectural states are $\sigma = \langle rf, pc, imem, dmem \rangle$—a register file rf, program counter pc, and separate instruction and data memories $imem$ and $dmem$ mapping addresses to bytes (we restrict to non-self-modifying programs)—with $step : \sigma \to \sigma$ giving the standard "one-instruction-at-a-time" fetch-decode-execute transition. A leakage function $leak$ computes the leakage of an instruction, excerpted below, for a constant-time contract for RISC-V with zero-skip multiplication. We additionally treat the byte-encoded instructions themselves as leaked, along with writes to interrupt-related CSRs (Chapter 12 explains why).

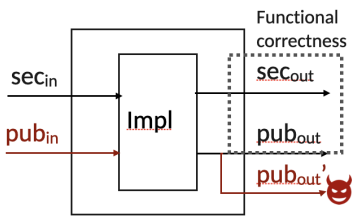
```
| Add rd rs1 rs2 ⇒ LeakAdd (* All inputs are secret. *)
| Beq rs1 rs2 offset ⇒ LeakBeq (rf[rs1] = rf[rs2]) (* The branch condition is leaked. *)
| Lw rd rs1 offset ⇒ LeakLw (rf[rs1]) (* The load address is leaked. *)
| Sw rs1 rs2 offset ⇒ LeakSw (rf[rs1]) (* The store address is leaked (but the store data is not). *)
| Mul rd rs1 rs2 ⇒ LeakMul (rf[rs1] = 0 ∨ rf[rs2] = 0)  (* Whether either argument is zero is leaked. *)
| Csrrw rd rs1 csr ⇒ match csr with
                     | mtvec ⇒ LeakCsrrwMtvec rf[rs1]
                     | mie ⇒ LeakCsrrwMie rf[rs1]
                     | _ ⇒ LeakCsrrw
                     end
```

9.3 Architectural specifications under nondeterminism

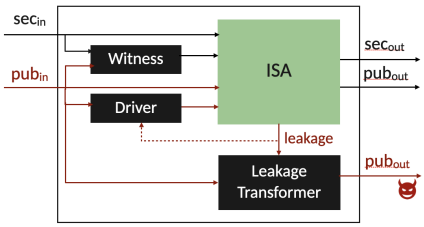
Recall the three challenges of Section 8.4.1: secure implementation choices affect the leakage trace, nondeterminism can leak secrets, and step sizes mismatch. We resolve all three with a single construction, applying the secret-independent driver and secret-dependent witness that determinized the multiplier of Section 9.1 to the ISA machine.

First, we lower the one-instruction-at-a-time ISA machine into a cycle-accurate state machine, aligning specification steps with implementation cycles and letting both interact with a synchronized model of the external world. Second, we *determinize* the remaining

[3] See Section 14.3 for discussion of generalizing beyond sequential ISAs and to multicore.



(a) Implementation state machine, cycle-level, with public and secret I/O at the wire level. The public output wires, observed by the adversary, are duplicated for convenience to assert both functional correctness and nonleakage as trace equivalence.



(b) Specification state machine, cycle-level, constructed by parameterising a trusted ISA machine generating declassified leakages with a secret-independent driver, a secret-independent leakage transformer, and an unconstrained witness. The black boxes are untrusted.

Figure 9-3: An implementation satisfies a leakage-aware ISA contract—and hence is both functionally correct and secure—if there exist a witness, driver, and leakage transformer under which the implementation and specification traces are equivalent. Red wires carry public, adversary-observable information. Only public information is fed into the leakage transformer, which produces the adversary-observable output.

nondeterminism with explicit existential parameters [52], of the same two kinds as in the multiplier:

- a secret-independent *driver*, fed only public data, that resolves secret-independent choices (i.e. when the machine steps and triggers I/O, or when it takes a pending interrupt); and
- an unconstrained *witness*, which may depend on secrets, modeling unspecified behaviour such as the value driven on a data bus while its `valid` bit is low.

A separate secret-independent *leakage transformer* maps the specification's declassified leakage trace to the adversary's cycle-level observations; its existence *is* the security property. An implementation is then functionally correct and secure when there *exist* a driver, witness, and leakage transformer under which it is trace-equivalent to the resulting deterministic, cycle-accurate specification (Figure 9-3). All three parameters are untrusted, so only the cycle-accurate ISA machine—not the implementation nor the parameters—must be audited. Chapter 10 develops the construction in full, and Chapter 13 shows that trace equivalence to it implies the classical noninterference property of Definition 1.

9.4 Two axes of modularity

Granite decomposes the proof along two axes, which together tame the state-space explosion of monolithic hardware verification and let the framework reason about early-stage designs.

Vertical Modularity. Figure 9-1 shows Granite's layered refinement chain: a constant-time program, a *software-style* ISA contract amenable to software proofs (Chapter 12), a *hardware-style* contract sharing combinational decode/execute/writeback logic (reused across spec and implementation), abstract-submodule and HDL-independent layers (useful for reasoning about early-stage designs and defenses), and finally Quartz with a trusted SystemVerilog pretty-printer. Crucially, refinement preserves not just functional state but leakage: we chain leakage-transformer functions so the implementation's cycle-level leakage trace is provably a function of the top-level specification's.

Horizontal modularity (Chapter 11). Horizontally, the abstract-hardware design decomposes into submodules—FIFOs, multipliers, branch predictors, memory—that interact only through method calls and are each specified (correctness and nonleakage) and verified independently (as previewed in Section 9.1). This tames state-space explosion and lets a designer swap algorithms or reason about an early-stage defense without redoing the top-level proof; Chapter 11 develops the details.

Chapter 10

Specifying HW/SW Leakage Contracts

This section shows how Granite turns a traditional ISA contract into a family of deterministic, cycle-accurate state machines suitable for leakage-aware refinement proofs. We build up to, and present details of, the specification in Figure 9-3 supporting I/O and interrupts. Starting with an example baseline ISA state machine, we lower it to a cycle-accurate machine that reconciles the step-size mismatch with implementations (Section 10.1), describe its existential parameters (Section 10.2), and discharge both functional correctness and nonleakage with a single trace-equivalence obligation (Section 10.3). We then discuss the bug classes ruled out and outline the proof strategy.

10.1 Lowering an ISA contract to a cycle-accurate machine

State machines. We model state machines as Mealy machines. A *state machine* $M : Machine\ I\ O$ is a triple $(State, s_0, \delta)$ of a state type, an initial state $s_0 \in State$, and a transition function $\delta : State \times I \to O \times State$. Driving M with an input sequence $\vec{i_n} = i_1 \cdots i_n$ yields an n-element output trace, written $Tr^n_M(\vec{i})$. Both the specifications and implementations are Mealy machines; this section works toward equating their traces.

Baseline ISA machine. The baseline ISA is a Mealy machine over architectural states $\sigma = \langle rf, pc, imem, dmem \rangle$—a register file, program counter, and separate instruction and data memories (Section 9.2). Instructions are encoded as an inductive type $Instr$ with a decode function $decode$ mapping machine words to $Instr$ and an execute function $exec : \sigma \times Instr \to \sigma \times LeakEvent$ that updates the architectural state and emits the per-instruction leakage event prescribed by the contract. The baseline one-instruction-at-a-time machine, which takes no inputs, has the following state-transition function:

$$step : \sigma \times unit \to \sigma \times LeakEvent \triangleq \lambda s.\ exec(s, decode(imem[s.pc])).$$

Supporting I/O with a wire-level, cycle-level machine. We want to support I/O (we use MMIO as a running example). As such, we aim to relate specifications and implementation *cycle-by-cycle*, so that both react to the same external inputs at the same instant. However, $step$ executes a whole instruction atomically, whereas an implementation may spread one instruction across many cycles and retire nothing at all on a stalled cycle. In addition, an MMIO load request can block until a response is received, breaking the one-instruction-at-a-time semantics. Furthermore, we need to add a trusted wire-level interface to support MMIO: to demonstrate, we extend σ with a handshaking ready/valid/data interface, considering ready/valid signals to be public and data signals to be secret.

Cycle-accurate machine with I/O. We lower the ISA into a cycle-accurate machine of type

$$Cycle{-}ISA : Machine\,(PubIn \times SecIn \times DriverOut \times WitnessOut)\,(PubOut \times SecOut \times LeakEvent)$$

whose transitions are split so that each carries at most one distinct leakage or observable effect, letting a single implementation cycle map to zero or more specification steps. As shown in Figure 9-3, the machine is existentially parameterised with a driver and witness:

$$Driver : Machine\,(PubIn \times LeakEvent)\,DriverOut$$
$$Witness : Machine\,(PubIn \times SecIn)\,WitnessOut$$

responsible for resolving secret-independent choices (in this case, whether to take a step and cause external I/O observations from sending/receiving MMIO requests) and plausibly secret-dependent choices (e.g. the value of the data wire when the valid signal is low) respectively. At heart this machine is still the trusted OIAAT ISA, now emitting its per-instruction leakage stepwise (under the constant-time policy: branch conditions, load/store addresses, and inputs to variable-latency instructions); it is the only component that must be audited; the driver and witness are untrusted.

Here, $Cycle{-}ISA$ is decomposed into three stages (extended in Figure 10-1 with interrupts) in which MMIO is the only observable effect:

1. `StepLeak`: when driven, emits the leakage of the next instruction (at the program counter) without executing it.

2. `StepInstr`: when driven, runs one fetch–decode–execute to completion—except when the instruction issues an MMIO request, in which case it emits the observable request (by appropriately setting ready/valid/data wires) and waits.

3. `StepWaitMMIOResp`: when driven, consumes an MMIO response if one is available (else blocks). *Which cycle* the response arrives is observable (altering the `ready` signal) and may differ from when the external world sent it.

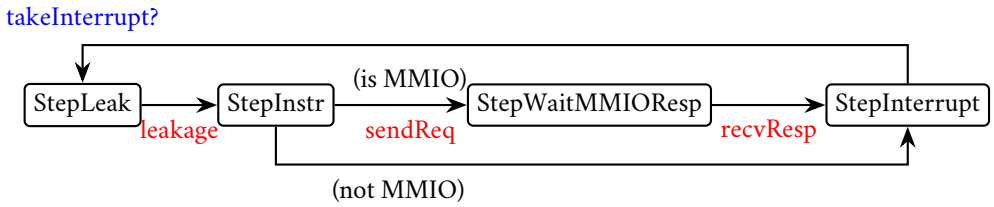


Figure 10-1: The $Cycle{-}ISA$ state machine. The driver directs the machine to take steps and decides when an interrupt is taken. The machine outputs declassified leakage and drives wires on the MMIO interface.

The wire-level interface and functional correctness. A trusted layer bridges the gap between the specification's abstract "send an MMIO request" and the wire-level "ready/valid" I/O interface: the outgoing `valid` is held high while a request is pending, and the response is consumed when the input `ready` is also high; similarly, the outgoing `ready` is held high exactly when the machine is in `StepWaitMMIOResp`. This interface specifies functional correctness: the ISA spec enforces that outgoing requests correspond to MMIO requests in the ISA semantics.

Supporting interrupts. In the above, an implementation has freedom to vary timing (in a secret-independent manner) on the MMIO interface and to set the MMIO data wire when the valid signal is low; we additionally want to support interrupts. Typically, interrupts must be precise, corresponding to a sequential execution of the program that can take an interrupt after an instruction, but the implementation can choose after which instruction to take an interrupt. This choice should be independent of secrets but is a true nondeterministic choice that depends on I/O and alters the instructions being executed.

We extend our three-state $Cycle{-}ISA$ machine to support interrupts with a fourth `StepInterrupt` state, summarized in Figure 10-1. $DriverOut$ is extended to provide (to `StepInterrupt`) a Boolean argument for whether to trap: if the Boolean is true, an interrupt is pending, and interrupts are enabled, then the machine traps to the handler; otherwise, no interrupt is taken. This staging lets the implementation choose at which instruction boundary to trap an interrupt.

Generality. Section 14.3 discusses how this transformation generalizes to settings with multiple "next states," such as verifying a processor independently of memory, where instruction- and data-memory requests are part of the external interface. To allow speculative loads with secret-independent timing and addresses, the machine can be extended with, e.g., a `SendLoadReq addr` directive issuable by the secret-independent driver at any step.

10.2 Determinizing nondeterminism via existential parameters

Recall from Section 9.3 that the cycle-level specification is parameterised by three untrusted black boxes; we give their types and their roles in the context of an ISA specification.

The driver. $Driver : Machine\ (PubIn \times LeakEvent)\ DriverOut$, fed only public data, emits a sequence of commands to the ISA machine. In the baseline of Figure 10-1 a command is a list of Booleans dictating how many steps to take and whether to trap (ignored unless at `StepInterrupt`); when a state admits multiple transitions (e.g. exposing memory requests, cf. Section 14.3) it selects the transition, such as whether to issue a speculative load.

The witness. $Witness : Machine\ (PubIn \times SecIn)\ WitnessOut$ models unspecified, possibly secret-dependent behaviour—in our case study, the MMIO output wire on cycles when `valid` is low, which the consumer must ignore. Chapter 11 shows composition with a parent module removes witness-induced leakage from the TCB.

The leakage transformer. $LT : Machine\ (PubIn \times LeakEvent)\ PubOut$ maps cycle-level leakage to adversary observations using public data only. Its existence is the security property: observations—including timing signals such as ready/valid—reveal nothing beyond what the contract declassifies.

10.3 Functional correctness and nonleakage via trace equivalence

The above construction allows a single trace-equivalence theorem to discharge both properties at once (and a trivial transitivity property, useful for modular proofs). Composing the pieces as in Figure 9-3, the specification machine

$$Spec_{\{LT, Driver, Witness\}} : Machine\ (PubIn \times SecIn)\ (PubOut \times PubOut \times SecOut)$$

exposes two public-output channels: one driven by the leakage transformer (observations) and one by the ISA's functional-correctness definition. Lifting the implementation $\mu : Machine\ (PubIn \times SecIn)\ (PubOut \times SecOut)$ to the same interface by duplicating its public-output wires lets both properties be discharged by a standard trace-equivalence theorem (this duplication is not essential; the equality can be asserted separately).

Definition 2 (Correctness and nonleakage under a leakage-aware contract)**.** An implementation is functionally correct and nonleaking under a leakage-aware ISA if there exist leakage-transformer, driver, and witness machines such that the implementation and specification traces are equivalent (for all cycle counts n and public and secret inputs, $\vec{Pub}_n$ and $\vec{Sec}_n$, of

length n):

$$\exists LT, Driver, Witness.\ \forall n, \vec{Pub}_n, \vec{Sec}_n.$$
$$O^n_\mu(\vec{Pub}_n, \vec{Sec}_n) = LT^n(\vec{Pub}_n, O^n_{Cycle-ISA\{Driver,Witness\}}(\vec{Pub}_n, \vec{Sec}_n)) \wedge$$
$$Func^n_\mu(\vec{Pub}_n, \vec{Sec}_n) = Func^n_{Cycle-ISA\{Driver,Witness\}}(\vec{Pub}_n, \vec{Sec}_n),$$

where O_μ denotes the adversary observation trace (of type $list\ PubOutput$), $Func$ the functional (I/O) trace (of type $list\ (PubOutput \times SecOutput)$), and $O_{Cycle-ISA}$ the specification leakage trace. Equivalently, as a single-trace equation:

$$\exists LT, Driver, Witness.\ \forall n, \vec{Pub}_n, \vec{Sec}_n.$$
$$Tr^n_\mu(\vec{Pub}_n, \vec{Sec}_n) = Tr^n_{Spec_{\{LT,Driver,Witness\}}}(\vec{Pub}_n, \vec{Sec}_n)$$

10.4 Classes of bugs ruled out by the top-level specification

Proving trace equivalence between the top-level specification and synthesizable RTL rules out entire classes of microarchitectural and security bugs without enumerating individual attacks, yielding robustness to attacks not yet discovered.

Functional-correctness bugs ruled out. Equivalence to a sequential, OIAAT ISA specification eliminates standard and edge-case correctness bugs: incorrect instruction-decoding and execution-logic errors, data hazards and pipeline races, and imprecise traps (for precise exceptions and asynchronous interrupts, bugs that trigger mid-instruction or corrupt saved contexts; the `StepInterrupt` stage guarantees interrupts land only at instruction boundaries).

10.4.1 Timing-side-channel and security bugs ruled out

Timing leakage is an emergent property of subtle microarchitectural interactions that classical functional testing ignores. Granite's noninterference proof rules out vulnerabilities—without naming speculated instructions or microarchitectural structures—such as:

- Transient/speculative-execution attacks (e.g. Spectre, Meltdown): side effects of transiently executed instructions (on misspeculated, unreachable paths) cannot leak secrets. Because the specification declassifies leakage only for instructions on correct paths, speculative state changes remain secret-independent; a secret-dependent speculative load or branch with observable timing effects would make it impossible to construct a secret-independent shadow machine with matching timing for all public data.
- Speculative interference and port-contention leaks: younger, misspeculated instructions cannot alter timing of older, bound-to-retire instructions via contention on ALUs,

MSHRs, or functional units, since such instructions do not appear in the specification leakage trace.

- Secret-dependent interrupt timing leaks: classical refinement permits handling an interrupt immediately when a secret is zero but delaying it otherwise; Granite rules out this behaviour.

10.5 Proof strategy and instantiating existential parameters

Definition 2 yields a formulaic proof strategy: construct a deterministic specification machine and cycle-level bisimulation relation, reducing the proof obligation to single-cycle preservation of a simulation relation between three machines—the $Cycle{-}ISA$ machine, the implementation, and the "shadow machine" (a copy of the implementation that runs only on public data). We summarize the elements below (see Chapter 12 for detail).

1) Instantiating the existential parameters. The witness is simply a copy of the implementation machine. The leakage transformer and driver can be instantiated with shadow copies of the implementation containing only public data. As the shadow copies execute, they replace data dependent on ISA-declassified information with information from leakage events (e.g. the address of a load/store request is replaced with the address from the leakage trace). The shadow machines differ from the implementation on, e.g., register-file contents and parts of microarchitectural state, but they will likely agree on aspects of microarchitectural state influencing control flow (such as branch-predictor state). The specification does not prescribe what the shadow machine consists of, and there are multiple valid instantiations (for example, leakage information can be threaded through at any point before it affects observable timing). The driver essentially precomputes whether, in the upcoming cycle, the next instruction reaches its visibility point (allowing a leakage event to be generated), an observable event occurs (e.g., sending/receiving an MMIO request/response), or it is resolved whether an interrupt is taken after the previously committed instruction.

2) Relating the implementation state to the ISA state machine. This element is standard in functional-correctness proofs in the style of Fjfj and Kami. We state a simulation relation between specification and implementation state as a flushing invariant [72]: an instruction bound-to-commit in the implementation relates to the specification's state after that instruction has committed. Nondeterminism not yet resolved, such as whether an interrupt is taken, is delayed in the specification until the implementation resolves it. In other words, the driver cannot drive choices with observable side effects (MMIO loads/stores, taking interrupts) until the implementation has resolved the choices (otherwise, one fails to construct a valid proof).

3) Relating the implementation state to the shadow machine. Given the shadow machine above, the implementation and shadow machine share the same microarchitectural state. The relation amounts to designating which signals are public (typically signals related to control logic or that affect timing) and equating them—e.g., branch-predictor state, state derived from fetched instructions, and the valid bits of pipeline FIFOs. Chapter 11 showcases how an abstract, method-call-history representation of submodules supports a lightweight strategy of asserting that public state is equivalent across submodules based on asserting equivalence of public traces, independent of their concrete implementation.

Chapter 11

Horizontal Modularity via Modular Abstractions

Granite is also *horizontally* modular: a design at the abstract hardware layer is decomposed into submodules—FIFOs, multipliers, branch predictors, the memory subsystem—that interact only through method calls. Each submodule is specified by its own contract capturing both functional correctness and leakage, independent of the rest of the design.

One method at a time. The key enabler is a *one-method-at-a-time* abstraction, the method-call analogue of the one-rule-at-a-time discipline of rule-based HDLs such as Bluespec [59]. Rather than reasoning about concurrently updating wires, Granite restricts intercomponent communication to method calls with a deterministic, sequential semantics: each call is evaluated atomically, and cycle-accurate timing is recovered by a `tick` method whose top-level invocation constitutes one clock cycle. This structure lets each submodule be verified in isolation by simulation against its specification and then substituted into the larger design, so the whole-processor proof reasons over canonical representations and abstracts over implementation details of subcomponents. This intercomponent modularity is formalised in Rocq using freer monads to represent component interactions, with the *one-method-at-a-time* sequential abstraction.

Substitution principle. A *substitution principle*, proven sound based on the one-method-at-a-time reasoning, allows substitution in proofs.

Definition 3 (Leakage refinement)**.** Let $SubImpl$ be a Mealy machine and $SubSpec[\cdot]$ be a parameterised family of Mealy machines. We say $SubImpl$ *leakage-refines* its local specification $SubSpec$ if there exist parameters $SubParams$ such that $SubImpl$ is equivalent to $SubSpec[SubParams]$ for all inputs, denoted as $SubImpl \sqsubseteq_{\exists} SubSpec[\cdot] \triangleq \exists SubParams.\, SubImpl \equiv SubSpec[SubParams]$.

Theorem 11.0.1 (Substitution principle)**.** Let $SubImpl \sqsubseteq_{\exists} SubSpec[\cdot]$ as above. If a top-level implementation $Impl(\cdot)$ verified with submodule $SubSpec[\cdot]$ is equivalent to $Spec$ for all

choices of $SubParams$, then the $Impl$ instantiated with the concrete submodule $SubImpl$ is equivalent to the specification:

$$SubImpl \sqsubseteq_{\exists} SubSpec[\cdot] \wedge (\forall SubParams.\, Impl(SubSpec[SubParams]) \equiv Spec) \\ \rightarrow Impl(SubImpl) \equiv Spec$$

Composing with parameterisation of $Spec$ follows readily.

11.1 Hardware semantics

To formalise modular hardware components within Rocq, Granite uses a lightweight program embedding structured as a freer monad, summarized in Figure 11-1.

```
Inductive prog {Method: Type → Type} {Ret: Type} :=
| Return : Ret → prog
| Call {A} : Method A → (A → prog) → prog.
Record Spec {Method: Type → Type} :=
{ State : Type;
  EvalMethod {A} : Method A → State → A * State;
  initialState: State }
Record Module {Method: Type → Type} :=
{ ModuleMethod: Type → Type;
  ModuleBase : Spec ModuleMethod;
  ModuleProgram: forall Ret, Method Ret →
                prog ModuleMethod Ret }
Fixpoint evalProg
  {Method} (spec: Spec Method) {R}
  (p: prog Method R) (st: spec.(State))
  : R * spec.(State) :=
  match p with
   | Return e ⇒ (e, st)
   | Call m f ⇒ let '(r, st') := spec.(EvalMethod) m st in
                evalProg (f r) st'
  end.
Definition refines Rel :=
  forall s1 s2, Rel s1 s2 →
  (forall method r s1', spec1.EvalMethod method s1 = (r, s1') →
          exists s2', spec2.EvalMethod method s2 = (r, s2') ∧
                      Rel s1' s2').
Definition simulates (spec1 spec2: Spec Method) :=
  exists Rel, Rel spec1.initialState spec2.initialState ∧
    refines Rel.
```

Figure 11-1: The freer-monad encoding of modular hardware in Rocq. For verification convenience (elided here), our Rocq implementation distinguishes value methods, which do not alter state, from action methods.

A hardware program of type `prog` is parameterised by a method signature `Method` and return type `Ret` (lines 1-3). A component specification is a record type `Spec` containing the internal state type, a per-method evaluation function, and an initial state (lines 4-7). A hardware module is an instance of `Module`, associating an interface signature with a base specification and an implementation of each method (lines 8-12). Programs are interpreted sequentially by invoking the base specification at each method call (lines 13-21) and can be lifted to define component specifications (lines 22-24).

Refinement between two component specifications sharing a method interface is a simulation relation over their state spaces: `Rel` is a simulation if, for any two related states, executing the same method yields equal output and related successor states (lines 25-29). This definition gives rise to a substitution principle, where a parent module driving the same method sequence into the implementation and specification cannot tell them apart.

The encoding is independent of the underlying HDL, enabling language-agnostic reasoning and early-stage exploration. When state and programs are written in a synthesizable fragment of Gallina, they can be proven to correspond to designs in a deeply embedded HDL and synthesized to RTL, for an end-to-end theorem.

11.2 Intercomponent modularity

Modules are specified as abstract specifications exposing method calls, formalised as Mealy machines whose inputs are method calls. We adopt two modeling principles. First, we use *canonical* functional representations (e.g., a hardware FIFO realized as a circular buffer has many states corresponding to a FIFO with one element; we represent FIFOs as lists). Second, we extend the state representation with a *leakage history* that projects the trace of method calls (inputs) down to the information permitted to influence timing (the public inputs)[1]. Submodules are existentially parameterised with the same parameters in Section 10.2: a secret-independent driver over the module's public leakage history fixes its timing, and an unconstrained witness supplies its unspecified outputs.

11.2.1 Example: zero-skip multiplier

Recall the multiplier from Section 9.1. Figure 11-2 gives the Rocq spec.

Functional correctness. The multiplier is functionally correct as long as it computes the product of its inputs and returns results in FIFO order. `Full` and `RespReady` act as the ready/valid handshake. Unlike conventional Bluespec models, where method conflicts block illegal transitions, here `Enq` and `Deq` may always fire—enabling purely local reasoning about each method, which simplifies verification; we synthesize circuits from this style in Chapter 13.

[1] The leakage history could also carry data declassified during execution, as in the top-level ISA spec, but our case studies did not require such declassification.

It is therefore the caller's responsibility to check `Full` before an `Enq`, and `RespReady` before a `Deq` or using the result of a `Peek`.

Security: the leakage contract. A multiplier implementation is secure if its timing behaviour is a deterministic function of the specification's declassified leakage trace—here, the history of action-method calls (`Enq`, `Deq`, and `Tick`) with each `Enq` recorded along with whether either input argument is zero. Zero-skip optimization is permitted because that bit is logged in the trace; all other latency variation must be a function of the trace, ruling out operand-dependent timing channels. It is the caller's responsibility to ensure that the zero-operand bit is indeed public.

Specification structure. The specification state tracks a FIFO of outstanding requests and a history of past operations. To admit a range of implementation choices (varying pipeline depths, multiplication algorithms), it is parameterised over two driver functions of only the public leakage trace, `resp_ready` and `is_full`, so response time depends exclusively on public or explicitly declassified data. To model unspecified behaviour, such as the value of `Peek` when `RespReady` is false (corresponding to the data wire while `valid` is low), the specification includes a witness parameter `default_peek`, letting that output vary or even hold secret-dependent values while invalid and shifting the obligation to the consumer to ignore it until `valid`. The `tick` method marks clock-cycle progression, giving implementations a point to update internal state.

Proof structure. A concrete implementation realizes every method using a synthesizable fragment of Gallina (e.g. using bitvectors and vectors rather than naturals and lists). We provide both a shallow-embedded and a deeply-embedded-HDL implementation (Chapter 12). As in Section 10.5, each existential parameter is instantiated with a shadow copy of the implementation: the witness `default_peek` runs the implementation on the method-call trace, while the secret-independent parameters `resp_ready` and `is_full` run shadow machines on *lifted* method calls:

```
Definition lift_leakage_event (ev: LeakEvent) : Method unit :=
  match ev with
  | LeakDeq ⇒ Deq | LeakTick ⇒ Tick
  | LeakEnq zero_arg ⇒
      Enq (if zero_arg then {| a := zeroes; b := zeroes |} else {| a := ones; b := ones |} )
  end.
```

The proof writer then states and proves a simulation relation between the implementation and both the specification and shadow machines. Against the specification, the relation maps valid implementation states to the canonical list-of-requests representation. Against the shadow machine, it relates public information: equality of public registers and "leakage equivalence"

of states (e.g. data derived from requests need not be equal, only equal modulo whether either operand was zero). The simulation lets the top-level proof treat the multiplier as a black box, substituting any compliant implementation, and allowing a high-level simulation relation in terms of the abstract multiplier's state (FIFOs and traces of method calls).

11.2.2 Specifying the memory

Memory is modeled abstractly in a similar style and with the same API as the multiplier (with a different enqueue request type and peek response type). Functionally, it is a map from address to byte together with a FIFO of requests and the history of method calls updating state (enqueue, dequeue, and tick). This pattern is standard for pipelined submodules that service requests in order, one at a time[2]. As with the multiplier, response time is asserted to be a function of public information; here the leakage trace includes addresses of requests but not data. This work does not tackle verifying memory hierarchies, and so, the spec of memory is trusted.

11.3 Intracomponent modularity

Within a single component such as the processor core, the `tick` cycle-update function is decomposed into sequential steps for individual pipeline stages (fetch, decode, execute, write-back). This lets designers and verification engineers establish Hoare-style pre-/postconditions at stage and function boundaries—structuring each stage to preserve a simulation relation, and supporting both high-level and optimized low-level implementations of functions.

[2]We can extend to out-of-order modules by relaxing the FIFO nature of the request queue and adding a secret-independent parameter to choose which request is handled next.

```
Record req_t := { a : bv width; b : bv width }.
Inductive Method: Type → Type :=
| Enq (arg: req_t) : Method unit
| Deq           : Method unit
| Tick          : Method unit
| RespReady     : Method bool
| Full          : Method bool
| Peek          : Method (bv (width + width)).
Inductive LeakEvent :=
| LeakEnq (zeroArg: bool) | LeakDeq | LeakTick.
Definition leakTrace_t := list LeakEvent.
Definition trace_t := list (Method unit).
Class specParams :=
{ default_peek: trace_t → bv (width + width)
; resp_ready : leakTrace_t → bool
; is_full : leakTrace_t → bool }.
Definition leakage (tr: trace_t) : leakTrace_t :=
  map (fun m ⇒
    match m with
    | Enq req ⇒ LeakEnq (req.a = 0 || req.b = 0)
    | Deq ⇒ LeakDeq | Tick ⇒ LeakTick end) tr.
Record St := { reqs: list req_t; hist : trace_t }.
Definition enq (req: req_t) (st: St) :=
  {| reqs := if is_full (leakage st.hist) then st.reqs
             else st.reqs ++ [req];
     hist := st.hist ++ [Enq req] |}.
Definition deq (st: St) :=
  {| reqs := if resp_ready (leakage st.hist)
             then list.tail st.(reqs) else st.(reqs);
     hist := st.hist ++ [Deq] |}.
Definition tick (st: St) :=
  {| reqs := st.reqs;
     hist := st.hist ++ [Tick] |}.
Definition full (st: St) := params.is_full (leakage st.hist).
Definition respReady (st: St) :=
    match st.(reqs) with
    | [] ⇒ false (* for functional correctness *)
    | _ ⇒ resp_ready (leakage st.hist) end.
Definition peek (st: St) :=
  if respReady (leakage st.hist) then
    match st.reqs with
    | [] ⇒ default_peek st.hist
    | req::_ ⇒ req.a * req.b end
  else default_peek st.hist.
Definition evalMethod {A} (m: Method A) (st: St) : A * St :=
  match m with
  | Enq req ⇒ ((), enq req st)
  ...
  | Full ⇒ (full st, st) end.
```

Figure 11-2: Zero-skip multiplier specification in Rocq. NB: simplified to remove value/action method distinction.

Chapter 12

Verification Case Study: a Pipelined Processor

We verify a four-stage pipelined core implementing a representative subset of RISC-V—with arithmetic, memory, control, and CSR instructions—against the cycle-accurate specification of Chapter 10, composed with the model of memory of Section 11.2.2.

Contract bug found in our work. We initially verified our processor directly against the constant-time policy: it declassifies load/store addresses, branch conditions, and whether a multiplier operand is zero. Verifying hardware against the contract exposed a declassification our software-level intuition had missed: writes to the interrupt-configuration CSRs (`mie`, `mtvec`) must themselves be leaked, because the processor jumps to `mtvec` on an interrupt, and thus the trap target influences observable control flow. Prior work verifying constant-time software down to RISC-V [25] did not account for this leakage, because reasoning at the ISA level alone does not force the question. The proof could not be completed without the extra leakage clause, demonstrating that verifying hardware against ISA specs catches subtle, easy-to-miss policy omissions.

Processor stages. The processor is a standard fetch-decode-execute-writeback pipeline: fetch speculates instruction loads using a BTB-predicted PC and an epoch tag; decode checks the epoch, stalls on scoreboard hazards (and until older instructions commit before a CSR access) then reads registers; execute resolves branches (bumping the epoch and flushing on mispredict) and dispatches variable-latency work to the multiplier and memory; writeback commits results and takes precise exceptions and interrupts (flushing the pipeline if need be). The case study's focus is not on the pipeline but on aspects connecting it to the spec: the visibility point, shadow core, and driver.

Identifying the visibility point. The specification is one-instruction-at-a-time, but the implementation has many instructions in-flight. The *visibility point* of an instruction is the cycle at which it becomes bound-to-commit—no longer discardable by an older instruction's mispredict, exception, or interrupt. It tells the driver when to advance the specification

(for functional-correctness proof) and it marks when an instruction's leakage (e.g. a branch direction) is safe to declassify to the shadow core. It is sufficient to define a visibility point (which could differ based on instruction type) such that any leakage information that could influence execution time in the shadow machine does so strictly after the visibility point. Then, the leakage information is available to be used early enough by the shadow core to replicate the implementation's timing behaviours.

In our core, we safely define the visibility point as a simple predicate on occupancy of pipeline queues: an instruction is visible when it is at the head of the decode-to-execute (`d2e`) queue, there are no instructions at the execute-to-writeback (`e2w`) queue, and it is at the correct epochs. With `e2w` empty, no older instruction remains that could trap or be interrupted. This visibility point suffices as leakage information is not needed in the shadow machine until the execute stage. Tighter, state-dependent conditions are possible (e.g. a lone, in-flight instruction with correct epochs is always correct-path) but not necessary.

Driving nondeterministic choices. Analogously, the driver should not drive the specification past nondeterministic choices influencing observables (MMIO, interrupts) until the implementation resolves nondeterministic choices. For example, the driver should not drive the spec to not take an interrupt until the implementation has resolved to not take an interrupt after the corresponding instruction in the spec. The driver is untrusted: a poor choice of driver will not lead to bugs in the security guarantee but simply a failure to prove the theorem.

Constructing the shadow core. To prove that the processor's timing is independent of secrets, Granite's framework uses a public shadow core that replicates the processor's control logic but operates only on public data. Data memory is initialised with constant dummy values (e.g. zero), with data affecting control flow driven by the public leakage trace. Before the execute stage, the shadow core runs identically to the real core: instructions are fetched speculatively based on the public pc and public branch-predictor state, and registers read from the shadow core in the decode stage contain dummy data. At execute, it substitutes leakage from the secret-dependent decisions:

1. The output of control logic determining whether a branch is taken for a control instruction is replaced with the Boolean from the leakage trace.
2. The address of a memory load/store is replaced with the address from the leakage trace.
3. Arguments to the multiplier are reconstructed based on a Boolean of whether either argument is zero (e.g. by passing in zeroes if true and ones if false).
4. Writes to the `mie` and `mtvec` CSRs are replaced accordingly in the shadow machine.

Because every decision that eventually affects timing is replaced by public leakage, the shadow core's timing is a function of only public data.

Relating the implementation and ISA machine. The driver keeps the specification in-sync with the implementation, advancing it at visibility points as follows:

- When an instruction leaves `e2w` to commit next cycle (its multiplier/memory responses are ready), the specification is at `StepWaitMMIOResp` or `StepInterrupt`. The driver advances the specification two steps for an MMIO response and one otherwise, passing the interrupt bit and transitioning to `StepLeak`.
- When an instruction advances from `d2e` to `e2w` (hence correct-path), all older instructions have committed, and the specification is at `StepLeak`. The driver advances the spec two steps (corresponding to `StepLeak` and `StepInstr`), ensuring leakage information is available before the implementation executes the instruction.

The implementation and ISA state machine are related by a flushing relation [72]: an instruction in `e2w` is related to the specification after it commits (the spec machine already executed the instruction and updated the register files), and unresolved nondeterminism (e.g. whether an interrupt is taken) is delayed in the specification until the implementation resolves it.

Relating the implementation and shadow machine. The simulation relation between the implementation and shadow machine equates state with downstream effects on timing. For example:

- Memory/multiplier: request histories are equal modulo store data (no effect on timing) and zero-operand quotient, respectively, forcing identical latencies.
- Pipeline FIFOs: equal occupancy and valid bits (hence identical stalls) and equal public bookkeeping state (such as instructions and epoch bits).
- Branch predictor: updated only from public information, so its state (modeled as a trace of method calls) stays public. This invariant is used to guarantee that the PC and speculated instructions remain public.

Notably, specifying leakage components of modules in terms of traces of method calls enables a streamlined recipe for stating simulation relations that is independent of underlying implementations. We prove simulation relations are preserved at each fetch-decode-execute-writeback stage boundary of the processor, for all submodule implementations satisfying their respective specifications (recall Theorem 11.0.1).

Chapter 13

Integration Verification

This section describes how software can be verified against the ISA specification via a certified static analysis and extends the processor proof down to RTL via Quartz, yielding an end-to-end proof that eliminates the ISA specification from the TCB. The static analysis and HDL are not the project's focus but serve to validate the ISA spec and our methodology for integration verification, culminating in a noninterference theorem for a constant-time Salsa20 binary down to RTL.

13.1 Software static analysis

We implement a sound (not complete) ISA-level static analysis deciding whether a RISC-V binary is constant-time. Implementation and proof of the analysis took ∼2 days.

Definition 4 (Static analysis sound)**.** $\forall prog.\ analyze(prog) \Downarrow Safe \rightarrow IsConstantTime_{ISA}(prog)$, where $\Downarrow Safe$ means the analysis terminates with outcome $Safe$.[1]

Definition 5 (Constant time)**.** A program is constant-time if for all instantiations of the existential parameters, the ISA-level leakage trace is secret-independent.

$$IsConstantTime_{ISA}(prog) \triangleq \forall Driver,\ Witness, n, \vec{Pub}_n, \vec{Sec}^n_1, \vec{Sec}^n_2.$$
$$O^n_{Cycle-ISA\{Driver,Witness\}}(\vec{Pub}_n, \vec{Sec}^1_n) = O^n_{Cycle-ISA\{Driver,Witness\}}(\vec{Pub}_n, \vec{Sec}^2_n)$$

Theorem 13.1.1 (End-to-end noninterference)**.** Let μ be an implementation satisfying Definition 2. For any program with $analyze(prog) \Downarrow Safe$,

$$\forall n, \vec{Sec}^1_n, \vec{Sec}^2_n.\ O^n_\mu(\vec{Pub}_n, \vec{Sec}^1_n) = O^n_\mu(\vec{Pub}_n, \vec{Sec}^2_n).$$

Proof sketch. Soundness of the static analysis gives $IsConstantTime_{ISA}(prog)$, so the ISA leakage trace is secret-independent; since the leakage transformer reads only public data and

[1] In Rocq, formalised as the existence of a sufficient fuel value.

$$\texttt{SymbVal}\ \tau ::= \texttt{Public (n: } \tau\texttt{) | Secret} \qquad \texttt{RegFile} = \texttt{Reg} \to \texttt{SymbVal Word}$$

$$\texttt{Memory} = \texttt{Addr} \to \texttt{SymbVal Byte}$$

$$\texttt{st} \in \mathrm{SymbState} = \{\texttt{Pc : Addr, Rf : RegFile, Mem : Memory, Csrs : CsrFile}\}$$

$$\texttt{res} \in \mathrm{SymbResult} ::= \texttt{Safe} \mid \texttt{Unsafe} \mid \texttt{Running(st)}$$

$$\frac{\texttt{rs1} = 0 \quad \texttt{rs2} = 0 \quad \texttt{offset} = 0 \quad \texttt{interruptsDisabled(st)}}{\langle \texttt{Beq rs1 rs2 offset, st} \rangle \to \texttt{Safe}}\ \textsc{Beq-Spin}$$

$$\frac{\neg\texttt{Spin} \quad \texttt{st.Rf(rs1)} = \texttt{Public v1} \quad \texttt{st.Rf(rs2)} = \texttt{Public v2} \quad \texttt{v1} = \texttt{v2} \quad \texttt{aligned(st.Pc} + \texttt{offset)}}{\langle \texttt{Beq rs1 rs2 offset, st} \rangle \to \texttt{Running(st}[\texttt{Pc} \leftarrow \texttt{st.Pc} + \texttt{offset}])}\ \textsc{Beq-Taken}$$

$$\frac{\neg\texttt{Spin} \quad \texttt{st.Rf(rs1)} = \texttt{Public v1} \quad \texttt{st.Rf(rs2)} = \texttt{Public v2} \quad \texttt{v1} = \texttt{v2} \quad \neg\texttt{aligned(st.Pc} + \texttt{offset)}}{\langle \texttt{Beq rs1 rs2 offset, st} \rangle \to \texttt{Unsafe}}\ \textsc{Beq-Exn}$$

$$\frac{\neg\texttt{Spin} \quad \texttt{st.Rf(rs1)} = \texttt{Public v1} \quad \texttt{st.Rf(rs2)} = \texttt{Public v2} \quad \texttt{v1 != v2}}{\langle \texttt{Beq rs1 rs2 offset, st} \rangle \to \texttt{Running(st}[\texttt{Pc} \leftarrow \texttt{st.Pc} + 4])}\ \textsc{Beq-Not-Taken}$$

$$\frac{\neg\texttt{Spin} \quad \texttt{st.Rf(rs1)} = \texttt{Secret} \vee \texttt{st.Rf(rs2)} = \texttt{Secret}}{\langle \texttt{Beq rs1 rs2 offset, st} \rangle \to \texttt{Unsafe}}\ \textsc{Beq-Secret}$$

Figure 13-1: Symbolic state and representative `Beq` rules of the constant-time analysis.

that trace, its output $LT^n(\vec{Pub}_n, O_{Cycle-ISA})$ is also secret-independent; and Definition 2 equates this output with O_μ. Noninterference is a trace property preserved by trace equivalence, so the result is preserved under successive trace-equivalence proofs down to RTL (Section 13.2). □

Static analysis. The static analysis symbolically executes the program over states that tag each register, CSR, and byte in memory as `Public n` or `Secret` (imprecise but sufficient in our case study, where the only initial public data is instruction memory), ensuring all branch conditions, arguments to variable-latency instructions, and addresses of memory accesses are independent of `Secret` values. Our analysis is not designed to handle infinite loops (it terminates soundly on the spin-loop pattern `Beq x0 x0 0`) or interrupt/exception handlers. Figure 13-1 shows the state representation and the representative `Beq` rules; a secret operand is rejected (Beq-Secret), a public branch advances the symbolic PC (Beq-Taken), and a misaligned branch target is rejected (Beq-Exn). The remaining rules are analogous.

Salsa20 example. We compile an existing constant-time C implementation of Salsa20 to RISC-V with a standard toolchain, treating data memory (key, message, nonce) as secret and the program as public, and run the analysis in under a second. Theorem 13.1.1 yields an

end-to-end noninterference proof for the binary on our processor. This proof rules out bugs in intermediate layers, including any bugs in our encoding of the RISC-V spec, and also does not require trusting the RISC-V compiler.

13.2 Lowering to RTL via Quartz

Our top-level proofs use a shallow embedding in a synthesizable fragment of Gallina, describing a cycle-accurate design as a first-order functional program using operations commonly available in hardware-description languages. To validate this design choice, we also give this fragment a deeply embedded syntax we call Quartz, and prove the design thus expressed equivalent to the Gallina-native version. In Quartz, functions are written in terms of combinational expressions and let-expressions, methods are functions that take state and input as arguments and return state and output, modules are collections of methods operating on the same type. This equivalence proof is straightforward, only deviating from a partial evaluation of the Quartz interpreter to reconcile encoding details such as use of native Gallina structs vs. deeply embedded structs interpreted as tuples (as Gallina does not support polymorphism over lists of struct fields).

The Quartz implementation of the processor is pretty-printed as SystemVerilog, generating reasonably readable code where variable names are preserved and e.g. source-level structs are represented as (packed, synthesizable) SystemVerilog structs. While we consider it desirable to verify the translation, we are not aware of an adequately featureful formalisation of SystemVerilog semantics against which it could be performed. Thus the 200-line syntax-mapping function is currently part of our TCB. We confirmed that the extracted top-level cycle function (and a trivial always-block wrapper) can be successfully synthesized for the ECP5 FPGA using open-source tools Yosys and NextPnR [**?**], but we have not integrated it into a SoC design for post-synthesis testing.

To prove another RTL description of the same cycle-update function in, e.g., Verilog is equivalent to our implementation, one could use automated tools for equivalence checking at RTL. The proof obligation amounts to proving equivalence at a single cycle, which is less likely to run into challenges related to state-space explosion.

Chapter 14

Discussion

This section discusses what is in the trusted computing base (Section 14.1); what changes to the spec and proof were needed when adding branch predictors, exceptions, and interrupts (Section 14.2); how the methodology applies to verifying the processor independently of the memory subsystem (Section 14.3); and limitations and nongoals (Section 14.4).

14.1 Trusted computing base

The TCB contains the $Cycle{-}ISA$ contract specifying functional and leakage semantics of a fragment of RISC-V (∼700 LoC), the observation/adversary functions defining the I/O interface, the abstract model of the memory hierarchy, the model of the external world, the Rocq kernel, and the Quartz pretty-printer from circuits to Verilog. The RTL processor, leakage transformer, driver, and every submodule implementation are not in the TCB; they are eliminated via machine-checked proof. When connecting with a software proof, such as via a verified software static analysis, the $Cycle{-}ISA$ is eliminated from the TCB. Intermediate specification layers are not in the TCB.

14.2 Design modifications

Our initial design and proof had no branch predictors nor support for exceptions, interrupts, and CSRs. We discuss how the proofs changed as these features were added.

Adding a branch predictor. Adding a branch history table and branch target buffer took less than 1 day with no changes to the spec. Predictors are specified as abstract state machines whose state is public (defined as a trace of method calls). As our processor enforces that branch predictors are only updated with public information, the branch histories in the implementation and shadow machine are equivalent. As there is no functional-correctness obligation, instantiating the branch-predictor specs with concrete implementations is trivial.

Adding exceptions and interrupts. This extension required modifying the spec, adding CSRs for handling exceptions, exception semantics for instructions, and an interrupt-handling step. Additionally, it required extending the driver commands from simply outputting the number of steps to including whether an interrupt is taken at the interrupt-handling step. Notably, adding interrupts moved the visibility point later: an instruction in the execute-to-writeback queue may now be followed by an interrupt that flushes the pipeline, so the driver and invariants were updated to treat an instruction as visible only once no older instruction can cause a flush.

A multicycle processor. To validate that the specification supports different processor architectures, we also verified a three-stage multicycle design, which required a significantly different top-level proof but no meaningful changes to the spec.

14.3 Independent verification of the processor

Fjfj [15] verifies a processor independently of the memory subsystem; the core question is specifying allowed load/store behaviour, since a processor may speculatively issue loads absent from the sequential ISA and reorder loads and stores. These speculative loads are permitted when the speculative addresses and the decision to issue are secret-independent. We give an example proof of a processor specified independently of memory, allowed to have speculative loads: the processor specification has load buffers and is existentially parameterised with a driver that can instruct it to take a normal step, issue a load request, or receive a response. As the driver is a function of only public information, implementation observations remain a function of public information—also demonstrating how driver commands generalize to specs with different next-state transitions. This specification assumes no functional correctness of memory, specifying allowed behaviours of the processor in the face of arbitrary memory behaviour. As in Fjfj [15], one should be able to prove that composing with a well-behaved memory yields the top-level ISA spec, though the challenge is that the two specifications' leakage traces differ.

We found that proving a processor while assuming an abstract model of memory allowed for a more satisfying specification without many additional assumptions. An implementation processor would still be free to issue loads and reorder stores speculatively, as these are internal to the system and invisible at the observation level. A specification of the processor independently of assumptions about memory has to specify what memory reorderings are allowed. This processor spec may be important for multicore semantics and weak memory models, but it is unnecessarily complicated for single-core designs. Nevertheless, we provide an example of how one could implement a specification machine with speculative loads and load buffering, while encapsulating nondeterminism and specifying leakage semantics.

14.4 Limitations and nongoals

The proofs we demonstrate rely on determinism of the hardware-description-language fragment we use, and thus it is not immediately clear how to extend our approach to reason about combined behaviour of modules in different clock domains. Less fundamentally, we use a minimal RISC instruction set (assorted instructions from RISC-V), consider only single-hardware-thread executions with read-only instruction memory (we expect challenges with extensions on this front to be predominantly related to functional correctness rather than timing leakage), and do not actually connect our proofs to the compiler-verification work [25] that inspired aspects of our ISA specification. The only side channel considered in this work is cycle-by-cycle timing; observations of within-cycle timing or power consumption are out-of-scope, as are attacks involving physical access.

Part III

Concluding Remarks

Chapter 15

Discussion

This chapter discusses lessons learned, provides additional context surrounding design decisions, and discusses future work.

15.1 Making concurrent hardware verification sequential, cycle-accurate, and tractable

15.1.1 Tension between ORAAT and cycle-accurate semantics

The one-rule-at-a-time (ORAAT) abstraction of Bluespec offers a compelling "guarded atomic actions" semantics for sequential reasoning of concurrent hardware designs, enabling software-like reasoning of individual rules. When work began on this thesis in 2018, Kami [23] was the state-of-the-art for mechanised proofs of functional correctness for designs expressed in a language with ORAAT semantics. However, both Kami and Bluespec's semantics were not designed to be cycle-accurate, important for reasoning about timing side channels; instead, they were designed with semantics for nondeterministic execution of rules (at the next step, any rule whose guards are met can fire) without a notion of cycles.

In 2020, Bourgeat et al. [16] released Kôika, a derivative of Bluespec with deterministic, cycle-accurate ORAAT semantics based on a user-provided schedule of rules together with a verified compiler to circuits. A core technique used in the language to enable data to flow between rules (for a read to read data written by a previous rule in the same cycle) is making Bluespec's Ephemeral History Registers (EHRs) explicit in the language. Reads and writes are annotated with ports 0 or 1, logged in per-cycle logs, and checked against the log-to-date for conflicts with previous reads and writes (causing the rule to abort upon failure). These conflict checks allowed Kôika's compiler to guarantee that the observed behaviour can be reproduced with serial execution of rules but critically added non-rule-local reasoning to the semantics; each read/write needed to be checked against the log of previous reads/writes to the same register in that cycle. A rule's semantics depended on previous rules, and each read/write was turned into a branch. Furthermore, the same simulation relation could not be

used for each rule, requiring additional information to track register read/write conflicts.

There are various tricks one can play to reduce the cost of non-rule-local semantics (we attempted some such as verified static analyses eliminating many of these checks, or verified symbolic execution) and checking for conflicts that abort rules; we designed Quartz to simplify verification by making checks for conflicts explicit rather than implicit. In particular, it supports a sequential semantics where reads read the latest write without conflict and it is the programmer's responsibility to check method calls are ready before calling them (e.g., checking that a FIFO is not empty before dequeueing). The sequential, conflict-free semantics restores rule-local reasoning. In terms of performance, Quartz is a lower-level representation than Bluespec or Kôika and essentially allows one to write arbitrary circuits (including the same circuits that would be generated from the Bluespec or Kôika compilers). Future work could involve an untrusted performance linter, to aid users in writing better circuits, together with tooling for transformations of circuits that are semantically equivalent.

15.1.2 Language encoding for verification

With proof assistants, seemingly inconsequential design choices for embedding semantically equivalent ideas can have great consequences in the ergonomics of verification. In other words, the surface language in which programs are written should be designed for proof ergonomics rather than compiler convenience (one can prove equivalence to an alternative representation for compiler convenience). Kôika's choices of encoding were designed for compiler verification and were not fully exercised for verification—we ran into challenges with direct usage for verification related to the aforementioned implicit-conflict semantics but also bitvector representation, dependent types, large term sizes, and lack of support for modular verification. Within a proof assistant, one can often construct additional layers of indirection to work around such issues (we eventually sidestepped them by constructing a Kôika-to-SMT toolchain and using Z3 to check sequences of rules). With Quartz, we prioritized design decisions for verification, for example:

- **Bitvector representation**: bitvectors of length n are represented as integers modulo 2^n, as opposed to an inductive type of vectors that has quadratic term size.
- **Struct representation**: structs are represented as nested pairs (with Rocq cleverness to convert native record types to pairs and access via field names to pair projections for usability), as opposed to e.g. a list of pairs of string to type.
- **No computed types**: HDLs contain bitwise operations such as slices and concatenation that can create terms with computed types (e.g. the sum of the input types for concatenation). These computed types create additional verification challenges for equivalence checking with dependent types such as bitvectors. We require operations to have explicit types.

- **Sequential semantics**: reads read from the latest writes and never conflict. This simplifies the semantics and allows modular code reasoning.
- **PHOAS for variable bindings** [22]: use the native host language's machinery for let bindings as opposed to a custom variable context that appears in the type signature of each term.
- **Generalized Algebraic Data Types (GADTs) for programs**: as compared to Kôika creating typed syntax after the fact from untyped syntax, this representation ensures well-typed programs by construction and allows direct control over the program representation used for proofs.
- **Native functions**: functions live in the proof-assistant environment and are not inlined, controlling term size and enabling modular proof obligations (including about parametric interpretations).

Challenges remain: for example, Quartz does not natively support modular compilation.

15.1.3 Certified programming without nondeterminism

Work in this thesis tamed nondeterminism by determinizing top-level and submodule specifications with existential parameters, without loss of generality. Supporting nondeterminism, in the context of security or otherwise, is another area where one can develop clever semantic representations and techniques. Indeed, there are natural examples where one would be tempted to support nondeterminism, such as unspecified behaviour when dequeueing from an empty FIFO. This transforms the semantics from a function to a relation. Work in this thesis began with the starting observation that hardware circuits that are in scope are deterministic and did not encounter a challenge where reasoning with nondeterministic semantics would significantly simplify proofs. Another advantage of deterministic semantics is natural support for specifications as executable oracles—allowing testing and validation of the top-level specs.

15.1.4 Is premature modularity the root of all evil?

Modular specification and verification is often cited as critical for scaling verification. However, it comes with the challenge and cost of designing modular abstraction boundaries between submodules and overhead where changing the contract between submodules involves changes in the submodule specs that would otherwise be invisible. Parfait [8] showed that one could symbolically execute a program on a simple processor for millions of cycles by collapsing abstraction boundaries—reasoning about a single program running on a processor does not require examining all possible executions starting from an arbitrary program. Similarly, MTIsolation found that it sufficed to verify all the rules in each component together, rather than one-rule-at-a-time. But we believe that modular abstractions are needed

to scale to more complex designs and furthermore that their specifications are useful in the design process. Defining the right abstraction boundaries is important—for example in Granite, reasoning about the processor independently of memory can be valuable for proofs of multicore systems but leads to extra complexity in precisely defining allowed orderings of memory requests; whereas proving it against an abstract model of memory still allows memory reorderings without complicating the specification with structures such as load and store buffers to support reordering.

15.1.5 Application to Verilog

Although the case studies in this thesis were not based on Verilog, many of the techniques developed should apply. At least, reducing security to the preservation of single-cycle simulation relations is largely independent of underlying HDL. MTIsolation in particular does not rely heavily on the ORAAT semantics and instead checks properties of multiple rules together. MTIsolation also lowers high-level simulation relations to low-level properties that can be checked with standard model checkers. We expect the same techniques to be applicable to Granite, although the abstract submodule specifications would need to be lowered to model-checker-friendly types. Finally, single-cycle equivalence checking of circuits could be a promising avenue towards obtaining an optimized final hardware circuit while maintaining a representation useful for verification.

15.2 Leakage-aware refinement via determinism

While Part I and Part II target different security properties (isolation between mutually distrusting domains, and secret-independent timing of a single domain), they share the same underlying specification discipline, which Section 1.4.4 introduced as *leakage-aware refinement via determinism*.

Explaining nomenclature. Fundamentally, both MTIsolation and Granite define correctness and security in terms of refinement, where an implementation is admissible if any of its behaviours are also a behaviour of the specification. The specification has more behaviours than the implementation, necessary to support nondeterminism arising from, for example, unspecified implementation design choices. However, as classical refinement is insufficient for information-flow security (and as both MTIsolation and Granite reduce timing-channel security to information-flow properties), we require a *leakage-aware* notion of refinement where we explicitly specify what information can determine nonterministic choices. Our strategy is to fully determinize the specifications via untrusted, deterministic parameters that are constrained in their usage based on information flow. One can then obtain a noninterference property by proving, via a relatively straightforward information flow argument, that only public information flows into timing signals in the specification.

15.2.1 Components for specifying and verifying leakage-aware refinement

We summarize the core components of leakage-aware refinement with the added context of both constructions.

1) A family of deterministic, cycle-accurate specification machines. The specification consists of a family of deterministic, cycle-accurate machines with the same circuit-level interface as the implementation. The circuit-level interface allows the specification to directly constrain the adversary's cycle-level and wire-level observations. Rather than abstracting nondeterminism away (as in functional-correctness frameworks such as Kami and Fjfj, which do not need to consider cycle-accurate timing), all nondeterministic choices are made deterministic via explicit, deterministic parameters. Lowering into a cycle-level machine makes cycle-level timing behaviour explicit.

2) Existential parameters partitioned by information flow. The family of specification machines is parameterised by untrusted functions that are constrained in their usage based on information flow:

- Nondeterministic choices that must not depend on secrets (such as timing) are determined by parameters that receive only public information. For example: in Granite, the *driver* sees only public inputs and declassified leakage; in MTIsolation, each black-box air-gapped machine sees only inputs belonging to the respective security domain.
- Nondeterministic choices that may depend on secrets are determined by parameters that receive both public and secret information. In our case study in Granite, this is the *witness* that determines the signal on the MMIO `data` wire when the output is not `valid`.

This construction allows specifications to soundly capture noninterference properties without overspecifying.

3) A trace-equivalence theorem. The top-level statement has the shape

$$\exists\, params.\ \forall\, inputs.\ \mathrm{trace}(\mathrm{Impl}(inputs)) \ =\ \mathrm{trace}(\mathrm{Spec}_{params}(inputs)),$$

with the existential quantifier outside the universal quantifier over inputs (and secrets) such that a static choice of parameters must explain the implementation's behaviour on every input. This quantifier order makes the parameters untrusted and soundly determinizes nondeterministic choices without leaking secrets: a parameter chosen per-secret could implicitly leak secrets based on the choice of parameter, but a parameter chosen before the secrets are decided cannot leak secrets based on the choice itself. The trace equivalence theorem can simultaneously capture both functional correctness (traces agree on functional outputs)

and nonleakage (the observable trace factors through information-restricted parameters). When adversary observations in the specification are determined solely by parameters with access to only public information (in other words, there exists a leakage transformer from public information to observations), noninterference trivially follows[1]. The same specification and theorem style is used for submodules, with leakage-aware specifications, and the top-level theorem. The specification style is deliberately closer to that of functional-correctness proofs [15, 23, 32] than to established noninterference-verification approaches and gives rise to a modular verification approach.

4) Instantiation by shadow copies of the implementation. Both MTIsolation and Granite instantiate the existential parameters with copies of the implementation that only have access to allowed information flows. MTIsolation instantiates each air-gapped machine as the implementation with circuitry belonging to the other core either erased or provably inert (Section 6.2); Granite instantiates the driver and leakage transformer with a shadow core, a copy of the implementation running on dummy data with data derived from declassified values replaced with the corresponding value from the leakage trace. As the shadow copy reuses the implementation's control logic, it reproduces the implementation's timing exactly; as the shadow copy is constructed from public (including declassified) information only, its timing behaviour depends on only public information. The shadow copies are proof artifacts and are not trusted: a poorly chosen construction leads to failure in proving the top-level theorem.

5) Proof via a single-cycle simulation relation. Both parts reduce the unbounded trace-equivalence statement to preservation of a simulation relation across one clock cycle, decomposed into per-component obligations. Single-cycle induction tames the state-explosion problem that multicycle model checking encounters on deep, speculative pipelines and lends itself to both SMT automation (as in MTIsolation's Rocq-SMT toolchain) and interactive proofs using Rocq (as in Granite, which uses a substitution principle to replace submodules by their specifications).

[1] Proof sketch: trace equivalence asserts the adversary's observations on the implementation machine (O_{impl}) are the same as on the spec machine (O_{spec}) for all public and secret data: $\forall Pub, Sec, O_{\mathrm{impl}}(Pub, Sec) = O_{\mathrm{spec}}(Pub, Sec)$. The existence of a leakage transformer from public information to observations asserts that $\exists LT. \forall Sec^1, Sec^2.\ O_{\mathrm{spec}}(Pub, Sec^1) = LT(Pub) = O_{\mathrm{spec}}(Pub, Sec^2)$. Together:

$$O_{\mathrm{impl}}(Pub, Sec^1) = O_{\mathrm{spec}}(Pub, Sec^1) = O_{\mathrm{spec}}(Pub, Sec^2) = O_{\mathrm{impl}}(Pub, Sec^2).$$

15.3 Future work

Scaling up MTIsolation and Granite. This thesis verifies embedded-class CPUs (mainly, in-order pipelined CPUs with separate instruction and data memories) with either a simple memory hierarchy, in the case of MTIsolation, or an abstract model of memory, in the case of Granite. Extending MTIsolation to out-of-order designs would primarily involve proving the purge operation correct, which could be straightforward if done in a naïve manner or could involve functional-correctness invariants if done in an optimized manner. The primary challenge of extending Granite to out-of-order designs is proving functional correctness of out-of-order designs, specifically defining the invariants and constructing the function mapping the implementation to the OIAAT specification. In practice, this construction involves identifying the "visibility point" of each instruction in the processor, the point when an instruction is bound to commit (but may not necessarily have committed yet). Out-of-order designs contain state distributed amongst different submodules in the processor, and the visibility point is less obvious than in our in-order, pipelined processor. But we expect the same techniques to apply.

Future work could also involve verifying memory hierarchies, accelerators, or heterogeneous systems. The challenge, especially for verifying heterogeneous systems, is formulating specifications for subcomponents. Lastly, improved verification tooling via SMT, AI, or otherwise should also help scale to more complex designs.

Connecting Granite with MTIsolation. A natural next step would be to connect the secret-independent timing style of Granite with the isolation proof of MTIsolation, to obtain an end-to-end proof that running an enclaved program on a shared machine does not leak secrets. This connection would involve specifying and adding a `purge` instruction to Granite and verifying as in MTIsolation that on commit, the processor and each submodule have reached states equivalent to a new machine (one could extend each submodule's API with a `purge` method and prove that it guarantees that, upon a successful purge, it behaves like a machine with its functional state reset and an empty leakage history).

Verifying other HW/SW contracts. The case studies in this thesis used relatively coarse-grained techniques that did not involve new ways of writing software: namely full process isolation or constant-time programming (a technique that has been around for decades). There is a rich space for hardware-software codesign in which software provides more information to the hardware about what data needs to be kept secret, via hardware-supported extensions of the ISA or otherwise, and hardware can aggressively optimize on public data while keeping secrets secret using taint-tracking techniques or otherwise. Furthermore, one could explore a notion of "tunable security" and dynamically reconfigurable security that enables different hardware-security modes and resource configurations based on software

assumptions. Another interesting line of work would be to extend this work to support probabilistic notions of security to reason about probabilistic defenses or to support cryptographic techniques for masking secrets. Related to timing, extending the ISA with (design-specific) performance or latency contracts would be an interesting next direction that would enable better hardware-software codesign.

Tackling other side channels. Other side channels such as power could be tackled by extending the adversary's observation function to include such side channels. For example, one could approximate power based on the number of registers that are high in a given cycle in a design and prove that this measurement is independent of secrets (of course, most designs are not secure against power side channels and obtaining a secure design would likely involve design modifications). Side channels that fall outside the digital abstraction involve trusting that the model corresponds to the physical reality.

Chapter 16

Conclusion

Timing side channels persist not only because they lie outside the traditional architectural contract that defines a processor's behaviour, but also because modern microarchitectures are sufficiently complex and aggressively optimized that their latency and leakage characteristics are effectively impossible to reason about through testing or informal analysis alone. Consequently, the architectural abstraction, as commonly defined, is an unreliable foundation for building confidential and secure systems. This thesis posits that timing-channel-aware contracts, supported by formal guarantees of adherence, are not merely an enhancement but a necessity to repair this foundation.

Concretely, this thesis develops a modular verification methodology for proving that hardware satisfies these contracts. We demonstrate the efficacy of this approach by verifying two distinct but complementary security properties: *interdomain isolation*, ensuring protection across security domains; and *intradomain data-independent timing*, ensuring that execution time of constant-time programs does not depend on secret data. These results establish that strong, machine-checked guarantees regarding timing behaviour are achievable on microarchitectural models, even in the presence of speculative execution and other microarchitectural optimizations.

Beyond security, timing-aware contracts offer a path toward rigorous performance engineering. By converting implicit microarchitectural behaviours into explicit architectural guarantees, these contracts strengthen confidentiality while simultaneously supporting more-predictable performance engineering [41].

In sum, reliable systems require reliable foundations. By elevating timing behaviour from an implementation detail to a first-class architectural concern and verifying that hardware adheres to a timing-aware specification, this work offers a principled path toward processors that support both robust security and high performance in the face of ever-increasing microarchitectural complexity and the fading of Moore's law.

Bibliography

[1] Kôika. https://github.com/mit-plv/koika, 2023. Accessed: 2023-12-06.

[2] Quartz. https://github.com/mit-plv/quartz, 2026.

[3] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. Predicting secret keys via branch prediction. Cryptology ePrint Archive, Paper 2006/288, 2006.

[4] Sam Ainsworth. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 592–606, New York, NY, USA, 2021. Association for Computing Machinery.

[5] 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)*, pages 132–144, 2020.

[6] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing fine-grained constant-time policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 83–96, 2022.

[7] Anish Athalye, Adam Belay, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 97–113, 2019.

[8] Anish Athalye, Henry Corrigan-Gibbs, Frans Kaashoek, Joseph Tassarotti, and Nickolai Zeldovich. Modular verification of secure and leakage-free systems: From application specification to circuit-level implementation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 655–672, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Anish Athalye, M Frans Kaashoek, and Nickolai Zeldovich. Verifying Hardware Security Modules with Information-Preserving Refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 503–519, 2022.

[10] 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

Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 1046–1060, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[12] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in kyber implementations. Cryptology ePrint Archive, Paper 2024/1049, 2024.

[13] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. Flexible instruction-set semantics via abstract monads (experience report). *Proc. ACM Program. Lang.*, 7(ICFP), August 2023.

[14] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Thomas Bourgeat, Jiazheng Liu, Adam Chlipala, and Arvind. Making concurrent hardware verification sequential. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[16] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of Bluespec: a core language for rule-based hardware design. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 243–257. ACM, 2020.

[17] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, 2018. USENIX Association.

[19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 769–784, New York, NY, USA, 2019. Association for Computing Machinery.

[20] 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, PLDI 2020, page 913–926, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Li-Chung Chiang and Shih-Wei Li. Reload+reload: Exploiting cache and memory contention side channel on amd sev. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1014–1027, New York, NY, USA, 2025. Association for Computing Machinery.

[22] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 143–156, New York, NY, USA, 2008. Association for Computing Machinery.

[23] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–30, 2017.

[24] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 607–622, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Owen Conoly, Andres Erbsen, and Adam Chlipala. Smooth, integrated proofs of cryptographic constant time for nondeterministic programs and compilers. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[26] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[27] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Secure processors part II: Intel SGX security analysis and MIT sanctum architecture. *Foundations and Trends in Electronic Design Automation*, 11(3):249–361, 2017.

[28] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 648–664, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7161–7178, 2023.

[30] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[31] Lucas Deutschmann, Johannes Müller, Mohammad Rahmani Fadiheh, Dominik Stoffel, and Wolfgang Kunz. A scalable formal verification methodology for data-oblivious hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(9):2551–2564, 2024.

[32] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 604–619, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. Foundational integration verification of a cryptographic server. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[34] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg 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, 2022.

[35] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 287–305, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1583–1600, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Jean-Claude Graf, Sandro Rüegge, Ali Hajiabadi, and Kaveh Razavi. VMScape: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*, May 2026.

[38] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1868–1883. IEEE, 2021.

[39] Mathé Hertogh, Dave Quakkelaar, Thijs Raymakers, Mahesh Hari Sarma, Marius Muench, Herbert Bos, and Erik van der Kouwe. Rain: Transiently Leaking Data from Public Clouds Using Old Vulnerabilities. In *IEEE Symposium on Security and Privacy (SP)*, 2026.

[40] Jann Horn. Reading privileged memory with a side-channel, January 2018.

[41] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *16th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 19), pages 517–530, Boston, MA, February 2019. USENIX Association.

[42] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[43] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. Flop: Breaking the apple m3 cpu via false load output predictions. In *USENIX Security*, 2025.

[44] Jason Kim, Daniel Genkin, and Yuval Yarom. Slap: Data speculation attacks via load address prediction on apple silicon. In *S&P*, 2025.

[45] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. ileakage: Browser-based timerless speculative execution attacks on apple devices. In *ACM CCS 2023*, 2023.

[46] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[47] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[48] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[49] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, page 104–113, Berlin, Heidelberg, 1996. Springer-Verlag.

[50] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, oct 1973.

[51] Donald C Latham. Department of Defense Trusted Computer System Evaluation Criteria. *Department of Defense*, 198:20, 1985.

[52] Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. Specification and verification of strong timing isolation of hardware enclaves. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 1121–1135, New York, NY, USA, 2024. Association for Computing Machinery.

[53] 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)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[55] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1041–1053, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Daniel Moghimi. Downfall: exploiting speculative data gathering. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[57] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic Hardware-Software Contracts for Security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 72–86, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Nicholas Mosier, Hamed Nemati, John C. Mitchell, and Caroline Trippel. Serberus: Protecting cryptographic code from spectres at compile-time. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4200–4219, 2024.

[59] Rishiyur S. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, page 69–70, USA, 2004. IEEE Computer Society.

[60] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, 2019.

[61] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 226–239, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1737–1752. IEEE, 2023.

[63] K.R. O'Neill, M.R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 12 pp.–201, 2006.

[64] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[65] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Adam Chlipala, and Arvind. Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers. In Tim Sherwood, Emery Berger, and Christos Kozyrakis, editors, *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual, April 19-23, 2021*, ASPLOS 2021. Association for Computing Machinery, 2021.

[66] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002, 2020.

[67] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. Tdxploit: novel techniques for single-stepping and cache attacks on intel tdx. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[68] A. Roscoe, Jim Woodcock, and L. Wulf. Non-interference through determinism. volume 4, pages 33–53, 01 1994.

[69] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 723–735, 2019.

[70] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery.

[71] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299. Springer, 2019.

[72] Jeffrey X Su, David L Dill, and Clark W Barrett. Automatic generation of invariants in processor verification. In *International Conference on Formal Methods in Computer-Aided Design*, pages 377–388. Springer, 1996.

[73] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the*

2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pages 2435–2450. ACM, 2017.

[74] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. RTL Verification for Secure Speculation Using Contract Shadow Logic. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 970–986, New York, NY, USA, 2025. Association for Computing Machinery.

[75] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960. IEEE, 2018.

[76] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1411–1428, Santa Clara, CA, August 2019. USENIX Association.

[77] 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*, CCS '21, page 429–444, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 178–195, New York, NY, USA, 2018. Association for Computing Machinery.

[79] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[80] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2128–2142, New York, NY, USA, 2023. Association for Computing Machinery.

[81] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual Volume 2: Privileged Architecture Version 1.7. Technical report, 2015.

[82] Robin Webbers, Robert Schenck, Wind Wong, Kristina Sojakova, and Klaus von Gleissenthall. Pantomime: Constructive leakage proofs via simulation. *Proc. ACM Program. Lang.*, 10(PLDI), June 2026.

[83] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadowng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, Technical report, 2018.

[84] N. Wistoff, M. Schneider, F. K. Gurkaynak, G. Heiser, and L. Benini. Systematic prevention of on-core timing channels by full temporal partitioning. *IEEE Transactions on Computers*, 72(05):1420–1430, May 2023.

[85] 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)*, pages 428–441, 2018.

[86] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. Pensieve: Microarchitectural modeling for security evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache Side-Channel attack. In *23rd USENIX security symposium*, pages 719–732, 2014.

[88] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery.

[89] Drew Zagieboylo, Charles Sherk, Andrew C. Myers, and G. Edward Suh. SpecVerilog: Adapting Information Flow Control for Secure Speculation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2068–2082, New York, NY, USA, 2023. Association for Computing Machinery.

[90] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 503–516, New York, NY, USA, 2015. Association for Computing Machinery.

[91] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Everywhere all at once: Co-location attacks on public cloud faas. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 133–149, New York, NY, USA, 2024. Association for Computing Machinery.

[92] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Last-level cache side-channel attacks are feasible in the modern public cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 582–600, New York, NY, USA, 2024. Association for Computing Machinery.

[93] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the*

2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, page 1789–1806, New York, NY, USA, 2017. Association for Computing Machinery.