Specification and Verification of Strong Timing Isolation of Hardware Enclaves

Stella Lau
Massachusetts Institute of Technology
Cambridge, MA, USA
stellal@csail.mit.edu

Thomas Bourgeat
EPFL
Lausanne, Switzerland
thomas.bourgeat@epfl.ch

Clément Pit-Claudel
EPFL
Lausanne, Switzerland
clement.pit-claudel@epfl.ch

Adam Chlipala
Massachusetts Institute of Technology
Cambridge, MA, USA
adamc@csail.mit.edu

Abstract

The process isolation enforceable by commodity hardware and operating systems is too weak to protect secrets from malicious code running on the same machine: attacks exploit timing side channels derived from contention on shared microarchitectural resources to extract secrets. With appropriate hardware support, however, we can construct isolated enclaves and safeguard independent processes from interference through timing side channels, a step towards confidentiality and integrity guarantees.

In this paper, we describe our work on formally specifying and verifying that a synthesizable hardware architecture implements strong timing isolation for enclaves. We reason about the cycle-accurate semantics of circuits with respect to a trustworthy formulation of strong isolation based on "air-gapped machines" and develop a modular proof strategy that sidesteps the need to prove functional correctness of processors. We apply our method on a synthesizable, multicore, pipelined RISC-V design formalized in Coq.

CCS Concepts

• **Security and privacy → Logic and verification**; **Side-channel analysis and countermeasures**.

Keywords

Hardware verification; isolation; side channels

ACM Reference Format:
Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2024. Specification and Verification of Strong Timing Isolation of Hardware Enclaves . In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690203

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690203

1 Introduction

We often want to run trusted, security-critical code on the same machine as untrusted code without leaking secrets. We can decompose this problem into two parts: i) attackers should neither be able to observe secrets from nor interfere with trusted code due to being colocated on the same machine, and ii) security-critical code should not itself leak secrets when executing independently on its own machine.

Traditionally, programmers rely on architectural process isolation provided by primitives such as virtual memory and context switching to achieve i) and constant-time programming techniques to achieve ii). However, the isolation and constant-time guarantees that can be enforced with commodity hardware are too weak: microarchitectural timing side channels are exploited to extract secrets from both colocated processes and code traditionally deemed constant-time [9, 18, 21, 27]. Despite years of mitigations, new attack vectors continue to be discovered.

This paper focuses on i). Recent projects on Intel SGX [10] and Keystone [20] introduced hardware modifications to reconstruct isolated processes, yet they remain vulnerable to timing-side-channel attacks [8]. Sanctum [11] and MI6 [6] featured novel hardware modifications to protect against a wider spectrum of microarchitectural attacks, including those using the cache and DRAM controller bandwidth. But, with the complexity of modern microarchitectures, it remains challenging to design architectures free of timing side channels *and* be confident in their security guarantees.

Our approach. We propose to build confidence in mechanisms for eliminating microarchitectural leakages by formally ruling out, with machine-checked proof, interference across isolated processes or security domains, hereby referred to as *enclaves*. This paper presents a methodology to formally verify that a register-transfer-level (RTL) design securely 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

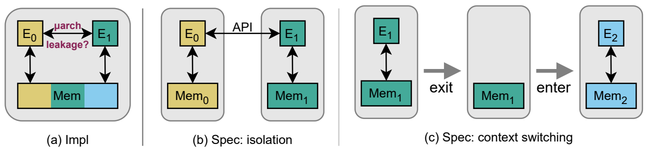


Figure 1: Preview of specifying strong isolation. Different colours denote different enclaves. (a) In the real system, enclaves on the same machine share microarchitectural resources and main memory. (b) In the specification, enclaves run on dedicated machines connected via a well-defined API. (c) Transitioning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

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.

This work provides 1) a formalisation of strong isolation in the presence of context switching, previewed in Figure 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. We extend this formalisation to capture the dynamically evolving security domains present with context switching, answering questions such as how do we eliminate microarchitectural leakage across context switches?; what information is preserved across context switch?; what is the initial state of a new enclave? We model context switching as "throwing away the old machine and starting a new machine" and *explicitly* define information preserved across context switches and used to initialise a new enclave. This pattern rules out leakages via leftover, e.g., branch-predictor state and L1 caches when transitioning across protection domains as any leftover microarchitectural state is "thrown away" when exiting an enclave (due to not being explicitly preserved) and thus cannot be used to initialise a new enclave. Crucially, by adopting the principle of using an allowlist rather than a blocklist, our specification rules out sources of interference or leakage not yet discovered. Furthermore, the specification is parametric on non-security-critical functional and timing behavior. 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 2) a methodology to formally verify isolation in an RTL design that sidesteps the need to prove functional correctness. We show that securely implementing and proving isolation can be decomposed into two parts largely independent of functional correctness: i) enforcing a simulation relation between running enclaves in the implementation and specification through spatially and temporally partitioning resources, and ii) 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 Coq-SMT strategy and toolchain, MTIsolation[1], 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 future work on full-stack, hardware-software guarantees.

Finally, we provide 3) a prototype multicore, pipelined RISC-V system formally proven to implement strong timing isolation for enclaves. The system is written in the Kôika [7] hardware description language (HDL) and translated into circuits using Kôika's verified compiler, ensuring that the security properties verified with Kôika's semantics are preserved with cycle-level accuracy. We provide a machine-checked proof and explore design modifications involving branch predictors and caches. We validate the prototype's functionality through executing a suite of RISC-V and C tests compiled with standard toolchains using Cuttlesim [26] 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.

Contributions. In summary, this paper contributes:

- A formalisation of strong timing isolation supporting dynamically evolving security domains based on the principle of using allowlists instead of blocklists.
- A modular methodology and Coq-SMT toolchain, MTIsolation, for formal verification of isolation for circuit-level designs that sidesteps functional-correctness proof.
- A case-study prototype and machine-checked proof of a multicore, pipelined RISC-V system implementing strong timing isolation.

Source code for MTIsolation and the case studies is available at https://github.com/mit-plv/isolation.

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 [17, 23]. We focus on decoupling security from functional correctness:

[1]For "Microarchitectural Timing Isolation". Pronounced "Mount Isolation".

verifying functional correctness was a nongoal. 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.

2 Background

2.1 Microarchitectural Timing Side Channels

For performance, microarchitectures contain resources such as caches and branch predictors. The availability of these resources affects execution time. Most modern architectures are designed to maximize resource utilization by sharing, e.g., cache lines between processes. However, when processes can measure time, there is opportunity for information leakage and interference through microarchitectural timing side channels [22, 34]. For example, an attacker colocated with a victim can *prime* a cache line with data, then *probe* the line, measuring the latency of its memory request, to observe whether the victim evicted the line. Moreover, they can interfere with the victim's timing, breaking isolation boundaries. Contention for these resources has been exploited to leak cryptographic secrets [5]. Numerous defenses have been proposed, such as disabling speculation and set-partitioning the cache. However, attacks continue to be discovered as these solutions are either unverified or too localised [6].

2.2 Enforcing Strong Timing Isolation

Work on trusted execution environments (TEEs) such as Intel SGX and Keystone introduces primitives to protect the execution of sensitive programs, but these systems are vulnerable to microarchitectural timing side channels. Sanctum and MI6 add hardware modifications to achieve strong isolation by spatially and temporally partitioning resources. Resources are assigned to protection domains independently of their demand; for example, last-level caches and DRAMs are partitioned across protection domains, and a domain can have at most, e.g., half the DRAM controller bandwidth irrespective of the memory intensity of colocated domains. In addition, microarchitectural state persisting across context switches breaks strong isolation. 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. We develop a specification of strong isolation that captures the dynamically evolving security domains present with context switching and adapt the partitioning and purging mechanisms proposed in MI6 to achieve provable strong isolation.

2.3 Hardware Verification with Kôika

Circuit-level formal verification of isolation is useful because, first, the precise statement of the isolation policy is often complex, and formalisation subjects it to additional scrutiny; and, second, correctly implementing microarchitectural isolation is challenging. However, verifying RTL is difficult as even simple designs have large and complex state spaces.

To verify a hardware design, we need a mathematical, cycle-accurate characterisation of the system. We implement our prototype in Kôika, a rule-based language in the same family as Bluespec [1], but we expect our methods are also largely applicable to designs implemented in non-rule-based HDLs. 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. Kôika and its formal, cycle-accurate semantics are mechanized in Coq, along with a verified compiler from Kôika to circuits.

3 Overview

Figure 2a introduces the implementation or "real-world"[2] system and enclave programming model used as a motivating example throughout this paper. Figure 2b 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), which guarantees that any information exfiltrated by a colocated attacker could be obtained by a remote attacker. As such, we do not protect against remote timing side channels such as NetSpectre [28]. 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 switching. For instance, our example model in Figure 2 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 [17], 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.

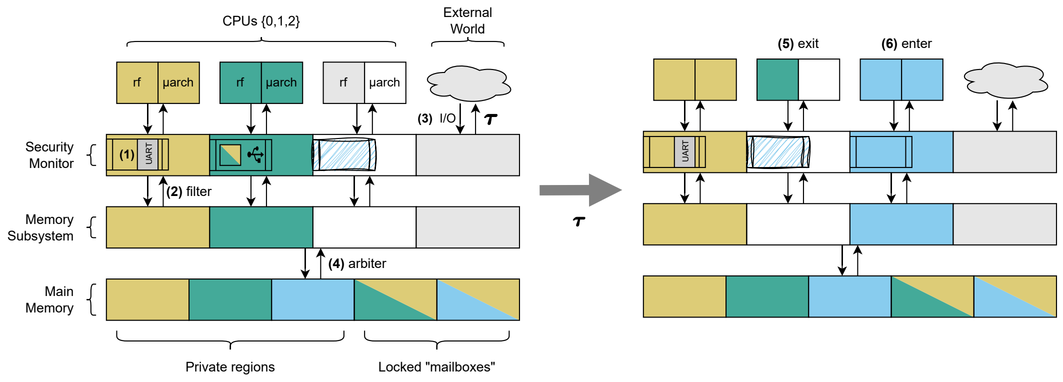
We do not consider availability or physical side channels.

3.2 Components to Prove Strong Isolation

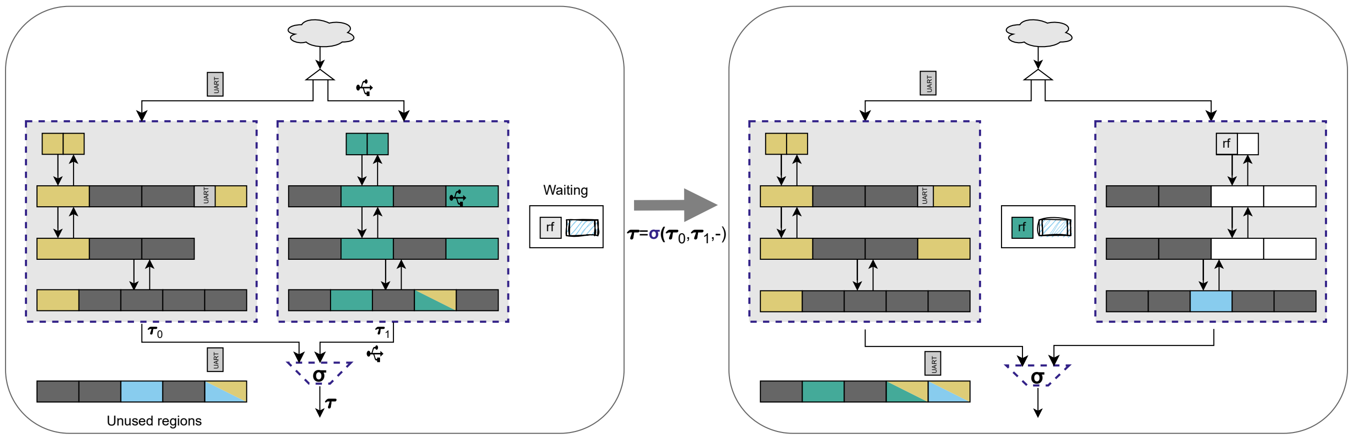
Here, we outline the components needed, and summarize our instantiations thereof, in a formal proof of strong isolation.

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 Coq. We use a hardware security monitor to enforce

[2]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 2: 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.

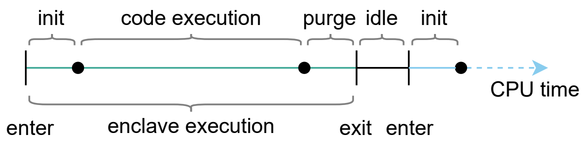
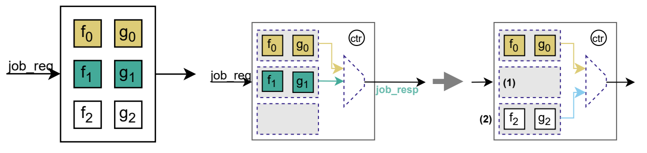


Figure 3: An example timeline of enclave execution on a core. We verify strong isolation holds throughout an enclave's lifetime or execution, inclusive of any initialization, code execution, and purging. We specify behavior on enclave context switching (exit and enter).

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 2b,



(a) Implementation **(b) Specification step transition**

Figure 4: (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.

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 initializing).

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)[3].

An implementation is secure if it has the same observable behavior 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.

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[4]

[3]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).

[4]Our verified implementation uses two sets of boxes—we use three here to compactly show 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.

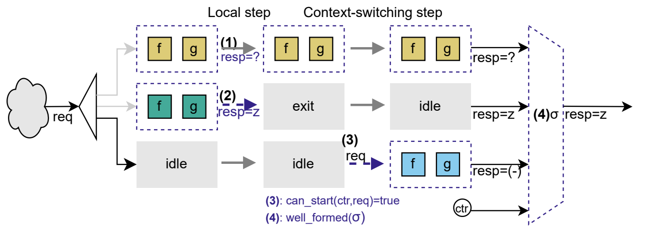


Figure 5: Resource-isolation specification's state transition, split into local and context-switching steps. (1): When running, subsidiary machines behave independently, with no inputs. (2): When finishing, no local machine state is preserved. (3): If a job `can_start`, a brand-new machine is initialised with only the new job. (4): σ combines the outputs from subsidiary machines. I/O arrows abbreviated for intermediate steps.

f and g boxes, as shown in Figure 4a. 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 arbitrator. Potential sources of security violations in insecure implementations include:

(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 eagerly run 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).

Specification. Figure 4b summarizes a spec system ruling out the above violations. The state consists of three subsidiary air-gapped machines, either `Running` or `Waiting`. Figure 5 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.

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.

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 § 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 2. As in § 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 § 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:

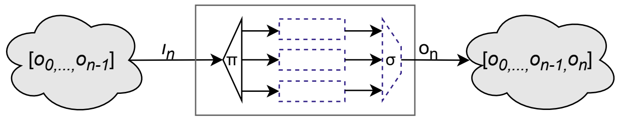


Figure 6: External world as a function over output history.

Definition 2 (Static strong isolation with fixed inputs)**.** *Suppose we have an implementation machine*[5] $M := (S, s_0, I, O, \delta, out)$ *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[6] *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 initialized with separate secrets. We detail the modeling of inputs 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 6. 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.

[5]A finite Mealy machine.
[6]As in § 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."

In other words, $\Omega_{out_\omega} := (S, s_0, O, I, \delta, out_\omega)$[7] 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 system of the implementation and secure system, 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.*

4.2.1 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, i.e., 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.

[7]A Moore machine.

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," initialized based on programming model. We use an allowlist instead of a blocklist to require *explicit* authorization of information preserved upon exit and used in entry. Figure 7 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 § 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. Diagramatically:

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

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 $\forall out_\omega.(M, \Omega_{out_\omega}) \equiv (\texttt{Spec(params)}, \Omega_{out_\omega})$ where $\equiv$ denotes trace equivalence.*

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 2[8]) 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 exclusively owned

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

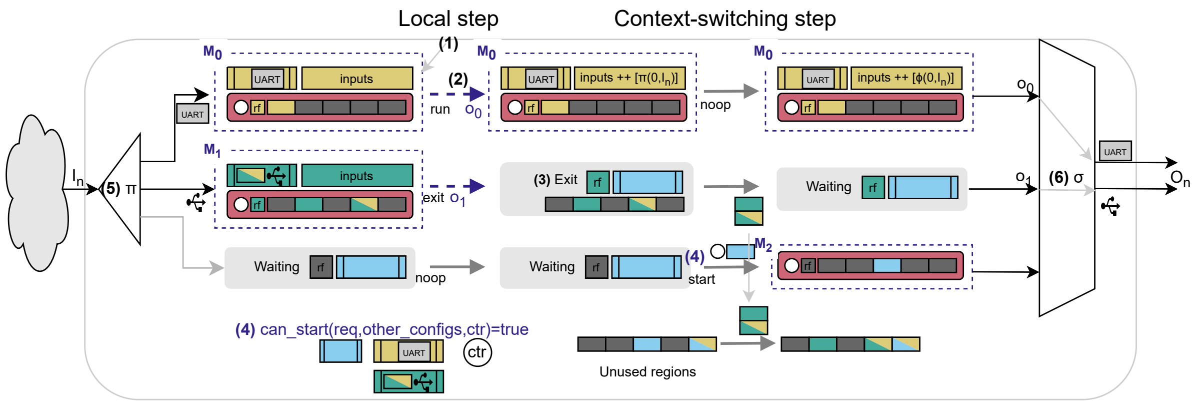


Figure 7: Spec system's transition corresponding to Figure 2, decomposed into local and context-switching steps. (1): `Enclave0` runs yellow enclave with UART. At cycle n, `Enclave0`'s state is a (parameterised) function of its initial state (denoted by the red box, containing cycle counter `ctr`, register 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`.

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 initialized as a fresh machine with assigned memory regions and register file.

5.1 Specification

Figure 2b summarizes a spec for our programming model. Figure 7 details the spec's state-transition function, showing information preserved upon exiting and used in initialisation. Figure 7 illustrates that after running for n cycles, an enclave can be viewed as a (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 mechanize this spec in Coq, with a snapshot in Figure 8.

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 § 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 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.

```
Definition step_local core 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) ol
    | None ⇒ TS_Core (CS_Running mst' config) obs
 | CS_Waiting new rf exit_cycle ⇒ TS_Core st empty_obs.
Definition step_enter core ts_state cycles mem_regions :=
 match ts_state with
 | 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 := init core (cycles+1) new (get_dram params mem_regions config') rf in
      TS_Core (CS_Running machine config')
  else ts_state
 | _ ⇒ ts_state.
```

Figure 8: A snapshot of the Coq specification (extended version in Appendix A.2), parameterised on the red text. 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.

Processor. The processor is based on the four-stage RISC-V core from [7]. 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 § 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 § 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.

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 Coq-SMT approach, using Coq's expressivity for clear specs and to soundly reduce security to single-cycle, circuit-level properties amenable to automatic verification with SMT solvers. We summarize the proof architecture in § 6.1, discuss instantiation of existentially quantified parameters in § 6.2, describe per-component obligations in § 6.3, and detail MTIsolation (the Coq-SMT toolchain) in § 6.4.

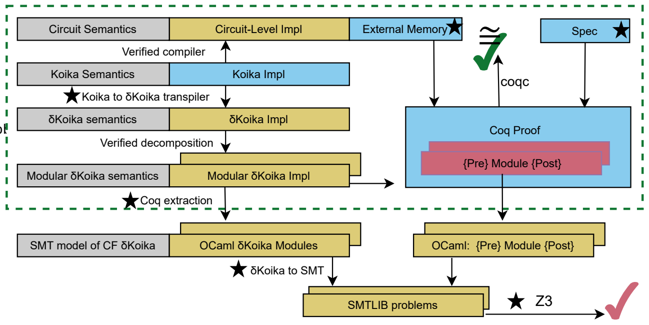


Figure 9: Architecture for proving that a circuit-level implementation satisfies a specification. Grey boxes are provided by MTIsolation. Blue boxes are implemented by the developer. Yellow boxes are intermediate steps generated by MTIsolation. Red boxes denote assumptions within Coq, validated using Z3. The green-dotted box indicates parts within Coq. The star indicates components in the TCB.

6.1 Proof architecture

Figure 9 summarizes our verification framework. The spec is written in Gallina (Coq'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 [2]. 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[9]. To restore modularity, we use static analyses (verified in Coq) 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

[9]Registers 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.

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 Coq spec based on the implementation (see § 6.2), the proof developer states (in Coq) 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 Coq) 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 Coq-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 (§ 6.4), and checks the generated SMT formulae using Z3 [14]. The developer strengthens the invariant as need be, 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 § 7.4).

6.3 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 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 subsytem. The memory subsystem implements a similar purge state machine as the processor, 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 Appendix B.1 with example invariants. We discuss invariants pertaining to caches in § 7.1.

6.4 MTIsolation: a 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 Coq provided by MTIsolation, included in Appendix A.2. 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 Coq 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

```
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 10: `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.

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 Coq. 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 10 shows examples of assertions in MTIsolation's symbolic language in Coq. Symbolic assertions for our design could be expressed in a fragment of 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 Coq before translation to SMT. This sugaring allows convenient reflection with Gallina's `forall` quantifier for concise term representation and usage in Coq proofs.

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. This extension required no changes to the spec and Coq 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 Appendix B.2). For example:

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.

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 initialized 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.

7.2 Verification Cost

Figure 11 shows the lines of code for the implementation, specification, and proof for various examples, and the verification time in Coq and Z3. We proved the resource-isolation example from § 4.1 entirely in Coq, with per-rule specifications and a postcondition semantics for δKôika, and used a hybrid Coq-SMT approach for the other examples.

As proof writing, debugging and iterating on invariants is an interactive process, 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).

	SLOC					Time(s)	
Design	Kôika	Csim[4]	Vlog[5]	Spec	Pf	Coq[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 11: 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.

7.3 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 microachitectural 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.

8 Related Work

Timing-side-channel defenses. TEEs [10] construct enclaves to avoid trusting the OS but are not designed to protect against timing side channels. TEEs typically rely on virtual memory for architectural isolation, whereas Sanctum, its successors [6, 20] and this work depend on physical memory isolation similar to RISC-V's PMP. Sanctum and MI6 use spatial and temporal isolation. They are methodologically closest to this paper but do not tackle the verification challenges in formally proving isolation. Subramanyan et al. [29] 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). Wistoff et al. [32] 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.

Noninterference. Hyperflow [16], mCertiKOS-secure [12], and Komodo [15] formalise noninterference and information-flow security using declassification policies to express allowed leakages. While necessary for expressivity, when defined at a low level associated with a particular implementation's code, understanding the security guarantee requires detailed auditing and deep understanding of the system. We propose a spec that is auditable without understanding low-level details of the implementation and captures isolation in the context of dynamic security domains.

HW/SW security verification. Several works [13, 17, 24, 30, 31, 33] formalized and verified HW/SW contracts for simple microarchitectures. These contracts promise that a hardware model does not leak more than a software model. In this style, an attacker actively controls the victim's initial state but is limited to passive observation of channels while the victim executes. In our formalization, the attacker can provably run concurrently with the victim without interference. The HW/SW contracts focus on speculative constant time without addressing strong isolation. These two properties are complementary yet orthogonal, with different threat models and assumptions. Strong isolation provides guarantees that reduce the attack surface to remote attacks, even for code not written in constant-time style with a clear separation of public/secret data.

To close off remote attacks and prove cryptographic code running on our machine does not leak secrets even through completion time, we would need to compose our isolation guarantees with speculative constant-time guarantees. LeaVe [31] also observes that different techniques can be used to verify ISA versus leakage and that functional guarantees can be composed with leakage guarantees. A difference is that LeaVe assumes ISA correctness (proving a HW/SW contract), whereas we prove hardware isolation independently of ISA.

Knox [4] and Notary [3] use automatic verification tools to collapse abstraction layers symbolically, from software down to RTL, and prove functional and leakage properties about a program on specific hardware without addressing colocation. VRASED [25] verifies a HW/SW codesign for remote attestation. While the above target verification of specific programs, this work proves properties about the microarchitecture *for all programs*.

9 Conclusion

We presented a methodology to formally verify strong timing isolation in enclave systems, ruling out microarchitectural timing side channels with machine-checked proof. Our methodology has demonstrated tractable for analyzing synthesizable hardware designs, not just models thereof. We believe it is worth continuing to explore how the method can generalize to more interesting enclave models and other hardware-security mechanisms.

Acknowledgements

We are grateful to Anish Athalye, Derek Leung, Jules Drean, Kyle Hogan, Samuel Gruetter, Yuheng Yang, and the anonymous reviewers for their feedback that improved this paper.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0018 and by the National Science Foundation under Grant No. CCF-1521584. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or DARPA. The work was also supported in part by Sandia National Laboratories and by gifts from the Amazon-MIT Science Hub and Seagate. Thomas Bourgeat was partially supported by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project.

Data-Availability Statement

Source code for the Coq-SMT toolchain and case studies is available at https://github.com/mit-plv/isolation. The evaluated artifact [19] associated with this paper is available at https://doi.org/10.5281/zenodo.12786597.

References

[1] 2004. 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)*. IEEE Computer Society, USA, 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[2] 2023. Kôika. https://github.com/mit-plv/koika. Accessed: 2023-12-06.

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

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

[5] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*. Springer, 201–215.

[6] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/3352460.3358310

[7] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 243–257. https://doi.org/10.1145/3385412.3385965

[8] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[9] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[10] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086

[11] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[12] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. 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* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 648–664. https://doi.org/10.1145/2908080.2908100

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

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

[15] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 287–305. https://doi.org/10.1145/3132747.3132782

[16] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1583–1600. https://doi.org/10.1145/3243734.3243743

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

[18] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy*. 591–604. https://doi.org/10.1109/SP.2015.42

[19] Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2024. *Artifact: Specification and Verification of Strong Timing Isolation of Hardware Enclaves*. https://doi.org/10.5281/zenodo.12786597

[20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages. https://doi.org/10.1145/3342195.3387532

[21] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/

presentation/lipp

[22] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[23] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 72–86. https://doi.org/10.1145/3470496.3527412

[24] Nicholas Mosier, Hamed Nemati, John C Mitchell, and Caroline Trippel. 2023. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. *arXiv preprint arXiv:2309.05174* (2023).

[25] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A verified Hardware/Software Co-Design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1429–1446.

[26] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Adam Chlipala, and Arvind. 2021. Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual, April 19-23, 2021 (ASPLOS 2021)*, Tim Sherwood, Emery Berger, and Christos Kozyrakis (Eds.). Association for Computing Machinery. https://pit-claudel.fr/clement/papers/cuttlesim-ASPLOS21.pdf

[27] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 753–768. https://doi.org/10.1145/3319535.3354252

[28] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. 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*. Springer, 279–299.

[29] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2435–2450. https://doi.org/10.1145/3133956.3134098

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

[31] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. 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* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2128–2142. https://doi.org/10.1145/3576915.3623192

[32] N. Wistoff, M. Schneider, F. K. Gurkaynak, G. Heiser, and L. Benini. 2023. Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning. *IEEE Trans. Comput.* 72, 05 (May 2023), 1420–1430. https://doi.org/10.1109/TC.2022.3212636

[33] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. 2023. Pensieve: Microarchitectural Modeling for Security Evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 59, 15 pages. https://doi.org/10.1145/3579371.3589094

[34] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX security symposium*. 719–732.

A Additional Coq Code

A.1 Coq Specification of Enclave Isolation

```
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 st 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.
Parameter 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 12: Specification of strong isolation in Coq. The inputs and outputs are bitvectors containing MMIO requests and responses. The specification is parameterised on the text in red.

A.2 Symbolic Language

```
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 13: Symbolic assertion language deeply embedded in Coq

B Memory Subsystem Invariants

B.1 Memory Arbiter

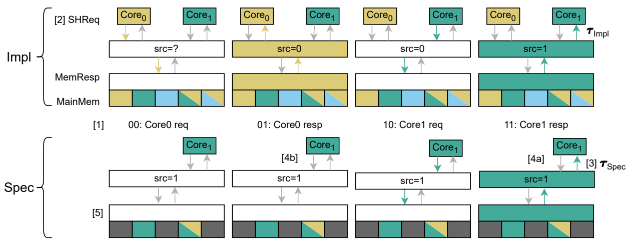


Figure 14: Memory arbiter for single-cycle, single-port memory. [1] Access to memory is time-partitioned: on cycles `00` and `01`, `Core0` and `Core1` respectively can exclusively send requests. [2] Status holding request (SHReq) register tracks the source of the outstanding request. [3] Specification 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.

B.2 Cache Extension

State	Example invariant
Ready	No metadata/cache/mainMem resps
ProcessRequest	Valid meta resp ⟶ 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 ⟶ impl-spec cache data equal
FlushPrivateData	All metadata lines are invalid; no mem resps
Flushed	All μarch state is invalid; no mem resps

Figure 15: Examples of cache invariants proved using SMT