Formally Verifying Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation

by

Anish Athalye

S.B., Massachusetts Institute of Technology (2017)
M.Eng., Massachusetts Institute of Technology (2017)

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 2024

Authored by: Anish Athalye
Department of Electrical Engineering and Computer Science
August 21, 2024

Certified by: M. Frans Kaashoek
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: Nickolai Zeldovich
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

Formally Verifying Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation

by

Anish Athalye

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Hardware and software systems are susceptible to bugs and timing side-channel vulnerabilities. Timing leakage is particularly hard to eliminate because leakage is an emergent property that can arise from subtle behaviors or interactions between hardware and software components in the entire system, with root causes such as non-constant-time code, compiler-generated timing variation, and microarchitectural side channels. This thesis contributes a new approach using formal verification to rule out such bugs and build systems that are correct, secure, and leakage-free.

This thesis introduces a new theory called *information-preserving refinement (IPR)* for capturing non-leakage in addition to correctness and security, implements a verification approach for IPR in the *Parfait* framework, and applies it to verifying hardware security modules (HSMs). Using Parfait, a developer can verify that an HSM implementation leaks no more information than is allowed by a succinct application-level specification of the device's intended behavior, with proofs covering the implementation's hardware and software down to its cycle-precise wire-I/O-level behavior.

This thesis uses Parfait to implement and verify several HSMs, including an ECDSA certificate-signing HSM and a password-hashing HSM, on top of Ibex and PicoRV32-based hardware platforms. Parfait provides strong guarantees for these HSMs: for example, it proves that the ECDSA-on-Ibex implementation—2,300 lines of code and 13,500 lines of Verilog—leaks nothing more than what is allowed by a 40-line specification of its behavior.

Thesis supervisor: M. Frans Kaashoek
Title: Professor of Electrical Engineering and Computer Science

Thesis supervisor: Nickolai Zeldovich
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I'd like to thank my advisors, Frans Kaashoek and Nickolai Zeldovich, for their guidance over the course of a decade. I've known Frans and Nickolai since high school, thanks to the MIT PRIMES program, and after that, as a student in their classes, as their TA, as a UROP in their lab, as their MEng student, and finally as their PhD student. I am deeply grateful for their unwavering guidance and encouragement, in both my thesis research as well as side quests. I attribute so much, from how I think and how I approach technical problems to what I value and what I believe in, to Frans and Nickolai.

Next, I'd like to thank my collaborators, in addition to Frans and Nickolai: Joe Tassarotti, Henry Corrigan-Gibbs, Robert Morris, and Adam Belay. I have been lucky to learn from each of their unique styles of thinking and approach to research.

This research was greatly improved thanks to feedback from Akshay Narayan, Alexandra Henzinger, Ariel Szekely, Derek Leung, Emina Torlak, George Candea, Haibo Chen, Kyle Hogan, Ralf Jung, Sacha Servan-Schreiber, Sanjit Bhat, Stella Lau, Tej Chajed, Thomas Bourgeat, Upamanyu Sharma, Xi Wang, Yun-Shen Chang, and seventeen anonymous reviewers.

I have had the pleasure of working with six wonderful mentees over the years: Noah Moroze, Damian Barabonkov, Ben Kettle, Katherine Zhao, Jess Xu, and Rick Ono.

I thank my colleagues and friends in the Parallel and Distributed Operating Systems (PDOS) group—Akshay, Alana, Alex, Amy, Ariel, Atalay, Austin, Baltasar, Ben, Cody, David, Derek, Frank, Gohar, Hannah, Inho, Jon, Jonathan, Josh, Kevin, Lily, Malte, Ralf, Ryan, Sanjit, Shoumik, Srivatsa, Stephen, Tej, Upamanyu, Yun-Sheng, and Zain—for creating a wonderful lab atmosphere, and for engaging technical discussions and debates.

Finally, I thank my family: my parents Ratnakar and Manisha, and my younger brother Ashay. They have been with me every step of the way and supported me in more ways than you can imagine.

This research was supported by NSF awards CNS-1812522 and CNS-2225441 and by a grant from Google.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main contribution of this thesis is a new formalism and approach for proving the absence of bugs that leak information, such as timing side channels, in addition to correctness bugs in hardware and software. This thesis applies this approach to verifying hardware security modules, with proofs covering their hardware and software stack down to the cycle-precise wire-I/O level.

1.1 Context: hardware security modules

Hardware security modules (HSMs) are widely used as a building block in computer systems where core security functionality is factored out onto this physically-separate device with special-purpose hardware and software. HSMs defend against a class of attacks where an adversary remotely compromises the host machine that the HSM is connected to: even if the adversary compromises the host machine, the adversary gains only query access to the HSM, not direct access to its internal state. Protecting secret state behind a carefully-designed API enables the HSM to enforce key security properties.

Figure 1-1 shows an example of this setup: a certificate authority using a certificate-signing HSM. The HSM is an independent system—with its own CPU, firmware, RAM, and persistent memory—connected over an I/O interface (UART, in this example) to the host machine. The HSM interacts with the outside world solely through its I/O interface to the host. Figure 1-2 shows the specification of this HSM, which computes ECDSA signatures while protecting the signing key. When a certificate authority uses such an HSM, an adversary that compromises the server cannot extract the signing key because the HSM's API does not expose an operation that returns the key.

As another example: cloud backup systems use HSMs to protect backup encryption keys with users' PINs. Figure 1-3 shows a specification for such an HSM, which defends against

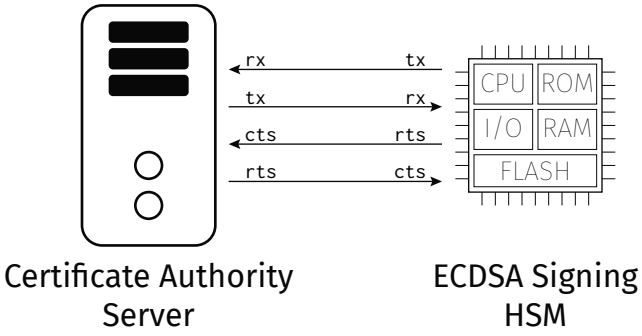


Figure 1-1: An example of a system that uses an HSM: a certificate authority server connected to an ECDSA signing HSM over a UART interface.

```
var prf_key: bytes[16]
var prf_counter: int{0 <= prf_counter < 2^64}
var signing_key: int{0 < signing_key < P256_ORDER}

def initialize(new_prf_key, new_signing_key):
  prf_key = new_prf_key
  prf_counter = 0
  signing_key = new_signing_key

def sign(message):
  if prf_counter == 2^64 - 1:
    return Error
  nonce = hmac_sha256(prf_key, prf_counter)
  prf_counter += 1
  return ecdsa_p256(message, signing_key, nonce)
```

Figure 1-2: Pseudocode specification for the ECDSA signing HSM. The specification exposes an operation to initialize internal state and to sign messages, but it does not expose an operation that returns the signing key.

```
var attempts: int
var pin: bytes[4]
var secret: bytes[32]

def store(new_secret, new_pin):
  secret = new_secret
  pin = new_pin
  attempts = 0

def retrieve(guess):
  if attempts >= 10:
    return Error
  if guess != pin:
    attempts = attempts + 1
    return Error
  attempts = 0
  return secret
```

Figure 1-3: Pseudocode specification for a simplified PIN-backup HSM, which stores a backup encryption key protected by a low-entropy PIN. The HSM provides security against brute-force attacks by enforcing guess limits.

brute-force attacks on the low-entropy PIN by enforcing guess limits for retrieval.

There are billions of deployed HSMs across a variety of server-side and client-side applications. For example, on the server side, certificate authorities like Let's Encrypt use HSMs to store their signing key and sign certificates [1]; cloud providers including Apple, Google, and WhatsApp use HSMs to enforce guess limits for cloud backup keys protected by low-entropy PINs [57, 64, 112]; and credit card networks use HSMs for PIN translation, re-encryption of PIN blocks between nodes in a payment network [48]. On the client side, the iPhone uses its secure enclave processor to safeguard data encryption keys [9], and users rely on USB security keys to protect their authentication keys in the face of a compromised computer [100]. While the term "hardware security module" traditionally refers to devices used on the server side (and often specifically to HSMs implementing the PKCS#11 API [86]), this thesis uses the term to refer to all such special-purpose hardware/software systems that factor out core security functionality.

1.2 Goal: verifying HSMs

Any vulnerability in an HSM's hardware or software can undermine the security of the HSM, and in turn, the security of the overall system. Although HSMs are relatively simple, HSMs have suffered from bugs throughout the hardware/software stack, such as logic bugs, memory corruption, hardware bugs, and timing side channels [18, 30, 33–38, 60, 76, 107, 116–118], motivating the need for a systematic approach to eliminating such bugs.

Mechanized proofs, which connect an implementation to a mathematical specification through a computer-checked proof, have shown promise in eliminating entire classes of bugs in both hardware [31] and software [53, 55, 61, 122]. Empirically, such systems are effective in eliminating bugs in their verified components [45].

The goal of this thesis is to develop a verification approach and framework to verify HSM implementations at the circuit level. HSMs are an ideal proving ground for a new approach to systematically ruling out correctness, security, and leakage bugs because they (1) are widely-used devices for which security is critical, warranting the effort required for formal verification, (2) suffer from security bugs in practice, and (3) are complete but simple enough computer systems to be amenable to an experimental verification effort.

1.3 Challenge: security across levels of abstraction

Systems have large gaps between the specifications of their intended behavior (e.g., tens of lines of code in a high-level language) and their implementations (e.g., thousands of lines of C code and tens of thousands of lines of Verilog code implementing an entire hardware and software system). Moving down the stack, each level of abstraction below the specification—C code, firmware binary, and circuit—introduces additional complexity and potential for bugs and leakage. For example: a C implementation of the ECDSA specification might have an off-by-one error in array indexing, accessing out-of-bounds memory; the compiler might have a bug in an optimization, producing a miscompiled binary from the C implementation of elliptic-curve crypto; or the CPU might have a pipeline hazard bug, causing it to incorrectly execute certain instruction sequences in the compiled binary.

In addition to such correctness bugs, each level of abstraction has the potential to introduce information leakage, including timing side-channel leakage, a class of leakage that has proven particularly hard to eliminate [3, 22, 62, 63, 69, 70, 115]. Examples of information leakage bugs include:

- Error messages: C code for the PIN-backup HSM (figure 1-3) that returns different

error codes for “guess limit exceeded” vs “incorrect PIN”

- Nondeterministic encodings: serialization code in the PIN-backup HSM that leaves uninitialized memory contents in unused bytes in an encoding when serializing a tagged union (e.g., the output `Error` or `secret`) into a fixed-length buffer
- Incorrect commit point: an implementation of the PIN-backup HSM that returns the output before updating persistent memory with `attempts = attempts + 1` (like the bug that affected the iPhone 5s [30])
- Non-constant-time code: a PIN-checking function that leaks how many bytes of a guess match the PIN by using the C `strcmp` function, which returns as soon as it finds a mismatch, leaking information through timing
- Compiler-introduced timing variation: a PIN-checking function that is careful to use bit twiddling and constant-time code at the C code level, but for which a compiler decides to emit a branch at the assembly code level as an optimization, leaking information through timing (like the bug that affected Kyber when compiled with Clang [39])
- Variable-latency instructions: an ECDSA HSM (figure 1-2) that leaks the key by using `mul` instructions to multiply secret data on a processor with a variable-latency multiplier (such as the ARM Cortex-M3 [92]), leaking information through timing

Formally verifying the absence of such bugs that leak information has been particularly challenging. Traditional approaches that focus on verifying correctness at a particular level of abstraction (e.g., C code) have a hard time capturing information leakage, because correctness does not imply non-leakage, and because lower levels of abstraction (e.g., execution at the hardware level) introduce new observables (e.g., cycle-precise timing behavior) that do not even exist at higher levels but can leak information. Even when focusing on a particular level of abstraction such as the circuit level, merely *defining* non-leakage—what it means for the wire-level behavior to leak no more information than a specification—is a challenge.

1.4 State of the art

The security community has long been aware of the importance and practical impact of timing vulnerabilities in cryptography [24, 25], and more broadly, unintended leakage by computer systems. However, prior to this thesis, there were no results verifying non-leakage from an application-level specification down to the cycle-accurate level in hardware.

Some production libraries including BoringSSL, Mbed TLS, and Amazon s2n use approaches based on software testing to validate constant-time execution of cryptographic

routines [15, 46, 59]. Several systems, including some like HACL* that are used in production, provide formal guarantees about side-channel resistance at the software level [6, 21, 122], but the guarantees do not extend down to the hardware level. Recent work on leakage models [7, 8, 28, 54, 78, 110] focuses on either software or hardware but also does not provide end-to-end guarantees. Prior work on verified hardware/software systems [5, 19, 40, 42, 71] focuses on correctness, not security and non-leakage. Chapter 2 elaborates on prior work.

1.5 Threat model

The systems we build consider an adversary that gains direct access to the wire-level digital I/O of the HSM, with the ability to set logic levels on the input wires and read logic levels on the output wires at every cycle. This captures many realistic attacks, such as an adversary that compromises the host machine and is able to send malformed commands or observe all wire-level outputs at every clock cycle. Such an adversary may be able to extract secrets from an HSM, even if that HSM operates correctly when the host machine is well-behaved.

This threat model is focused on remote compromise of the host machine, one of the primary attacks that HSMs aim to defend against. It does not include physical attacks on the HSM: while the threat model includes (digital) timing side channels, it does not include arbitrary side channels [121] such as electromagnetic radiation [4], temperature [58], and power [75].

1.6 Implementation approach

This thesis contributes an HSM architecture that helps developers avoid timing leakage. HSMs following this design run an execution loop that (1) reads a command from the I/O interface, (2) handles the command to produce an in-memory state update and response, (3) updates persistent state atomically, and (4) sends the response over the I/O interface.

Steps (1) and (4) do not compute over the HSM's internal state, so it is easy to avoid secret-dependent timing behavior in this code. Step (3) requires some care to implement atomicity; it involves persisting a fixed-size state buffer to persistent memory, and the developer must write this function to run in constant time.

At the core of the HSM is a `handle` function that implements step (2), implementing command deserialization, core functionality, and response serialization. The developer must write this function to run in constant time (i.e., a constant number of hardware cycles), so that the time it takes to process each command depends only on the type of each

command (e.g., `initialize` or `sign`), not on the internal state of the HSM.

Writing code that executes in constant time at the hardware level requires careful programming at the C code level, a compiler that preserves constant-time behavior, and hardware that executes the code in constant time. As explained in section 1.3, avoiding leakage bugs across levels of abstraction is a challenge, which is why this thesis formally verifies non-leakage at the circuit level.

1.7 Proof approach: information-preserving refinement

This thesis presents *information-preserving refinement (IPR)*, a new formalism that captures security and non-leakage across levels of abstraction, along with *Parfait*, a verification approach for IPR. Parfait verifies the cycle-by-cycle execution of an HSM implementation at the register-transfer level (RTL), with the compiled firmware loaded into the circuit's ROM, proving the absence of bugs across levels of abstraction. Parfait can systematically eliminate leakage bugs in addition to correctness and security bugs in hardware/software systems, capturing and ruling out all of the bugs outlined in section 1.3.

1.7.1 Defining non-leakage

The Parfait approach relates the behavior of the *physical implementation* at the wire-level interface—the ground truth of what the host machine controls and observes at the digital level, which captures timing channels at a cycle-accurate level—to a *functional specification* of the methods that the HSM exposes. Figure 1-1 shows an example of a physical implementation: the host connects to this HSM via two input wires and two output wires, which the host can read/write at every cycle. Figure 1-2 shows the functional specification for this HSM. It exposes two operations, `initialize` and `sign`. The specification ensures unique nonces across operations, prevents nonce reuse, and does not expose an operation for reading back the PRF key or signing key.

Parfait relates a physical implementation to a functional specification with a new definition called *information-preserving refinement (IPR)*, illustrated in figure 1-4. IPR is inspired by formalizations of zero knowledge in cryptography [51, 52] and uses the real/ideal paradigm to define security.

IPR defines a *real world* that models the host's view of the HSM in the real world under the Parfait threat model, and it defines an *ideal world* that is as abstract as the specification, implicitly capturing security guarantees. The IPR definition states that the real world and ideal world must be observationally equivalent, capturing the notion that the imple-

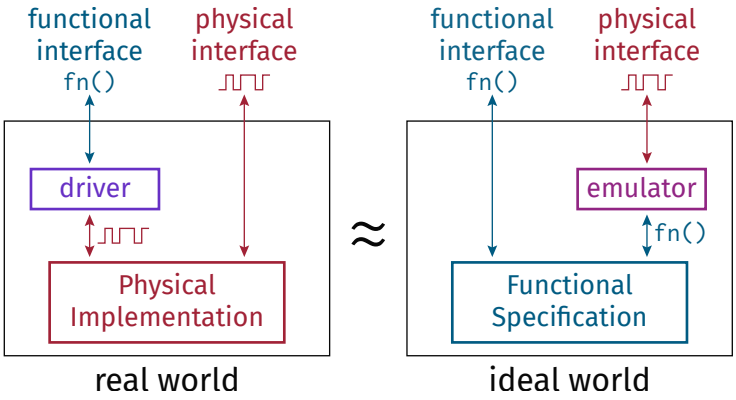


Figure 1-4: An illustration of the definition of IPR, an equivalence between a real world and an ideal world, applied to HSMs.

mentation implements the spec and that its wire-level I/O behavior leaks no additional information.

In IPR, a *driver* describes the I/O protocol that a host machine can follow to get correct results from the HSM, describing how each spec-level operation is implemented in terms of wire-level I/O with the HSM. The driver is a part of the specification. Its dual, an *emulator*, is a proof artifact that describes how wire-level behavior can be explained in terms of spec-level operations. An emulator has no access to a specification's internal state, but it must mimic the implementation's behavior at the wire level with only query access to the specification. The existence of an emulator shows that no matter what wire-level inputs are given to the device (including inputs that violate the I/O protocol), the HSM's wire-level behavior reveals no more information than the specification.

If an implementation leaks more information than the specification (e.g., has a timing side channel), an emulator satisfying the IPR definition does not exist. For example, suppose that a leaky implementation of the PIN-backup HSM (figure 1-3) used the C `strcmp` function to compare a guess against the stored PIN. In this case, a wire-level input corresponding to a guess `1234` against a stored PIN `1337` (where 1 byte matches) would return a different (delayed) wire-level output compared to the guess `0000` (where 0 bytes match). There does not exist an emulator that matches the implementation behavior in both cases: the emulator can query the specification to determine that the spec-level output is `Error`, but it cannot determine the output timing to match the implementation's wire-level behavior in both cases. A secure implementation would return an error in a constant number of cycles, and an emulator would be able to match this behavior by querying the specification, waiting for the (constant) number of cycles it takes an implementation to produce an output and then sending the output over its physical interface.

1.7.2 Proving non-leakage

Parfait leverages the transitivity of IPR to enable a modular proof approach, separating verification of software and hardware and making verification manageable. Parfait introduces a new HSM software architecture that enables modular proofs, an approach to verifying IPR for HSMs that follow this design, frameworks to support mechanical proofs for software and hardware, and a formalized theory of IPR that ties everything together. Figure 1-5 summarizes the developer workflow for implementing and verifying an HSM with Parfait.

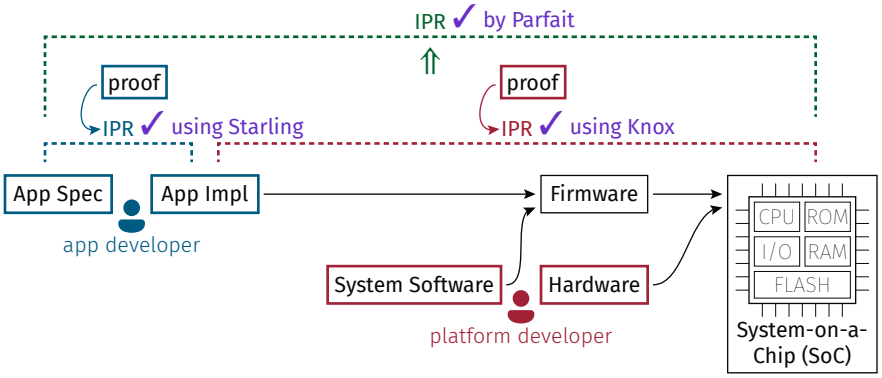


Figure 1-5: The Parfait developer workflow. The app developer writes the components in blue, and the platform developer writes the components in red. The developer uses tools and frameworks provided by Parfait to verify IPR for software and hardware. Parfait provides a verified IPR theory that ties together the software and hardware proofs, imposing no burden on the developer.

The application developer writes: (1) a top-level functional specification App Spec, (2) an application software implementation App Impl (at roughly the C code level), and (3) a computer-checked proof of IPR between the app spec and the app implementation, supported by Parfait's *Starling* framework for verifying IPR for software. The platform developer writes: (1) System Software such as boot code and peripheral drivers, (2) the Hardware implementation, and (3) a computer-checked proof of IPR between the app implementation and the complete system-on-a-chip (SoC), supported by Parfait's *Knox* framework for verifying IPR for hardware. Formally-verified theory supplied by Parfait ties these proofs together to yield the final IPR between the specification and the SoC.

1.8 Contributions

This thesis contributes:

A new formalism for modeling and reasoning about leakage. *Information-preserving refinement (IPR)* relates implementations to specifications and captures the notion that the implementation leaks no more information than the specification. Using the Coq proof assistant [103], this thesis formalizes the *theory of IPR*, including four verified proof strategies for IPR used in different parts of Parfait.

A framework for verifying non-leakage for hardware security modules. The *Parfait* framework introduces a modular approach to proving IPR for HSMs, enabled by a new software architecture. Parfait consists of two components that implement the proof strategies verified in Coq: the *Starling* framework, built on top of the F* [102] proof-oriented programming language, for reasoning about software; and the *Knox* framework, built on top of the Rosette [104] symbolic execution library, for reasoning about hardware. The Knox framework introduces two artifacts that are usable outside the context of Parfait: the *Rosys* Verilog-to-Rosette toolchain and the *Riscette* executable Rosette semantics for CompCert RISC-V assembly.

Verified HSMs. To evaluate Parfait, this thesis implements and verifies several HSMs. This includes an ECDSA signing HSM, where Parfait proves that an implementation, written in 2,300 lines of code and 13,500 lines of Verilog, satisfies a specification written in 40 lines of code.

1.8.1 Limitations

The techniques and tools developed in this thesis have limitations, described in more detail in the relevant chapters. Some examples include: Parfait uses several proof tools (Coq, F*, and Rosette) without a mechanized connection between them (section 3.3); IPR as formalized in this thesis does not support specifications with randomness (section 4.4); Parfait HSMs use relatively simple CPU designs used in simpler HSMs today, and supporting high-performance processors like out-of-order Intel x86 CPUs will likely require additional ideas and techniques (section 10.2).

1.8.2 Open-source software

We have open-sourced all software developed as part of this thesis. See the project page at anish.io/parfait for links to all of the software, or see table 1-1 below.

Table 1-1: Open-source software released as part of Parfait.

github.com/anishathalye/ipr	Coq formalization of IPR theory
github.com/anishathalye/starling	Starling: F*-based software verification framework
github.com/anishathalye/knox	Knox: Rosette-based hardware verification framework, including the Rosys Verilog-to-Rosette toolchain and the Riscette executable semantics for CompCert RISC-V assembly
github.com/anishathalye/parfait-hsm	HSMs verified with Parfait
github.com/anishathalye/knox-hsm	Simpler HSMs verified directly with Knox

1.9 Prior publications

This thesis expands upon work covered in three peer-reviewed publications:

- **Notary (SOSP 2019)**: introduces the *Rosys* Verilog-to-Rosette toolchain.

  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 (SOSP)*. Huntsville, Ontario, Canada, Oct. 2019, pages 97–113.

- **Knox (OSDI 2022)**: introduces the definition of *information-preserving refinement (IPR)* and the *Knox* framework.

  Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Hardware Security Modules with Information-Preserving Refinement". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, July 2022, pages 503–519.

- **Parfait (SOSP 2024)**: introduces the mechanization of IPR, the *Parfait* HSM design and modular verification approach, *Starling* framework, *Riscette* semantics, and an extension of Knox to support assembly-level specifications.

  Anish Athalye, Henry Corrigan-Gibbs, M. 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 30th ACM Symposium on Operating Systems Principles (SOSP)*. Austin, TX, Nov. 2024.

Chapter 2

Related work

This thesis develops a new approach for verifying non-leakage across levels of abstraction that can capture leakage-freedom when "leakage" is not separate from "output," unlike prior approaches that rely on leakage models (section 2.1) and noninterference (section 2.2). The definition of information-preserving refinement (IPR) is inspired by simulation-based definitions of security in cryptography (section 2.3). Parfait verifies HSMs end-to-end from specification to register-transfer level (RTL), but unlike prior work on hardware/software verification (section 2.4), Parfait proves a non-leakage property in addition to functional correctness. Parfait uses translation validation (section 2.5) as a technique to reason about correctness and non-leakage at the hardware level while sidestepping the need to directly prove the hardware correct. The IPR definition bears some resemblance to the properties secure compilers guarantee about their compilation results (section 2.6). The Parfait approach targets systems running a single application; other work addresses the problem of isolating multiple applications running on a single machine (section 2.7).

2.1 Leakage models

Leakage models are a standard technique used to reason about timing side-channel leakage. A leakage model is an assumption about what an adversary can observe, informed by a threat model. For example, a leakage model might assume that an adversary can observe a program's control flow and addresses of memory accesses (and nothing else); this might be derived from a threat model that assumes the existence of a cache side channel and an adversary co-located on the same machine. Approaches using leakage models commonly define non-leakage as adversary observations being independent of secrets, formalized using noninterference (section 2.2).

Constant-time programming. Timing side-channel leakage [63] is particularly devastating in cryptographic libraries because it is practically exploitable, even over the network [24, 25]. For this reason, most cryptographic libraries include timing side-channel leakage within their threat model and attempt to mitigate it by following a *constant-time programming* discipline. For example, an implementation will assume a leakage model that includes control flow and memory accesses, and to prevent leakage under this model, avoid branching on secrets or accessing memory locations where the address depends on a secret.

Several production cryptography libraries including BoringSSL, Mbed TLS, and Amazon s2n assume various leakage models and include constant-time testing in continuous integration [15, 59]. Some perform testing without soundness guarantees [56, 65], while others use tools based on formal methods with soundness guarantees [8, 15, 20, 111].

Several formally verified cryptographic libraries [16], where the focus is on verifying functional correctness, also reason about timing leakage. Fiat Crypto [41] does not explicitly specify a leakage model but is secure under the assumption that control flow (and nothing else) leaks. Jasmin [6] assumes that control flow and memory read/write addresses (and nothing else) leak. HACL* [122], ValeCrypt [21, 47], and EverCrypt [93] assume that control flow, memory read/write addresses, and inputs to variable-latency instructions (and nothing else) leak.

All of these works protect only against one specific type of leakage—that specified by their leakage model—so their soundness depends on whether the leakage model is accurate. For example: HACL* assumes constant-time word-level multiplication, but this guarantee is not provided by certain processors such as the ARM Cortex-M3 [10, 92]; and EverCrypt proves that secrets never influence branches, addresses of memory accesses, or inputs to instructions assumed to be timing-dependent (e.g., division), but it does not provably eliminate other leakage such as that related to speculative execution.

In contrast, Parfait assumes that the adversary can observe (and *control*) the wire-level signals on the I/O interface to the HSM at a cycle-precise level, and it proves that the wire-level behavior of the circuit at the register-transfer level (RTL) does not leak more information than the specification.

Constant-time-preserving compilation. Constant-time programming can avoid certain classes of side-channel leakage, but any guarantees it provides can be undone by a compiler that does not preserve constant-time behavior. Standard compiler correctness as proven for compilers like CompCert [67] does not automatically provide such a guarantee. CompCert-CT [17] defines C-level and assembly-level leakage models, modifies CompCert to eliminate sources of potential leakage, and proves that the modified compiler preserves constant-time

behavior of programs with respect to these leakage models.

Leakage models for hardware. Recent hardware-focused work has proposed leakage models that more accurately model modern hardware than the simpler leakage models assumed by the cryptography libraries described above; in particular, these leakage models focus on capturing leakage from speculative execution [28, 54, 78]. Validating hardware against leakage models through techniques like fuzzing [23, 26, 87, 88, 105] has revealed gaps between leakage models and hardware implementations.

LeaVe [110] formally verifies a processor's RTL against a leakage contract. LeaVe has verified several simple RISC-V processors, including variants of the Ibex processor used in Parfait case studies. To simplify verification, LeaVe assumes functional correctness of the processor. In contrast, Parfait does not need to assume functional correctness; instead, Parfait verifies correctness of the application's execution on the processor as part of proving IPR.

2.2 Noninterference

Noninterference [50] captures confidentiality properties in systems where high-sensitivity inputs should not affect low-sensitivity outputs, which are separate from high-sensitivity outputs. A range of formal methods have been developed for proving noninterference and analyzing information flow [96, 109]. Constant-time cryptography (section 2.1) formalizes freedom from timing side channels as noninterference by defining a leakage trace (the low-sensitivity output) that captures adversary observations, such as every program counter value, and proving that any two executions that have matching public inputs but differing secrets (high-sensitivity inputs) produce identical leakage traces.

Noninterference with declassification [82] separates low and high-sensitivity inputs (i.e., public and secret inputs) and supports controlled influence of secrets on outputs through an explicit declassify function that marks secret-dependent values as safe to output. Ironclad [55] uses this style of security definition; the proofs cover only software, not hardware, and do not rule out timing side channels.

The Parfait setting does not have separate low/high-sensitivity inputs/outputs. There is just one input, the logic levels on the input wires at every cycle, which the adversary can control upon compromise of the host machine. Similarly, there is just one output, the logic levels on the output wires at every cycle, and this is what the adversary observes upon compromise of the host.

Noninterference does not apply in the Parfait setting; the output can and will be secret-

dependent. For example, the wire-level output of the ECDSA HSM (figure 1-2) depends on the signing key. Instead, IPR says that the output does not leak more information than the specification, which is not a noninterference property. Similarly, noninterference with declassification does not apply in the Parfait setting. Instead, IPR says that after the HSM receives inputs from the driver (i.e., corresponding to a spec-level operation), its future behavior does not leak more information than the specification, which is not a declassification property.

2.3 Simulation-based definitions of security

The definition of information-preserving refinement (IPR) is inspired by the real/ideal paradigm and simulation-based definitions of security for cryptographic zero knowledge, multiparty computation (MPC), and universal composability [27, 51, 52, 68]. The emulator in IPR is similar to the simulator in MPC, which formalizes the notion of zero knowledge in an MPC protocol. Parfait uses this concept to define non-leakage for a hardware/software system rather than a cryptographic protocol. Among other differences, simulation-based definitions compare computationally-indistinguishable probability distributions, while IPR is formalized as an exact input-output trace equivalence between the real world and ideal world state machines.

2.4 Hardware/software verification

The CLI stack [19], Verisoft project [5], CakeML verified stack [71], and Bedrock2 lightbulb and garage door [40, 42] verify functional correctness properties for hardware/software systems, with an emphasis on modular verification. In contrast to Parfait, proofs for these systems only establish functional correctness and do not rule out information leakage.

2.5 Translation validation

Translation validation is an approach used to verify that a transformation of a program, typically from a high-level programming language to a lower-level representation like machine code, refines the source [83, 91, 97, 106]. Translation validation is applied to check the translation of a *particular* program; the technique is an alternative to proving that the transformation (e.g., compilation) is correct for all inputs.

Parfait uses translation validation to verify IPR for some of the levels of abstraction in the Parfait stack: Parfait does not verify the correctness of the CPU and system-on-a-chip

(SoC) in general, or prove that a compiler always preserves non-leakage. Instead, Parfait proves that a particular program satisfies IPR when executed on a specific hardware device.

Most prior uses of translation validation have focused on showing refinements for functional correctness and have stopped validation at the assembly level; Parfait additionally validates non-leakage as defined by IPR and validates execution down to the level of hardware.

2.6 Secure compilation

In Parfait, the circuit cannot be derived from the specification via compilation, but the IPR definition bears some resemblance to the properties secure compilers guarantee about their compilation results. Fully-abstract compilers [2] preserve and reflect observational equivalence from the source to the target language. Some security properties can be stated as program equivalences [89], but IPR's non-leakage property is not captured by this type of definition. In fact, some Parfait specifications such as the ECDSA HSM (figure 1-2) have no instances that are observationally (extensionally) equivalent but not intensionally equal, so a secure-compilation-style equivalence preservation at the circuit level would be vacuous. Trace-preserving compilation [90] preserves trace equivalence between source and target and handles invalid target-level inputs. The definition is not general enough to apply to the HSM setting because source-level inputs don't map to single target-level inputs (function call to wire input for a *single* cycle), and there is no notion of "ignoring invalid inputs" (for any wire-level inputs, the HSM will have wire-level outputs). Furthermore, similar to the case of program equivalence, some Parfait specifications such as the ECDSA HSM have no instances that are trace-equivalent but not equal, so trace-equivalence preservation at the circuit level would be vacuous.

2.7 Process isolation

Parfait HSMs run a single application, and verification ensures that the software or hardware doesn't leak information to the outside world. Other work addresses the problem of leakage or interference between different processes colocated on the same machine [84].

The seL4 kernel [80, 81], mCertiKOS hypervisor [32], Komodo enclave monitor [44], and Nickel information flow control framework [98] have proved noninterference and other information flow properties between processes; these proofs do not cover leakage through microarchitectural state or timing. Ge et al. [49] extend seL4 with mechanisms to prevent microarchitectural leakage between security domains by, among other techniques, using

instructions to reset microarchitectural state on domain switches. Sison et al. [99] have formalized the security guarantees provided by this approach under an abstract model of OS and hardware behavior.

Process isolation (whether or not it takes into account microarchitectural side channels) is different from Parfait's goals: a Parfait HSM runs just a single application, and it must leak no more information than its specification allows; this is not an isolation/noninterference-style property. Even if it did operate in a multi-process setting, an HSM application that leaks its private key through timing (or just directly sends it out over the I/O interface) to the outside world could be strongly isolated from other processes on the same machine, but such a system is clearly insecure.

Chapter 3

Overview

This chapter presents an overview of information-preserving refinement (section 3.1) and the Parfait approach for verifying HSMs with IPR (section 3.2).

3.1 Information-preserving refinement

Information-preserving refinement is a new formalism for defining non-leakage. IPR relates two state machines with differing interfaces and states that they are equivalent modulo interface differences, including capturing that one leaks no more information than the other. Parfait applies IPR to reason about HSM correctness and non-leakage by treating both HSM specifications and their circuit-level implementations as state machines and proving IPR between them. This section presents an informal overview of IPR, and Chapter 4 provides the formal definition.

3.1.1 State machines

A state machine M is a 6-tuple (S, s_0, I, O, T, ψ) consisting of:

- A set of states S
- An initial state $s_0 \in S$
- A set of inputs I
- A set of outputs O
- A transition relation $T \subseteq (S \times I) \times (S \times O)$, relating a state and an input to a new state and an output (this relation is often a total function)
- A reset function $\psi : S \to 2^S$, mapping a state to all possible states it can reset to

We define state machine equivalence between two state machines M_1 and M_2 that have identical input/output types, $M_1 \approx M_2$, as trace equivalence (that their I/O behaviors are identical).

Functional specifications. Developers write Parfait specifications as state machines that describe the intended input-output behavior of a system at the level of function calls and return values. Figure 3-1 shows an example, the transition function `step` from the specification of the ECDSA signing HSM, written in explicit state machine style in the F* proof-oriented programming language [102].

```
let step (st:state_t) (cmd:command_t) : state_t & response_t =
  match cmd with
  | Initialize prf_key signing_key ->
    { prf_key = prf_key; prf_counter = uint 0; signing_key = signing_key },
    Initialized

  | Sign msg ->
    if uint_v st.prf_counter < maxint U64 then
      let data = uint_to_bytes_be st.prf_counter in
      let nonce = hmac SHA2_256 st.prf_key data in
      let sig = ecdsa_signature_agile NoHash _ msg st.signing_key nonce in
      { st with prf_counter = incr st.prf_counter }, Signature sig
    else
      st, Signature None
```

Figure 3-1: The transition function from the functional specification of the ECDSA signing HSM, written in F*, corresponding to the pseudocode in figure 1-2. The definitions of `hmac` and `ecdsa_signature_agile` are used directly from HACL*, a verified cryptography library.

The specification additionally defines the state type (`state_t`), initial state, and input/output types (`command_t` and `response_t`); specifications don't have a reset behavior; figure A-1 contains the complete specification. Developers write specifications in Parfait as *whole-command state machines*, with no notion of timing—the interaction model is that when the client of the state machine invokes an operation (e.g., `Sign`), the entire operation is a single atomic step of the state machine, and there are no observables aside from the return value.

The specification state machine describes how the implementation must behave, and it implicitly describes what is and isn't allowed to leak. For example, the specification for

the ECDSA signing HSM ensures unique nonces across operations, and it doesn't support reading out the signing key or PRF key.

Physical implementations. Like specifications, implementations in Parfait are modeled as state machines, but at the cycle-precise wire-I/O level, so that they capture an implementation's behavior as well as what would otherwise be described as "leakage" or "side channels," according to our threat model. This captures the ground truth of what the host machine can influence and observe when interacting with the HSM within Parfait's threat model.

Figure 1-1 shows an illustration of the ECDSA signing HSM and its interface. Interpreted as a state machine, an HSM implementation's state consists of all registers and memories in the device, its input is the values on the input wires (`rx` and `cts` in this example), and its output is the values on the output wires (`tx` and `rts` in this example). Its step function is given by the circuit, describing HSM execution for a single clock cycle. Its reset behavior havocs all volatile registers and memories in the circuit, and asserts the reset line.

This provides a unified way of reasoning about leakage: there is no more separation of "outputs" from assumed "side-channel leakage" as is done in other approaches like noninterference (section 2.2). Parfait models the implementation at a level that describes everything that a (possibly malicious) host can influence and observe under our threat model—the wire-level I/O behavior—and Parfait proves a correspondence between the implementation and the specification that prevents leakage beyond what is allowed by the spec, as described next in section 3.1.2.

This way of thinking is applicable to reasoning about leakage in general, not just for timing side channels and hardware/software systems like that in figure 1-1. For example, if we wanted to rule out leakage only in software (e.g., through leaky serialization [95]), we could model an implementation at the level of C code with entire function calls as atomic state machine steps.

3.1.2 Defining non-leakage

The goal of information-preserving refinement is to define what it means for an implementation state machine M_i with an low-level interface I_i / O_i to implement a specification state machine M_s with a high-level interface I_s / O_s and leak no additional information. IPR achieves this by defining a correspondence between implementation and specification that relates the two in terms of both the high-level interface (e.g., spec-level operations) and the low-level interface (e.g., wire-level I/O). Illustrated in figure 3-2, IPR is defined as

an observational equivalence between two worlds: the real world, and an ideal world that implicitly captures security guarantees.

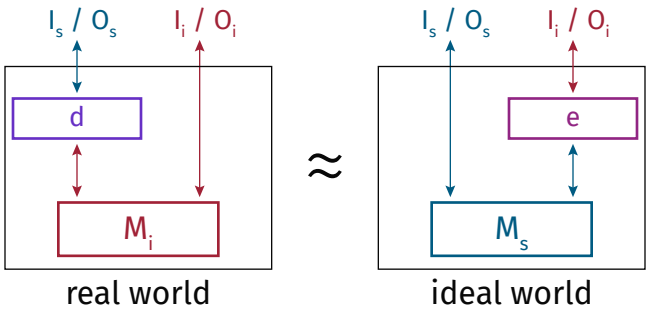


Figure 3-2: The definition of IPR: an implementation M_i is an information-preserving refinement of a specification M_s with respect to a driver d if there exists an emulator e such that the real world is observationally equivalent to the ideal world.

Real world. The real world is set up to have the client interact with the implementation M_i. In the real world, a program called the *driver* describes how high-level operations are implemented in terms of low-level interaction with the implementation. Specifically, a driver is a program that takes a high-level input of type I_s, interacts with the implementation by making (any number of) queries of type I_i and receiving responses of type O_i, and finally produces a high-level output of type O_s. In the real world, the client interacts with the same underlying implementation either directly over the low-level interface or indirectly over the high-level interface through the driver.

In the context of HSMs, the real world models the host machine connected to the actual HSM implementation. The host can take a physical view of the device and directly perform arbitrary wire-level I/O (reading and writing the I/O pins at every cycle). The host can also take a functional view of the device and follow the HSM's I/O protocol, which is described by the driver, akin to a device driver in an operating system. The driver translates spec-level operations to wire-level I/O, describing how the host invokes the operation and computes the return value by interacting with the HSM over its wire-level interface.

The host can switch freely between the functional view and the physical view at any time. Switching from the functional view to the physical view models compromise of the host machine; switching from the physical view back to the functional view models recovery (for example, by unplugging the device and moving it to an uncompromised machine).

Ideal world. The ideal world is set up to provide the same interface as the real world but is as abstract as the specification, implicitly capturing security guarantees. In the ideal world, the client interacts with the specification M_s. Because the ideal world also needs

to expose a low-level interface to match the interface exposed by the real world, the ideal world contains an *emulator*, a dual of the driver, that is a program that describes how low-level behavior can be obtained from high-level operations. Specifically, an emulator is a program that takes a low-level input of type I_i, interacts with the specification by making (any number of) queries of type I_s and receiving responses of type O_s, and finally produces a low-level output of type O_i. This ideal world implicitly captures security: it merely offers an interface, directly, or indirectly through the emulator, to the specification.

In the context of HSMs, when the host takes a functional view of the device, operations are invoked directly on the specification, so the behavior is correct and secure by definition. Under the functional view, spec-level operations are not seen by the emulator. When the host takes a physical view of the device, the wire-level I/O behavior it observes is produced by an emulator that only has query access to the specification, so the physical interface leaks no more information than the specification exposes through its API. Furthermore, when the host switches back to the functional view of the device, it continues interacting with the same specification that was queried by the emulator, so the effect of any queries made by the emulator in order to produce wire-level outputs is present in the specification state. In the ideal world, any execution, no matter how it switches between functional and physical interfaces, corresponds to some sequence of operations invoked on the specification. The ideal world can be instantiated with *any* emulator, and it remains secure.

Equivalence. We say that an implementation M_i is an information-preserving refinement of a specification M_s with respect to a driver d if there exists an emulator e such that the real world and ideal world (as defined above, and illustrated in figure 3-2) are observationally equivalent.

The real world and ideal world themselves can be seen as state machines, and each exposes both a low-level and high-level interface, so the state machines' input is of type $I_i + I_s$ and the output is of type $O_i + O_s$. We define observational equivalence between the IPR worlds as state machine equivalence ($\approx$), which applies to machines with matching interfaces. The IPR definition is, in other words:

$$M_i \approx_{IPR[d]} M_s \quad := \quad \exists e, \textit{real\_world}\,(M_i, d) \approx \textit{ideal\_world}\,(M_s, e)$$

3.2 Verifying IPR with Parfait

Figure 3-3 gives an overview of the Parfait verification approach and developer workflow. Parfait leverages transitivity to provide a modular proof approach for IPR.

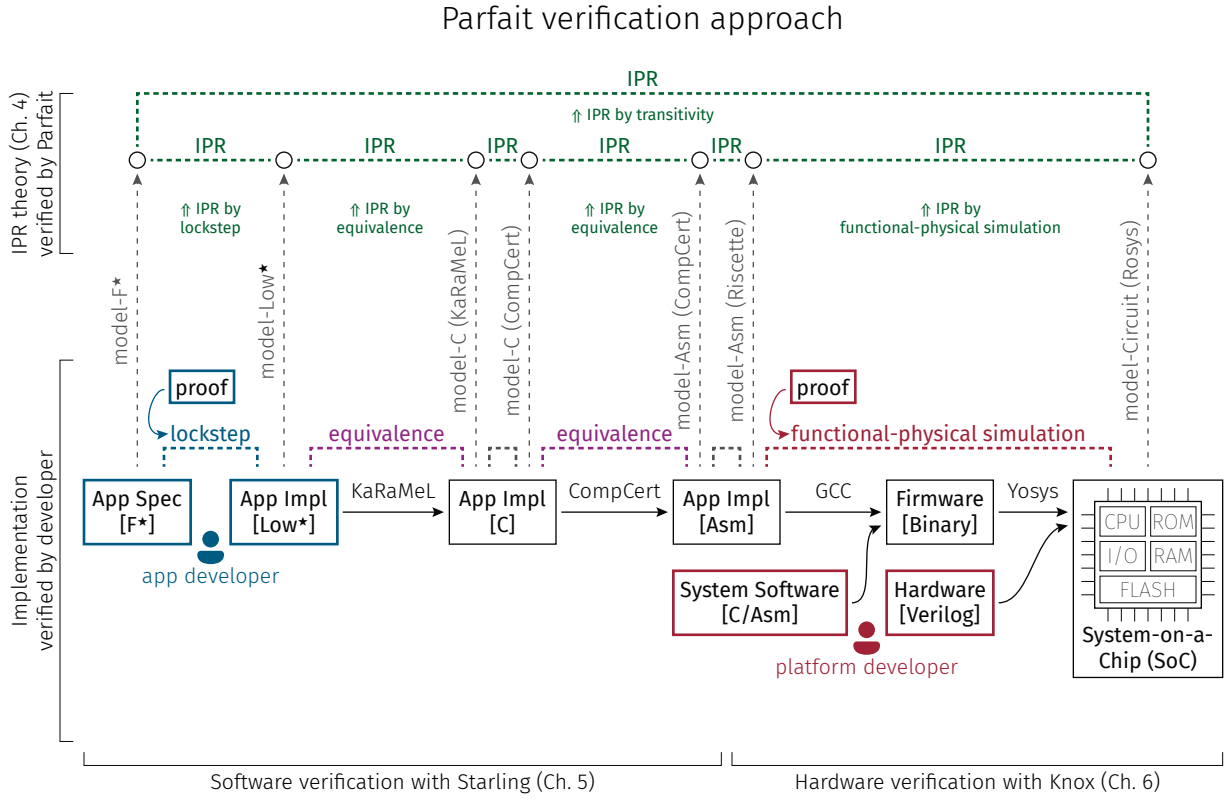


Figure 3-3: The Parfait verification approach. The app developer writes the components and proof in blue (verified with the Starling framework in F* [102], chapter 5), off-the-shelf verified compilers provide the proofs in magenta, and the platform developer writes the components and proof in red (verified with the Knox framework in Rosette [104], chapter 6). The Parfait framework provides a theory of IPR (in green, verified in Coq [103], chapter 4) that is verified once-and-for-all. A metalogical argument (in gray) connects the mechanized proofs done by the HSM developer to the mechanized theory of IPR provided by Parfait to prove a top-level theorem of IPR between the App Spec and the System-on-a-Chip (SoC).

3.2.1 Developer workflow

Parfait supports two largely independent developers: an app developer who writes the application and a platform developer who provides the system software (for persistence, peripheral I/O, etc.) and hardware for running the application.

App development. The app developer writes an implementation of the application functionality in Low* [94] (the App Impl [Low*] in figure 3-3), a C-like language for low-level programming embedded in F*. This includes both the application logic as well as code to decode incoming requests and encode the responses. Specifically, the app developer implements the `handle` function referenced in figure 3-4, which operates on in-memory command and response buffers. Off-the-shelf verified compilers turn this code into an application binary (the App Impl [Asm] in figure 3-3).

```
1  uint8_t state[STATE_SIZE];
2  uint8_t cmd[COMMAND_SIZE];
3  uint8_t resp[RESPONSE_SIZE];
4
5  void main() {
6    while (1) {
7      read_command(&cmd);          // read command from I/O interface
8      load_state(&state);          // load state from persistent memory
9      handle(&state, &cmd, &resp); // do core computation
10     store_state(&state);         // atomically persist state
11     write_response(&resp);       // write response to I/O interface
12   }
13 }
```

Figure 3-4: System software for the main loop of the HSM, invoking the `handle` function implemented by the app developer. The system software implements the I/O functions (`read_command` and `write_response`) and crash-safe persistence using journaling (`load_state` and `store_state`).

Platform development. The platform developer writes a system-software library, which implements the system's overall execution loop and includes everything in the firmware image except the implementation of the `handle` function. The system software includes startup code written in assembly to boot the processor and set up the environment for executing C code, the code shown in figure 3-4, and the implementations of `read_command`,

etc. The platform developer then links this code with the application code (the implementation of `handle`) from the app developer; the resulting linked binary is the HSM's firmware (the Firmware in figure 3-3). The platform developer then implements the Hardware in Verilog and embeds the HSM's firmware in the hardware's ROM.

The result is a complete System-on-a-Chip (SoC). A user can fabricate it directly or put it onto an FPGA and run it as a hardware implementation of the HSM app.

3.2.2 Proof approach: transitive IPR

Parfait's approach to proving IPR between the specification and the SoC is to introduce several intermediate levels of abstraction, prove IPR between levels, and use the transitivity of IPR to obtain a top-level theorem relating specification to implementation, as illustrated in figure 3-3. Parfait relies on the developer to write mechanized proofs about software and hardware (using F* [102], in blue, and using Rosette [104], in red), with the support of the Starling and Knox frameworks. Parfait uses verified compilers (in magenta), and Parfait provides a mechanized theory of IPR (in Coq [103], in green) that is verified once-and-for-all. A metalogical argument (in gray) ties these together to yield the top-level proof of IPR between specification and implementation.

Levels of abstraction. Table 3-1 shows the five *levels of abstraction* used as part of a Parfait HSM's overall proof, corresponding to the five artifacts shown in figure 3-3. As illustrated in figure 3-3, each of these levels can be *modeled* as (i.e., interpreted as) state machines in the theory of IPR.

Table 3-1: An overview of the levels of abstraction used in Parfait's proof. This table shows the state, input/output types, and transition functions for the levels when modeled as state machines in the theory of IPR. The specification defines its own types for the state, input, and output.

Level of abstraction	State	Input / Output	Transition
App Spec [F*]	`state_t`	`command_t` / `response_t`	`step()` call
App Impl [Low*]	Bytes	Bytes	`handle()` call
App Impl [C]	Bytes	Bytes	`handle()` call
App Impl [Asm]	Bytes	Bytes	`handle()` call
System-on-a-Chip	Registers / Memories	Wires	Cycle step

Parfait's overall approach involves proving IPR between each level of abstraction. As the rest of this thesis describes, this decomposition enables Parfait to leverage the best tools for each refinement, reducing the overall proof effort and reusing existing proofs. At the top

level is the app-developer-supplied application spec, such as figure 3-1. The second level is the app implementation, which implements the core application logic and is written in Low*, operating on machine integers, buffers, etc. In the ECDSA signing HSM, this code is where the app developer represents bignums as arrays of limbs, implements performance optimizations such as Montgomery multiplication, and so on. The third and fourth levels are compiled versions of the implementation: a C program and an abstract assembly program (a precursor to the final `.s` file). The final level is the complete SoC (including the firmware image in its ROM), with hardware execution modeled at the cycle-precise level. The first four levels are whole-command state machines, where the execution of an entire operation is a single step. The last level introduces cycle-precise timing.

Between each of these levels of abstraction, a *driver* (section 3.1.2) describes how inputs/outputs at the higher level of abstraction (e.g., App-level inputs/outputs) map to I/O at the lower level of abstraction (e.g., bytes). The drivers between the intermediate levels (Low* to C, and C to Asm) are identity drivers. The driver for the spec level describes how commands are encoded as bytes and responses are decoded from bytes, and the driver for the SoC level describes how byte-level commands are sent to the device over the wire, and how byte-level responses are read from the device over the wire. The top-level driver, between App Spec and SoC, is a composition of all the drivers between levels of abstraction: it describes how spec-level operations translate to wire-level I/O.

IPR drivers model the behavior of a well-behaved host machine: the drivers are a part of the specification. The HSM implementation's I/O peripheral driver, a part of its system software, is unrelated to the IPR drivers: all of the implementation's software, including this driver code, is covered by verification.

IPR proof approaches. Parfait formalizes the theory of IPR along with four proof techniques for IPR in chapter 4.

IPR by transitivity is a central technique in Parfait used to obtain a top-level theorem relating the App Spec to the SoC.

IPR by lockstep applies when two state machines have differing input/output types but there is a one-to-one correspondence between the steps of the spec and implementation state machine. This is the case between the first two levels of abstraction: the F* App Spec operates at the level of abstract app commands/responses (F* data types `command_t` and `response_t`), and the implementation operates on buffers of bytes, but a single step of the spec state machine corresponds to a single step of the implementation state machine (a single invocation of `handle`). A set of conditions we call *lockstep* is sufficient to prove IPR in this case. This technique does not require the developer to supply an emulator; instead, the

developer only needs to supply encode/decode functions that convert between spec-level and implementation-level inputs and outputs.

IPR by equivalence applies when two state machines have identical input/output types and are behaviorally equivalent. This applies when using verified compilers, where the state machines given by the corresponding models are equivalent: behavioral *equivalence* implies IPR. This technique does not require the developer to supply an emulator; the state machines are related by the identity emulator.

IPR by functional-physical simulation is a generalization of forward simulation [74] to the IPR setting, which applies when there is a *functional-physical simulation relation* that holds between high-level operations or sequences of low-level operations. The existence of such a relation implies IPR. This technique requires the developer to supply an emulator (section 6.3 describes a strategy for constructing emulators for the circuit level of abstraction).

These four proof techniques for IPR are verified once-and-for-all by the Parfait framework using the Coq proof assistant.

Software and hardware proofs. The app developer starts by writing a top-level application specification in a functional style, represented by the App Spec in figure 3-3, such as the ECDSA spec (figure 3-1). Application specifications describe input-output behavior with no notions of data encodings, wire-level signals, or timing behavior. Next, the developer uses the *Starling* framework to prove *lockstep* between the specification and the app implementation. Starling encodes the lockstep property as a standard Hoare logic precondition/postcondition for the Low* `handle` function, which allows the developer to reuse existing verified software; for example, in our prototype, we build on the HACL* verified cryptography library, including reusing its proofs. From the Low* implementation, Parfait uses verified compilers (KaRaMeL [94] and CompCert [67]) to produce an assembly implementation that is behaviorally equivalent to the Low* code, so *equivalence* holds between the interpretations of the levels as state machines. Chapter 5 describes this software verification workflow in detail.

Next, the platform developer proves that executing the final SoC, with the binary firmware image embedded as the ROM contents, securely implements the assembly, using the *Knox* framework to prove the *functional-physical simulation* property. Chapter 6 describes this hardware verification workflow in detail.

Metalogical arguments. Finally, Parfait combines these proofs together into a single end-to-end proof showing that the SoC securely implements the App Spec. This ensures there is

no leakage by the implementation—be it encoding bugs, compiler bugs, or timing bugs in the CPU hardware. To combine proofs together in a sound way, Parfait *model*s each level of abstraction to a state machine in the language of information-preserving refinement (IPR) and uses the verified proof strategies, including the transitivity of IPR, to prove the top-level IPR between App Spec and SoC.

Each level (e.g., Asm code) has an interpretation as a state machine in the formalization of IPR. This is the connection to the theory mechanized in Coq. For example, figure 5-3 shows how the Asm code is interpreted as a state machine. This modeling is metalogical (i.e., an on-paper argument rather than a machine-checked proof), as is the connection between (1) Starling's encoding of lockstep in F* and the Coq definition of lockstep, and (2) Knox's encoding of functional-physical simulation in Rosette and the Coq definition of functional-physical simulation.

For compiling Low* to C, Parfait uses KaRaMeL; compiler correctness implies that interpretations of the Low* and C as whole-command state machines are equivalent state machines (section 5.3), a metalogical argument that connects to the Coq proof that equivalence implies IPR. KaRaMeL has a semantics for the C target, and CompCert has a semantics for its C source; Parfait requires that the semantics align, another metalogical argument (section 5.5).

For compiling C to Asm, Parfait uses CompCert; its (proven-in-Coq) correctness implies that the interpretations of the C and Asm as state machines are equivalent, another metalogical argument. Like the C code, the Asm code has a dual interpretation, one as a CompCert target (the CompCert RISC-V semantics), and another according to the Riscette semantics, our implementation of the CompCert RISC-V semantics in Rosette. Like the dual interpretation of C, these semantics must align (section 6.6).

Trusted computing base. Among the code the HSM developer writes, the App Spec and IPR driver (model of how a well-behaved host communicates) are in the trusted computing base (TCB). The Low* app implementation, system software including device drivers (that run on the HSM), and hardware are all untrusted, and covered by verification. Chapter 7 describes the TCB of the Parfait framework itself.

3.3 Discussion

Parfait uses separate verification tools and frameworks for different parts of the stack, using the best tool for the job: the Starling framework for software built on top of F* to leverage Low* and reuse HACL* proofs; the Knox framework for hardware built on top

of Rosette [104] to leverage Rosette's hybrid symbolic execution capabilities; and theory verified in Coq [103] to tie everything together. Parfait does not currently have a mechanized connection between these tools. This is a tradeoff: using the best tool for each job means that formality gaps arise when combining separate verification tools used to verify sophisticated systems. This is a relatively new technique in systems verification, used in projects like DaisyNFS [29], which verifies a file system using Dafny [66] and Coq. Formality gaps can be eliminated (usually with considerable effort) by using a single verification framework for the whole stack [40, 42].

Chapter 4

Formalizing IPR

This chapter begins with a formalization of state machines (section 4.1) and then precisely states the definition of information-preserving refinement (section 4.2). Finally, it formalizes the four proof techniques for IPR (section 4.3) that are used in the Parfait approach, including the *lockstep* property proved by Starling when verifying software and the *functional-physical simulation* property proved by Knox when verifying hardware. This presentation closely follows the Coq formalization.[1]

4.1 State machines

We formalize state machines as follows:

```
Inductive result (T S : Type) :=
| Result : T -> S -> result T S.

Record machine (input output : Type) :=
  {
    state : Type;
    init : state;
    step : state -> input -> result output state -> Prop;
    reset : state -> state -> Prop;
  }.
```

State machines are parameterized by an `input` and `output` type, have an internal `state`, and start with an initial state given by `init`. Their behavior is described by two relations:

[1] github.com/anishathalye/ipr

`step` is the transition relation that describes how an input produces an output along with a new state; `reset` describes the reset behavior of the machine. We illustrate machines as shown in figure 4-1, emphasizing the I/O interface of the machine.

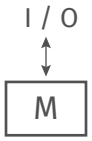


Figure 4-1: Illustration of a state machine M of type `machine` I O.

4.1.1 Application to HSMs

Parfait models both functional specifications and physical implementations as state machines. Figure 3-1 shows an example of a functional specification, a state machine specifying the ECDSA signing HSM (in F*, not Coq). Its inputs and outputs are high-level commands and responses (like `Sign msg` and `Signature sig`). In practice, our specifications' behaviors are described by step functions, so the `step` relation is deterministic and total:

```
Definition deterministic {I O : Type} (M : machine I O) : Prop :=
  forall s i r1 r2,
    M.(step) s i r1 ->
    M.(step) s i r2 ->
    r1 = r2.

Definition total {I O : Type} (M : machine I O) : Prop :=
  forall s i,
  exists r,
    M.(step) s i r.
```

Furthermore, specifications do not make use of reset behavior: for specs, `reset` is the identity relation:

```
Definition no_reset {I O : Type} (M : machine I O) : Prop :=
  M.(reset) = fun s s' => s = s'.
```

Parfait models implementations as state machines as well; their inputs and outputs correspond to wire-level inputs and outputs at the cycle-precise level. For example, for the ECDSA HSM implementation (figure 1-1), the inputs and outputs (in Coq syntax) are:

```
Record input := { rx : bool; cts : bool }.

Record output := { tx : bool; rts : bool }.
```

The implementation's `state` is the internal state of the circuit (all registers and memories), and the `step` relation describes the execution of the circuit for a single cycle. In our implementations, the circuit's `step` behavior is deterministic and total. The state machine's `reset` relation models what happens when the HSM is reset or unplugged from the host machine. It is a total (but not deterministic) relation that havocs any volatile registers and memories in the circuit's state and asserts the reset signal for a cycle, which will cause the circuit to set the program counter to the boot address, among other initialization behavior.

When discussing state machines corresponding to functional specifications, we use the term *functional interface* to describe the spec-level I/O interface; for physical implementations, we use the term *physical interface* to describe the wire-level I/O interface.

Parfait specifications and implementations have deterministic `step` functions, but the IPR formalization supports nondeterminism. Furthermore, the formalism uses `machines` to define the IPR real and ideal worlds, and the real world (section 4.2.2) can be nondeterministic even with a deterministic implementation if the driver is nondeterministic; our hardware drivers are nondeterministic in practice (section 6.3).

4.1.2 Refinement and equivalence

When state machines have matching `input` and `output` types, we can define what it means for them to be observationally equivalent. We define this state machine equivalence in terms of trace inclusion. We first define traces:

```
Inductive event (I O : Type) :=
| IO : I -> O -> event I O
| Reset : event I O.

Definition trace (I O : Type) := list (event I O).
```

An event is either an I/O event or a reset event, and a trace is any sequence of events. Then, we define executions of a machine:

```
Inductive execution {I O : Type} (M : machine I O) :
  M.(state) -> trace I O -> M.(state) -> Prop :=
| ExecutionEmpty : forall s,
    execution _ s nil s
| ExecutionStep : forall s i s' o tr s'',
    M.(step) s i (Result o s') ->
    execution _ s' tr s'' ->
    execution _ s (IO i o :: tr) s''
| ExecutionReset : forall s s' tr s'',
    M.(reset) s s' ->
    execution _ s' tr s'' ->
    execution _ s (Reset :: tr) s''.
```

An `execution` $M\ s\ tr\ s'$ relates a machine M starting from state s and a trace tr to a final state s'. Next, we define what it means for a trace to be included in the set of traces of a machine:

```
Definition in_traces {I O : Type} (M : machine I O) (tr : trace I O) : Prop :=
  exists sf, execution M M.(init) tr sf.
```

A trace tr is contained in the set of traces for a machine if there exists a final state s_f such that there's an execution from the initial state and trace to the final state. Finally, we define the notion that one machine refines another using trace inclusion, and we define behavioral equivalence as bidirectional refinement:

```
Definition refines {I O : Type} (M1 M2 : machine I O) : Prop :=
  forall tr,
    in_traces M1 tr ->
    in_traces M2 tr.

Definition equivalent {I O : Type} (M1 M2 : machine I O) : Prop :=
  refines M1 M2 /\ refines M2 M1.
```

This definition of equivalence applies only to relate two state machines with the same I/O interface.

4.2 Defining non-leakage

We define and formalize a new notion of state machine refinement called *information-preserving refinement (IPR)* to relate two state machines, formalizing the intuition from section 3.1.2. IPR relates an implementation state machine M_1 : `machine` I_1 O_1 to a specification state machine M_2 : `machine` I_2 O_2 with a different I/O interface and captures non-leakage.

4.2.1 I/O multiplexing

As a component of our formalization, we define the notion of multiplexing the interface to a state machine to expose a "left and right interface," as illustrated in figure 4-2. This merely tags inputs/outputs as "left" or "right" and does not otherwise change the behavior of the machine.

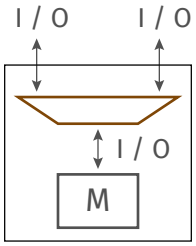


Figure 4-2: I/O multiplexing, to adapt the I/O interface of a machine M : `machine` I O to a `machine` $(I + I)$ $(O + O)$.

Formally, multiplexing is defined as follows:

```
Variable I O : Type.
Variable M : machine I O.

Inductive mux_step : M.(state) -> (I + I) -> result (O + O) M.(state) -> Prop :=
| MuxStepL : forall s i s' o,
    M.(step) s i (Result o s') ->
    mux_step s (inl i) (Result (inl o) s')
| MuxStepR : forall s i s' o,
    M.(step) s i (Result o s') ->
    mux_step s (inr i) (Result (inr o) s').

Definition mux : machine (I + I) (O + O) :=
  {|
    state := M.(state);
    init := M.(init);
    step := mux_step;
    reset := M.(reset);
  |}.
```

An input tagged `inl` always produces an `inl`-tagged output, and analogously, an `inr`-tagged input produces an `inr`-tagged output.

4.2.2 Real world

A *driver*, similar to a device driver in an operating system, describes how to obtain spec-level behavior from the implementation-level interface. In the HSM setting, the driver describes how spec-level operations translate to wire-level interaction with the HSM implementation. For example, for the ECDSA HSM (figure 1-1) the driver encodes how a spec-level command like `Sign msg` is encoded as bytes, how the encoded bytes are sent to the HSM over its wire-level interface following the UART protocol, and how the response is read and decoded into a spec-level response like `Signature sig`.

In the Coq formalization, drivers are written in a shallowly-embedded language `dproc` that has the ability to `Call` implementation-level operations:

```
Inductive dproc (I O : Type) : Type -> Type :=
| DCall : I -> dproc _ _ O
| DRet : forall T, T -> dproc _ _ T
| DBind : forall T T1, dproc _ _ T1 -> (T1 -> dproc _ _ T) -> dproc _ _ T
| DWhile : dproc _ _ bool -> dproc _ _ unit -> dproc _ _ unit
| DChoose : forall T (p1 p2 : dproc _ _ T), dproc _ _ T.

Definition driver (I1 O1 I2 O2 : Type) := I2 -> dproc I1 O1 O2.
```

Drivers are programs that take in a spec-level input of type I_2 and return a `dproc` that may make calls to an underlying implementation with I/O types I_1 / O_1 and that returns a final spec-level output of type O_2. The `DWhile` command allows drivers to perform loops. `DChoose` allows drivers to be nondeterministic; for example, a driver for the ECDSA HSM might use `DChoose` to model a host that can wait an arbitrary number of cycles between sending bytes as part of the UART protocol. We also introduce some notation for constructors, including do-notation `_ <- _; _` and `>>=` for monadic bind and `||` for nondeterministic choice.

The driver's execution semantics are given by the relation `dexec`, which describes what it means to execute a `dproc` with respect to an underlying implementation state machine:

```
Inductive dexec {Il Ir Ol Or : Type} (M : machine (Il + Ir) (Ol + Or)) :
  forall T, dproc Il Ol T -> M.(state) -> result T M.(state) -> Prop :=
| DexecCall : forall s i s' o,
    M.(step) s (inl i) (Result (inl o) s') ->
    dexec _ _ (Call i) s (Result o s')
| DexecRet : forall T (v : T) s,
    dexec _ _ (Ret v) s (Result v s)
| DexecBind : forall T T1 (p : dproc _ _ T1) (p' : T1 -> dproc _ _ T) s v s' res,
    dexec _ _ p s (Result v s') ->
    dexec _ _ (p' v) s' res ->
    dexec _ _ (p >>= p') s res
| ...
```

A `dexec` M T d s (`Result` v s') relates a `dproc` d returning a value of type T running against a machine M with starting state s to an output v and final machine state s'. For brevity, we omit the rules for `While` and `Choose`. `Call` executes an operation against the left-side interface of a state machine of type `machine` $(I_l + I_r)$ $(O_l + O_r)$.

Definition. Figure 4-3 shows the definition of the IPR real world, which is parameterized by a machine M_1 : `machine` I_1 O_1 and a driver that implements a spec-level (I_2 / O_2) interface in terms of interactions with the implementation state machine. In the real world, the client of the state machine can either use the spec-level interface or the implementation-level interface to interact with the underlying state machine M_1.

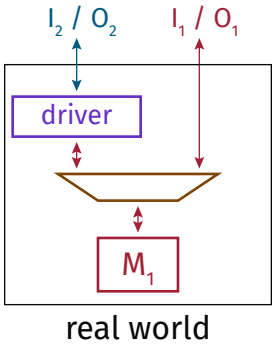


Figure 4-3: The IPR real world, which defines a state machine that adapts the implementation using the mux and driver to expose both a spec-level and an implementation-level interface.

In the HSM setting, this captures that a host machine can follow an I/O protocol (dictated by the driver) to obtain spec-level behavior from the HSM, or it can interact with the HSM in an arbitrary way at the wire level; the host can freely switch between these interfaces. A well-behaved host interacts with the implementation only through the driver. If the host is compromised, it may perform arbitrary wire-level I/O, interacting with the implementation directly through the wire-level interface and bypassing the driver (which captures protocol-following behavior as well, because such behavior is a subset of arbitrary wire-level I/O). A host switching back from the implementation-level interface to the spec-level interface represents the host recovering or the HSM being removed from the compromised machine and attached to a different one. In this case, we assume the HSM is issued a reset command by the host or receives a power-on reset. To model this, the real world keeps track of whether it's in "low-level mode" or "high-level mode" based on the previous input, and the real world issues a reset to the underlying implementation before transitioning from low-level mode to high-level mode.

Formalization. We begin by formalizing the notion of adding a driver to adapt the left-side interface of a state machine M_1 : `machine` $(I_1 + I_r)$ $(O_1 + O_r)$ to have a higher-level interface I_2 / O_2, adapting the overall machine to have type `machine` $(I_2 + I_r)$ $(O_2 + O_r)$. The state of this driver-equipped machine is the state of the underlying state machine M_1, along with a "mode" that tracks whether the last input was a low-level or high-level input:

```
Variable Ir Or I1 O1 I2 O2 : Type.
Variable M1 : machine (I1 + Ir) (O1 + Or).
Variable d : driver I1 O1 I2 O2.

Inductive driver_state (S : Type) :=
| DriverHigh : S -> driver_state S
| DriverLow : S -> driver_state S.
```

The step function for this state machine is defined as follows:

```
Inductive driver_step :
  driver_state M1.(state) ->
  (I2 + Ir) ->
  result (O2 + Or) (driver_state M1.(state)) ->
  Prop :=
| DriverStepLowLow : forall s il s' ol,
    M1.(step) s (inr il) (Result (inr ol) s') ->
    driver_step (DriverLow s) (inr il) (Result (inr ol) (DriverLow s'))
| DriverStepHighLow : forall s il s' ol,
    M1.(step) s (inr il) (Result (inr ol) s') ->
    driver_step (DriverHigh s) (inr il) (Result (inr ol) (DriverLow s'))
| DriverStepLowHigh : forall s s' i2 s'' o2,
    M1.(reset) s s' ->
    dexec _ _ (d i2) s' (Result o2 s'') ->
    driver_step (DriverLow s) (inl i2) (Result (inl o2) (DriverHigh s''))
| DriverStepHighHigh : forall s i2 s' o2,
    dexec _ _ (d i2) s (Result o2 s') ->
    driver_step (DriverHigh s) (inl i2) (Result (inl o2) (DriverHigh s')).
```

When the machine gets a low-level input, it simply passes it on to the underlying implementation machine M_1 (cases `DriverStepLowLow` and `DriverStepHighLow`). When the machine gets a high-level input, it executes the driver to drive the underlying machine (cases `DriverStepLowHigh` and `DriverStepHighHigh`). If the machine gets a high-level input while in low-level mode (`DriverStepLowHigh`), it first resets the underlying machine before running the driver. Next, we define the reset behavior of the real world machine:

```
Definition driver_reset (s1 s2 : driver_state M1.(state)) : Prop :=
  match s1 with
  | DriverHigh s1'
  | DriverLow s1' =>
      match s2 with
      | DriverHigh s2' => M1.(reset) s1' s2'
      | _ => False
      end
  end.
```

Reset is defined to reset the underlying implementation and switch back into high-level mode. Combining these definitions, we define `add_driver`:

```
Definition add_driver : machine (I2 + Ir) (O2 + Or) :=
  {|
    state := driver_state _;
    init := DriverHigh M1.(init);
    step := driver_step;
    reset := driver_reset;
  |}.
```

Finally, we define the `real_world` state machine, which is parameterized by a machine M_1 : `machine` I_1 O_1 and a driver d : `driver` I_1 I_2 O_1 O_2. This is the formal definition corresponding to figure 4-3:

```
Definition real_world {I1 O1 I2 O2 : Type}
  (M1 : machine I1 O1) (d : driver I1 O1 I2 O2) :
  machine (I2 + I1) (O2 + O1) :=
  add_driver (mux M1) d.
```

4.2.3 Ideal world

An *emulator* is a dual of the driver: it describes how to obtain implementation-level behavior from a spec-level interface. It differs from the driver in that the emulator is deterministic, and it has the ability to store internal state.

In the Coq formalization, emulators are written in a shallowly-embedded language `eproc` that have the ability to `Call` spec-level operations, as well as store and retrieve internal state via `Put` and `Get`:

```
Inductive eproc (S I O : Type) : Type -> Type :=
| ECall : I -> eproc _ _ _ O
| EGet : eproc _ _ _ S
| EPut : S -> eproc _ _ _ unit
| ERet : forall T, T -> eproc _ _ _ T
| EBind : forall T T1, eproc _ _ _ T1 -> (T1 -> eproc _ _ _ T) -> eproc _ _ _ T
| EWhile : eproc _ _ _ bool -> eproc _ _ _ unit -> eproc _ _ _ unit.

Record emulator (I2 O2 I1 O1 : Type) :=
  {
    estate : Type;
    einit : estate;
    estep : I1 -> eproc estate I2 O2 O1
  }.
```

Emulators have an internal state type, an initial state, and a step function that takes in an implementation-level input of type I_1 and returns an `eproc` that may make calls to an underlying specification state machine with I/O types I_2 / O_2 and return a final implementation-level output of type O_1. Similar to drivers, we introduce notation for constructors, including do-notation `_ <- _; _` and `>>=` for monadic bind.

The emulator's semantics are given by the relation `eexec`, which describes what it means to execute an `eproc` with respect to an underlying specification state machine:

```
Inductive eexec {S Il Ir Ol Or : Type} (M : machine (Il + Ir) (Ol + Or)) :
  forall T, eproc S Ir Or T ->
  (M.(state) * S) ->
  result T (M.(state) * S) ->
  Prop :=
| EexecCall : forall ms es i ms' o,
    M.(step) ms (inr i) (Result (inr o) ms') ->
    eexec _ _ (Call i) (ms, es) (Result o (ms', es))
| EexecGet : forall ms es,
    eexec _ _ (Get) (ms, es) (Result es (ms, es))
| EexecPut : forall (es' : S) ms es,
    eexec _ _ (Put es') (ms, es) (Result tt (ms, es'))
| ...
```

An `eexec` M T e (s, σ) (`Result` v (s', σ')) relates a `eproc` e with emulator state σ returning a value of type T running against a machine M with starting state s to an output v, final emulator state σ', and final machine state s'.

Definition. Figure 4-4 shows the definition of the IPR ideal world, which is parameterized by a machine M_2 : `machine` I_2 O_2 and an emulator that implements an implementation-level (I_1 / O_1) interface in terms of interactions with the specification state machine. Like in the real world, in the ideal world, the client of the state machine can either use the spec-level interface or the implementation-level interface to interact with the underlying state machine M_2.

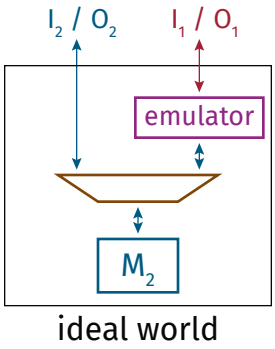


Figure 4-4: The IPR ideal world, which defines a state machine that adapts the specification using the mux and emulator to expose both a spec-level and an implementation-level interface.

The ideal world is designed to implicitly capture security guarantees by being as abstract as the specification. In the HSM setting, the setup captures that a host machine can

obtain spec-level behavior, or it can interact at a wire level, freely switching between these interfaces. A well-behaved host interacts with the ideal world machine only through the spec-level interface, and this interaction is fed through directly to the underlying specification state machine M_2, so these interactions are secure by construction. If the host is compromised, it may perform arbitrary implementation-level I/O; this goes to the emulator, which invokes spec-level operations but has no visibility into the internals of the specification, only query access—any outputs that come out of the emulator are merely computed based on outputs from the specification itself, so interaction with the emulator cannot leak any more information than the specification itself does.

Like the real world, the ideal world keeps track of a "mode." When the ideal world is operating in high-level mode, the emulator retains no state; when the ideal world gets a low-level input, the emulator state is initialized to the constant `einit`, and emulator state is retained for subsequent low-level operations until the next high-level input. In the HSM setting: if the host machine is compromised and switches from the high-level interface to the low-level one, it interacts with the specification through an emulator that's initialized with a constant state, so the emulator does not obtain any extra information or retain any information from past interactions of the host with the emulator. This is relevant if the host is compromised, restored, and then compromised again: the information that can be obtained by an adversary could also be obtained via query access to the specification at the moment of the second compromise.

Formalization. We begin by formalizing the notion of adding an emulator to adapt the right-side interface of a state machine M_2 : `machine` $(I_l + I_2)$ $(O_l + O_2)$ to have a lower-level interface I_1 / O_1, adapting the overall machine to have type `machine` $(I_l + I_1)$ $(O_l + O_1)$. The state of this emulator-equipped machine is the state of the underlying state machine M_2, along with a "mode" that tracks whether the last input was a low-level input or high-level input; additionally, in the case of the low-level mode, the state contains an emulator-internal state.

```
Variable Il Ol I1 O1 I2 O2 : Type.
Variable M2 : machine (Il + I2) (Ol + O2).
Variable e : emulator I2 O2 I1 O1.

Inductive emulator_state (S ES : Type) :=
| EmulatorHigh : S -> emulator_state S ES
| EmulatorLow : S -> ES -> emulator_state S ES.
```

The step function for this state machine is analogous to `driver_step`, except that it initializes the emulator state when switching from high-level to low-level mode, and it threads through the emulator state when operating on a sequence of low-level inputs. The reset behavior of this machine is analogous to `driver_reset`.

Using the step and reset relations, we can define:

```
Definition add_emulator : machine (Il + I1) (Ol + O1) := ...
```

Finally, we define the `ideal_world` state machine, which is parameterized by a machine M_2 : `machine` I_2 O_2 and an emulator e : `emulator` I_2 O_2 I_1 O_1. This is the formal definition corresponding to figure 4-4:

```
Definition ideal_world {I1 O1 I2 O2 : Type}
  (M2 : machine I2 O2) (e : emulator I2 O2 I1 O1) :
  machine (I2 + I1) (O2 + O1) :=
  add_emulator (mux M2) e.
```

4.2.4 IPR definition

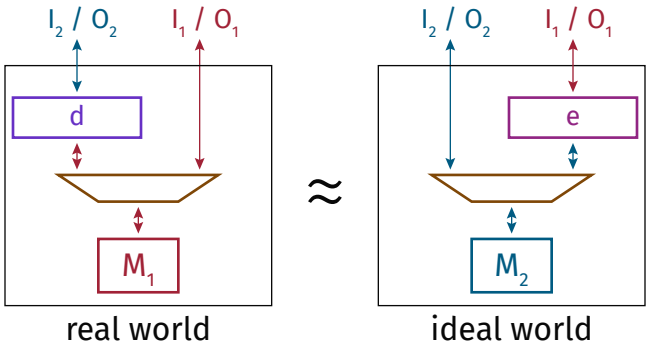


Figure 4-5: The definition of IPR: an implementation M_1 is an information-preserving refinement of a specification M_2 with respect to a driver d if there exists an emulator e such that the real world is observationally equivalent to the ideal world.

Figure 4-5 illustrates the definition of IPR, an equivalence between the real world and ideal world. We formalize M_1 being an information-preserving refinement of M_2 with respect to a driver d, $M_1 \approx_{IPR[d]} M_2$, as follows:

```
Definition IPR {I1 O1 I2 O2}
  (M1 : machine I1 O1) (M2 : machine I2 O2) (d : driver I1 O1 I2 O2) : Prop :=
  exists (e : emulator I2 O2 I1 O1),
    equivalent (real_world M1 d) (ideal_world M2 e).
```

4.3 Proof techniques

Parfait uses four proof techniques for IPR for different levels of abstraction. The overall approach relies on the transitivity of IPR. Proofs of some intermediate levels rely on the property that equivalence between state machines implies IPR with the identity driver. The software-level proof uses the *lockstep* proof technique, which simplifies proving IPR when there is a one-to-one relationship between implementation-level and spec-level state machine steps. Finally, the hardware-level proof uses the *functional-physical simulation* proof technique, which supports proofs of IPR using refinement relations even when there isn't a one-to-one correspondence between state machine steps.

4.3.1 Transitivity

Figure 4-6 illustrates the transitivity of IPR. We first formalize the composition of drivers, denoted as $d_1 \circ d_2$, as inlining an invocation of the lower-level driver wherever the higher-level driver issues a `Call`:

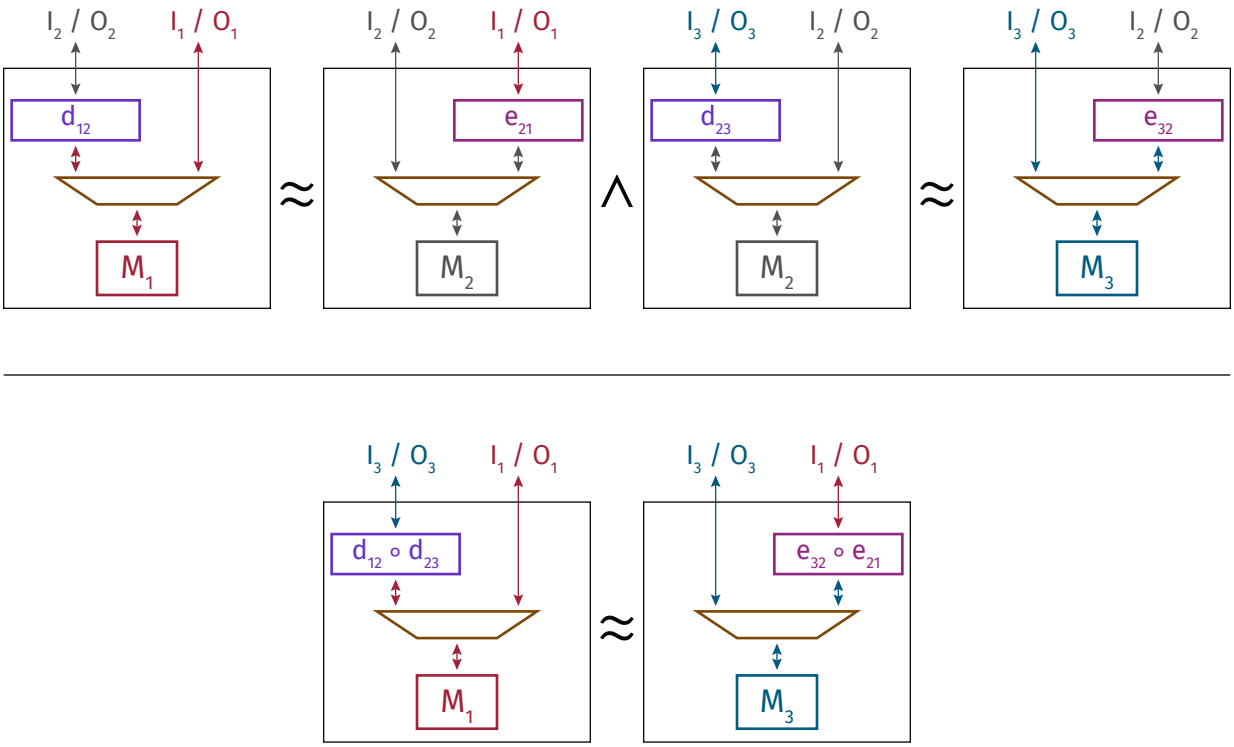


Figure 4-6: Inference rule illustrating the transitivity of IPR. The drivers and emulators compose when relating M_1 to M_3.

```
Fixpoint inline_calls {I1 O1 I2 O2 T : Type}
  (p : dproc I2 O2 T)
  (d1 : driver I1 O1 I2 O2) : dproc I1 O1 T :=
  match p with
  | Call i => d1 i
  | Ret v => Ret v
  | p1 >>= p2 => x <- inline_calls p1 d1; inline_calls (p2 x) d1
  | While g b => While (inline_calls g d1) (inline_calls b d1)
  | p1 || p2 => inline_calls p1 d1 || inline_calls p2 d1
  end.

Definition driver_compose {I1 O1 I2 O2 I3 O3 : Type}
  (d1 : driver I1 O1 I2 O2)
  (d2 : driver I2 O2 I3 O3) : driver I1 O1 I3 O3 :=
  fun i3 => inline_calls (d2 i3) d1.
```

Using this, we can formalize transitivity of IPR, the formal definition corresponding to figure 4-6:

```
Theorem IPR_by_transitivity :
  forall (I1 O1 I2 O2 I3 O3 : Type)
    (M1 : machine I1 O1) (M2 : machine I2 O2) (M3 : machine I3 O3)
    (d12 : driver I1 O1 I2 O2) (d23 : driver I2 O2 I3 O3),
    IPR M1 M2 d12 ->
    IPR M2 M3 d23 ->
    IPR M1 M3 (driver_compose d12 d23).
```

This proof technique for IPR constructs an emulator between M_1 and M_3 implicitly, using the emulators between M_1 / M_2 and M_2 / M_3.

Proof sketch. Figure 4-7 illustrates the proof of transitivity of IPR, which uses a series of equivalences between hybrid machines. Equation (1) uses a lemma that `driver_compose` corresponds to the composition of `add_driver`: that adding a single composed driver $d_{12} \circ d_{23}$ to a state machine is equivalent to nested calls to `add_driver` to add the two drivers to the state machine one on top of another. Equation (2) uses a lemma that wrapping equivalent state machines with `add_driver` produces equivalent state machines; the inner machines here are equivalent according to the first hypothesis of the transitivity property. Equation (3) uses a lemma that `add_driver` and `add_emulator` commute. Equation (4) is analogous to equation (2), using a lemma that wrapping equivalent state machines with `add_emulator` produces equivalent state machines; the inner machines here are equivalent according to the second hypothesis of the transitivity property. Equation (5) is analogous to equation (1), using a lemma that emulator composition corresponds to the composition of `add_emulator`.

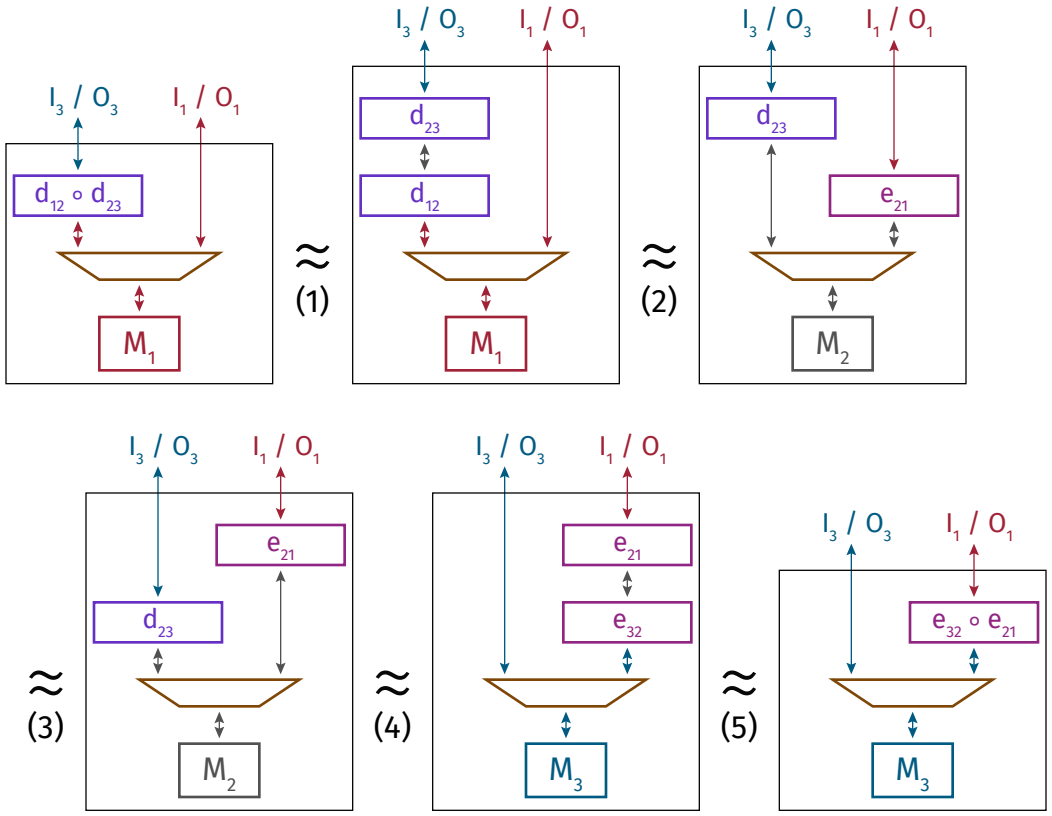


Figure 4-7: The IPR transitivity proof uses a series of equivalences using four hybrid machines.

4.3.2 Equivalence

Figure 4-8 illustrates IPR by equivalence, where two state machines that are equivalent have an IPR between them with the identity driver.

```
Definition identity_driver (I O : Type) : driver I O I O :=
  fun i => (Call i).

Theorem IPR_by_equivalence :
  forall I O (M1 M2 : machine I O),
    equivalent M1 M2 ->
    IPR M1 M2 (identity_driver I O).
```

This proof technique for IPR does not require the user to supply an emulator; instead, the proof uses an identity emulator.

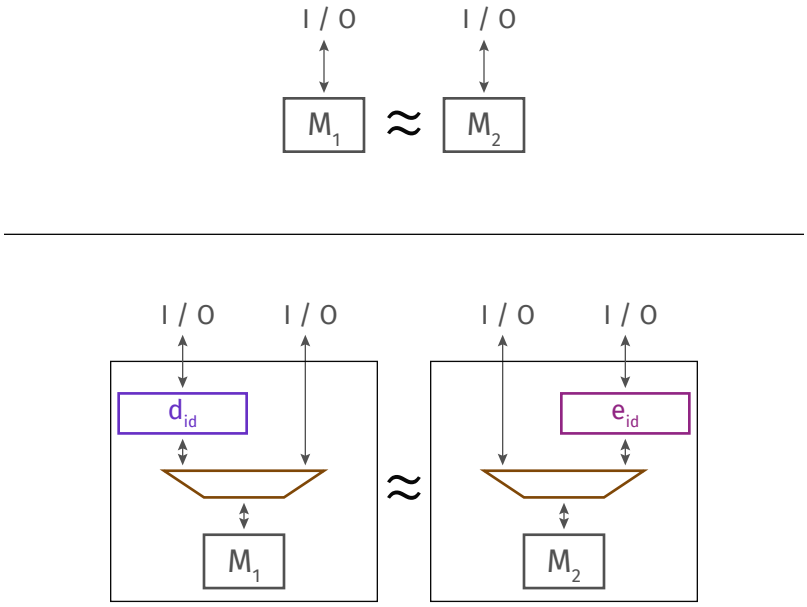


Figure 4-8: IPR by equivalence: state machines that are equivalent have an IPR between them with the identity driver.

4.3.3 Lockstep

The *lockstep* proof strategy applies when there is a one-to-one relationship between steps of an implementation state machine and a specification state machine. In this case, the driver takes on a simplified form, composed of encode/decode functions describing how input/output types are converted.

```
Variable I1 O1 I2 O2 : Type.

Variable encode_input : I2 -> I1.
Variable decode_output : O1 -> O2.

Definition lockstep_driver : driver I1 O1 I2 O2 :=
  fun i2 =>
    o1 <- Call (encode_input i2);
    Ret (decode_output o1).
```

The lockstep proof strategy implicitly constructs an emulator based on user-supplied encode/decode functions that are duals of those comprising the driver:

```
Variable decode_input : I1 -> option I2.
Variable encode_output : option O2 -> O1.
```

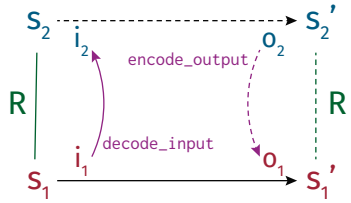
The `decode_input` function produces an `option`-typed output to allow for low-level inputs that do not correspond to any high-level input. The `encode_output` function consumes an `option`-typed input to support producing low-level outputs for the situation where there is no valid high-level input and hence no valid high-level output. Given such functions, the emulator is defined as follows:

```
Definition lockstep_emulator : emulator I2 O2 I1 O1 :=
  {|
    estate := unit;
    einit := tt;
    estep :=
      fun i1 => match decode_input i1 with
                | Some i2 => o2 <- Call i2; ERet (encode_output (Some o2))
                | None => ERet (encode_output None)
                end;
  |}.
```

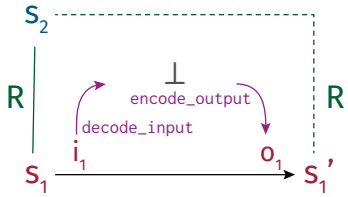
The lockstep proof strategy requires supplying the encode/decode functions (which implicitly defines an emulator), proving a correspondence between encoders and decoders, finding a refinement relation between machine states, and showing a property similar to forward simulation [74] which we call *lockstep simulation*, which together imply IPR:

```
Variable M1 : machine I1 O1.
Variable M2 : machine I2 O2.

Variable R : M1.(state) -> M2.(state) -> Prop.

Theorem IPR_by_lockstep :
  no_reset M1 ->
  no_reset M2 ->
  total M1 ->
  deterministic M2 ->
  (forall i2, decode_input (encode_input i2) = Some i2) ->
  (forall o2, decode_output (encode_output (Some o2)) = o2) ->
  R M1.(init) M2.(init) ->
  (forall s1 s2 i1 o1 s1',
      R s1 s2 ->
      M1.(step) s1 i1 (Result o1 s1') ->
      match decode_input i1 with
      | Some i2 => exists o2 s2',
                     M2.(step) s2 i2 (Result o2 s2') /\
                     R s1' s2' /\
                     o1 = encode_output (Some o2)
      | None => o1 = encode_output None /\
                s1' = s1
      end) ->
  IPR M1 M2 lockstep_driver.
```

The theorem requires some additional conditions to hold on M_1 and M_2: neither can have reset behavior, M_1 must be total, and M_2 must deterministic. Figure 4-9 illustrates the key property of lockstep simulation between M_1 and M_2.



(a) Lockstep simulation (`Some` case): if s_1 steps to s_1' with input i_1 and output o_1, and `decode_input` $i_1 =$ `Some` i_2, then it must be possible for any s_2 related by R to s_1 to step with input i_2 and some output o_2 to an s_2' that's related by R to s_1' such that `encode_output` (`Some` o_2) $= o_1$.



(b) Lockstep simulation (`None` case): if s_1 steps to s_1' with input i_1 and output o_1, and `decode_input` $i_1 =$ `None`, then it must be the case that `encode_output` `None` $= o_1$ and for any s_2 related by R to s_1, s_2 must also be related by R to s_1'.

Figure 4-9: Lockstep simulation, for the case where the low-level input corresponds to some high-level input (a) or none (b).

4.3.4 Functional-physical simulation

Functional-physical simulation is a generalization of forward simulation [74] that supports separately reasoning about high-level inputs and sequences of low-level inputs, when there exists a refinement relation between the implementation machine and specification machine that holds only in the high-level mode of the real/ideal worlds and after reset but not in between low-level steps. This proof technique for IPR requires the user to explicitly construct and pass in an emulator.

Functional simulation, reasoning about individual high-level inputs, is formalized as follows:

```
Variable I1 O1 I2 O2 : Type.
Variable M1 : machine I1 O1.
Variable d : driver I1 O1 I2 O2.
Variable M2 : machine I2 O2.

Variable R : M1.(state) -> M2.(state) -> Prop.

Definition functional_simulation : Prop :=
  (* R holds on initial state *)
  R M1.(init) M2.(init) /\
    (* R is preserved by reset *)
    (forall s1 s2 s1',
        R s1 s2 ->
        M1.(reset) s1 s1' ->
        R s1' s2) /\
    (* at least one driver execution terminates *)
    (forall s1 s2 i2,
        R s1 s2 ->
        exists o2 s1',
          dexec (mux M1) _ (d i2) s1 (Result o2 s1')) /\
    (* driver output matches spec, and R holds *)
    (forall s1 s2 i2 o2 s1',
        R s1 s2 ->
        dexec (mux M1) _ (d i2) s1 (Result o2 s1') ->
        exists s2', M2.(step) s2 i2 (Result o2 s2') /\ R s1' s2').
```

Figure 4-10 illustrates the last conjunct, the core of the functional simulation definition, which states that there must be a correspondence between driver executions and the specification's high-level input-output behavior.

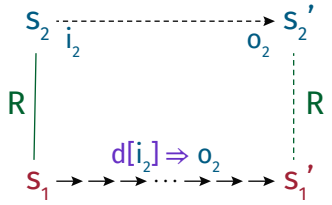


Figure 4-10: Functional simulation: if s_1 steps to s_1' through the interaction of the driver d running on high-level input i_2 that produces output o_2, then it must be possible for any s_2 related by R to s_1 to step with the input i_2 to produce output o_2 and a final state s_2' that is related by R to s_1'.

Physical simulation reasons about sequences of low-level inputs, where a refinement

relation holds only at the start of the simulation and after a reset.

```
Definition io_to_machine_trace (io : list (I1 * O1)) : trace I1 O1 :=
  map (fun '(i, o) => IO i o) io.

Definition io_to_ideal_machine_trace (io : list (I1 * O1)) :
  trace (I2 + I1) (O2 + O1) :=
  map (fun '(i, o) => IO (inr i) (inr o)) io.

Definition physical_simulation : Prop :=
  exists (e : emulator I2 O2 I1 O1),
  forall s1 s2 io s1' s1'',
    R s1 s2 ->
    execution M1 s1 (io_to_machine_trace io) s1' ->
    M1.(reset) s1' s1'' ->
    exists s2' e2',
      execution (add_emulator' M2 e) (EmulatorHigh s2)
        (io_to_ideal_machine_trace io) (EmulatorLow s2' e2') /\
        R s1'' s2'.
```

Figure 4-11 illustrates the definition, which states that there must be a correspondence between the implementation's low-level input-output behavior and emulator execution.

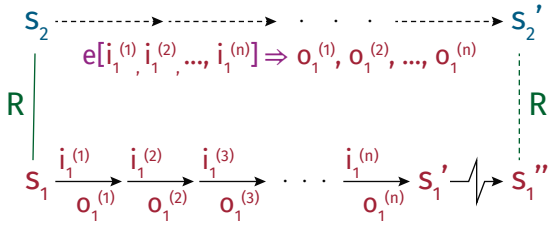


Figure 4-11: Physical simulation: if s_1 steps by a sequence of inputs $i_1^{(1)}, i_1^{(2)}, \ldots, i_1^{(n)}$ producing outputs $o_1^{(1)}, o_1^{(2)}, \ldots, o_1^{(n)}$ to a state s_1' and resets to state s_1'', then it must be possible for any s_2 related by R to s_1 to step, through the interaction of the emulator e running on the low-level inputs that produces matching low-level outputs, to a final state s_2' that is related by R to s_1''.

Finally, we can state the theorem that states that IPR is implied by functional simulation and physical simulation:

```
Definition resettable {I O : Type} (M : machine I O) :=
  forall s, exists s', M.(reset) s s'.

Theorem IPR_by_functional_physical_simulation :
  total M1 ->
  deterministic M2 ->
  resettable M1 ->
  no_reset M2 ->
  functional_simulation ->
  physical_simulation ->
  IPR M1 M2 d.
```

This theorem requires some additional conditions to hold on M_1 and M_2: M_1 must be total, M_2 must be deterministic, M_1 must be possible to reset, and M_2 must not have reset behavior.

4.4 Limitations

Randomness. The formalization of IPR presented in this thesis does not support randomness, so Parfait cannot be used to verify HSMs that use true random number generators (TRNGs). Karatroc [120] extends IPR with support for randomness.

While state machines, and therefore specifications, can be nondeterminstic, it is not a substitute for support for randomness, because it's possible to leak information through nondeterminism. In practice, our specifications are deterministic, to enable easier auditing and because we don't need nondeterminism.

As an alternative to randomness, HSMs can use cryptographically-secure pseudo-random number generators (CSPRNGs), and this fits into IPR, because IPR supports internal state. The specification can internally use a CSPRNG, the spec can be augmented to expose an operation to add entropy to the CSPRNG, and this operation can be called by the host at device initialization time (and again at any time later) to seed the random number generator. IPR ensures that the CSPRNG's internal state cannot be leaked by the implementation. The Parfait ECDSA HSM uses this approach to compute the signature nonce.

Emulator efficiency. To meaningfully apply IPR to specifications that involve cryptography, the adversary must be efficient, and therefore, the emulator must be efficient as well. Without an efficiency requirement, an implementation that, for example, leaks an RSA signing key, could be justified by an emulator that calls the specification to get the public key,

factors products of large primes in exponential time to compute the private key, and then perfectly mimics the physical interface because it has determined the implementation's internal state.

The emulator must satisfy a coarse-grained notion of efficiency: being prohibited from performing exponential-time computation and brute-forcing secrets. Without an efficiency requirement, IPR captures an information-theoretic notion of information preservation, rather than a computational one.

The Parfait framework does not fully formalize or mechanically verify emulator efficiency. Instead, the proofs rely on a manual audit of the emulator code. Most emulators are constructed implicitly by the IPR proof techniques; the emulators we construct directly (section 6.3) are simple, so the efficiency property is easy to check. In fact, the Parfait emulators in our case studies satisfy a stricter definition of efficiency than necessary — per cycle of the circuit that they emulate, they perform at most one query to the specification and perform computation roughly equivalent to what the circuit does in one cycle — meaning that an adversary could run the emulator with computational resources equivalent to the circuit itself.

Chapter 5

Verifying IPR for software with Starling

Parfait and its *Starling* framework support the application developer in implementing the HSM software and proving IPR between the specification and the assembly-level code. A key challenge in the software verification component of Parfait is minimizing proof effort and enabling reuse of existing specifications, implementations, and proofs. Building on top of existing verified software is a challenge because these libraries are focused on verifying functional correctness, not non-leakage and IPR.

Starling addresses this challenge by encoding the *lockstep* property, which implies IPR (section 4.3.3), as a Hoare logic precondition/postcondition. Thanks to this encoding, the application developer can build on top of existing verified software libraries and their functional correctness proofs, which are themselves written in precondition/postcondition style.

Figure 5-1 shows the software verification component of Parfait's overall verification approach (figure 3-3). The application software proof consists of three refinements between four levels of abstraction: from the top-level functional specification in F* [102] (section 5.1), to the implementation in Low* [94] (section 5.2), to its extraction to C, to its compilation to assembly (section 5.3). With the support of Parfait's Starling framework, the app developer proves that the Low* implementation satisfies the lockstep property with respect to the functional specification. The rest of the verification is automated. KaRaMeL [94] together with the CompCert [67] verified compiler produce an assembly-level implementation that is *equivalent* to the Low* implementation; this compilation step imposes no proof burden on the developer. Parfait's verified theory of IPR (section 4.3) shows that lockstep and equivalence imply IPR.

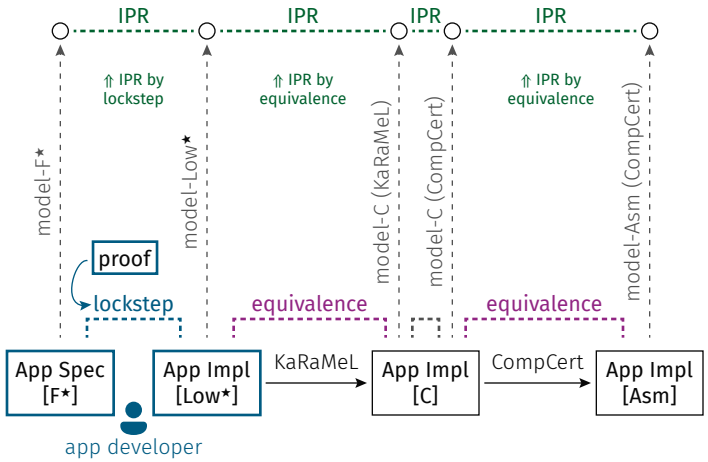


Figure 5-1: The Parfait software verification approach. The app developer, with the support of Parfait's Starling framework, proves the *lockstep* property between the spec and the implementation.

5.1 Functional specification

Top-level specifications in Parfait are written in explicit state machine style in F*, satisfying the `spec_t` interface shown in figure 5-2. Specifications can use mathematical constructs such as natural numbers or definitions of elliptic-curve point multiplication. Figure 3-1 shows an example, the step function for the ECDSA signing HSM's spec.

```
noeq type spec_t = {
  state_t:Type;
  command_t:Type;
  response_t:Type;
  initial_state:state_t;
  step:state_t -> command_t -> state_t & response_t;
}
```

Figure 5-2: Encoding of specifications in Starling.

The specification also includes the driver (not shown in figure 3-1), which describes how spec-level functionality is obtained from interacting with the implementation. Because there is a one-to-one correspondence between spec steps and implementation steps—one run of `handle` maps to one step of the spec `step` function—the driver is comprised of encode/decode functions to (1) serialize the spec-level input (e.g., `Sign msg`) as a buffer, and (2) decode the response buffer into a spec-level output (e.g., `Signature sig`).

5.2 Implementation and proof

The developer writes the implementation in Low* as a function that mutates fixed-length buffers: it gets a command and a state buffer along with a pointer to a response buffer, and it updates the state and produces a response. Parfait interprets the implementation as a state machine in the formalism of IPR as follows. The state, input, and output types are defined as fixed-length sequences of bytes, and the `handle` function (and its interpretation according to the Low* semantics) defines the step function of the state machine as an atomic step. Figure 5-3 shows pseudocode that more precisely describes how the metalogical model-Low* interprets the implementation as a state machine.

```
type state = bytes[STATE_LEN]
type input = bytes[COMMAND_LEN]
type output = bytes[RESPONSE_LEN]

def step(state: state, input: input) -> (state, output):
  m = lowstar_machine("AppImpl.low.fst")

  # copy state and command into machine memory
  state_ptr = m.alloc(STATE_LEN)
  m.storebytes(state_ptr, state)
  command_ptr = m.alloc(COMMAND_LEN)
  m.storebytes(command_ptr, input)

  # allocate space for response
  response_ptr = m.alloc(RESPONSE_LEN)

  # run handle function according to LowStar semantics
  m.invoke("handle", state_ptr, command_ptr, response_ptr)

  # retrieve updated state and result buffer
  new_state = m.loadbytes(state_ptr)
  output = m.loadbytes(response_ptr)
  return (new_state, output)
```

Figure 5-3: Pseudocode describing how model-Low* interprets the implementation as a state machine.

The Starling framework encodes the *lockstep* property (section 4.3.3)—in particular, the encode/decode correspondences and the *lockstep simulation* property—into F*, and the app

developer proves the properties.

Encode/decode correspondences. Starling provides the developer with signatures for the encode/decode functions for commands and responses, as shown in figure 5-4. The postconditions on the encode functions ensure that the functions satisfy the correspondence required for the lockstep property.

```
// encoder of a command into a byte sequence of a given length
let command_decoder (len:nat) (command_t:Type) =
  (s:Seq.seq uint8{Seq.length s = len}) -> option command_t

// decoder of a command from a byte sequence
let command_encoder (#len:nat)
                    (#command_t:Type)
                    (decoder:command_decoder len command_t) =
  cmd:command_t ->
  Pure (s:Seq.seq uint8{Seq.length s = len})
       (requires True)
       (ensures fun ret -> decoder ret == Some cmd)
```

Figure 5-4: Starling's encoding of encode/decode correspondence in F* as a postcondition on the encoder. The encoding for responses (not shown) is analogous.

Lockstep simulation. Starling encodes lockstep simulation into the signature of the Low* `handle` function, `handle_st`, as shown in figure 5-5. Rather than use a refinement relation between states, Starling uses a `state_encoder` function that encodes a spec-level state as bytes. The `handle_st` signature is parameterized by a specification and the encode/decode functions for commands/responses. The function takes as input the `state`, `command`, and `response` buffers, as well as the spec-level state, supplied as a ghost argument `state_spec`. The precondition states that the `state` and `state_spec` must correspond. The postcondition encodes the lockstep simulation condition (illustrated in figure 4-9), decoding the low-level input `command` into the spec-level `cmd_spec` and handling both the case of valid low-level input (`cmd_spec = Some v`, corresponding to figure 4-9a) and the case of invalid low-level input (`cmd_spec = None`, corresponding to figure 4-9b).

Informally, relating the `handle` function back to the spec `step` function while ensuring non-leakage establishes two properties, ensured by the combination of (1) the encode/decode correspondences, and (2) the postcondition on `handle`, which are:

```
let state_encoder (len:nat) (state_t:Type) (initial_state:state_t) =
  (st:state_t) -> (s:Seq.seq uint8{Seq.length s = len})

inline_for_extraction
let handle_st
  (spec: spec_t)
  (#state_len:nat) (#command_len:nat) (#response_len:nat)
  (encode_state:state_encoder state_len spec.state_t spec.initial_state)
  (#decode_command:command_decoder command_len spec.command_t)
  (encode_command:command_encoder decode_command)
  (#decode_response:response_decoder response_len spec.response_t)
  (encode_response:response_encoder decode_response) =
     state:B.buffer uint8{B.length state = state_len}
  -> state_spec:erased spec.state_t
  -> command:B.buffer uint8{B.length command = command_len}
  -> response:B.buffer uint8{B.length response = response_len} ->
  Stack unit
  (requires fun h -> ... /\ encode_state state_spec == B.as_seq h state)
  (ensures fun h0 () h1 -> ... /\
    (let cmd_spec = decode_command (B.as_seq h0 command) in
     match cmd_spec with
     | Some v -> let (state_spec_final, resp_spec) = spec.step state_spec v in
                 encode_state state_spec_final == B.as_seq h1 state /\
                 encode_response (Some resp_spec) == B.as_seq h1 response
     | None -> B.as_seq h1 state == B.as_seq h0 state /\
               encode_response None == B.as_seq h1 response))
```

Figure 5-5: Starling's encoding of lockstep simulation in F* as the signature of `handle`.

- When the command can be decoded as a spec-level command (i.e., not `None`), then the behavior matches the spec: the final state, when decoded, matches the final spec state, and encoding the spec-level response matches the value in the response buffer. This rules out encodings that leak information: `encode_response` is a deterministic function of only the spec-level response, and the buffer contents are equal to this, capturing non-leakage.
- When the command cannot be decoded as a spec-level command (i.e., `None`), then the state remains unchanged, and the response is deterministic, as given by `encode_response None`. A client that never supplies bad inputs will never observe this, but a client that does supply bad inputs will learn no information. This, along

with Low*'s other properties, such as verifying memory safety, ensures that even bad inputs (e.g., trying to trigger a buffer overflow) cannot corrupt the state or leak information.

The application developer proves that their implementation of `handle` satisfies `handle_st`, parameterized by the spec and encode/decode functions for commands/responses, which proves the lockstep property between implementation and specification. Starling's encoding of lockstep into a precondition/postcondition in `handle_st` enables building on top of existing proofs of correctness; for example, our ECDSA signing implementation builds on top of both the implementations and proofs of ECDSA from HACL* [122].

5.3 Assembly-level implementation and proof

Parfait compiles Low* to assembly code using a stack of verified compilers. KaRaMeL [94] compiles Low* code to C. KaRaMeL theorems establish that safety and functional correctness verified at the F* level translate to generated CompCert Clight code. Parfait then uses the formally-verified CompCert compiler to generate an assembly implementation of the `handle` function that follows the RISC-V calling convention, expecting pointers to the `state`, `command`, and `response` buffers in the `a0`, `a1`, and `a2` registers.

Parfait uses CompCert's RISC-V backend and dumps the AST of the last verified pass of the compiler, called Asm (after which the compiler usually runs un-verified expansion, assembly, and linking). The Asm machine model still uses CompCert's structured memory model and has pseudo-instructions for allocating and freeing stack frames.

Figure 5-6 describes how model-Asm (CompCert) interprets the Asm as a state machine using the CompCert semantics, where the invocation of `handle` is treated as a single atomic step of the state machine, analogous to model-Low* (figure 5-3). The step function takes as inputs a state buffer and command buffer and returns a new state buffer and response buffer as outputs.

The C code has two interpretations as a state machine, one according to KaRaMeL C semantics, and another according to the CompCert C semantics. To relate the Low* level to the C level with IPR, and to relate the C level to the Asm level with IPR, Parfait leverages the fact that all of the induced state machines are *equivalent*, which implies IPR (section 4.3.2).

```
type state = bytes[STATE_LEN]
type input = bytes[COMMAND_LEN]
type output = bytes[RESPONSE_LEN]

def step(state: state, input: input) -> (state, output):
  m = compcert_asm_abstract_machine("AppImpl.asm.json")

  # copy state and command into machine memory
  state_ptr = m.alloc(state_len)
  m.storebytes(state_ptr, state)
  command_ptr = m.alloc(command_len)
  m.storebytes(command_ptr, input)

  # allocate space for response
  response_ptr = m.alloc(response_len)

  # set up arguments following RISC-V ABI
  m.regs["a0"] = state_ptr
  m.regs["a1"] = command_ptr
  m.regs["a2"] = response_ptr

  # run handle function according to CompCert Asm semantics
  m.regs["pc"] = m.address_of("handle")
  m.run()

  # retrieve updated state and result buffer
  new_state = m.loadbytes(state_ptr)
  output = m.loadbytes(response_ptr)
  return (new_state, output)
```

Figure 5-6: Pseudocode describing how model-Asm (CompCert) interprets the CompCert Asm as a state machine according to the CompCert RISC-V Asm semantics.

5.4 Discussion

Parfait's software verification approach minimizes developer effort. Starling enables the developer to prove IPR using F* and Low*, standard software verification tools with a rich ecosystem of compiler and IDE support, which are designed for verifying C-like code. Parfait's IPR-by-lockstep proof technique and Starling's encoding of lockstep as a Hoare logic precondition/postcondition enable reusing and building on top of existing specifications, implementations, and proofs from verified libraries like HACL*, as we demonstrate in chapter 8. Parfait's IPR-by-equivalence proof technique and use of existing verified compilers to produce the assembly-level implementation further minimizes developer effort.

5.5 Limitations

KaRaMeL semantics are intended to coincide with CompCert C, but there is no mechanized connection between the two. Parfait assumes that the induced state machines (from model-C (KaRaMeL), interpreting the code according to the KaRaMeL C semantics, and model-C (CompCert), interpreting the code according to the CompCert C semantics) are behaviorally equivalent. Parfait does not need to assume that the semantics perfectly coincide (e.g., stepwise correspondence between the semantics), only that the induced state machines coincide, which boils down to assuming that that the final values computed/transformed by the `handle` function match between the two semantics. Additionally, unlike the CompCert compiler, KaRaMeL is only partially verified, and on-paper, rather than with a mechanically-checked proof of correctness.

Chapter 6

Verifying IPR for hardware with Knox

Parfait and its *Knox* framework support the platform developer in implementing the HSM hardware and proving IPR between the assembly-level code and its circuit-level execution. Knox addresses two key challenges, both related to the tractability of verification and minimizing developer effort. The first challenge is bridging the large gap between assembly-level execution, where there is no notion of timing, and the cycle-precise execution at the circuit level, with reasonable proof effort. The second challenge is modularity, separating the software proof from the hardware proof, so the platform developer doesn't need to understand the software and its proof, and vice versa. Knox addresses these challenges with automation, leveraging symbolic execution, satisfiability modulo theories (SMT) solvers, and novel performance optimization techniques.

Figure 6-1 shows the hardware verification component of Parfait's overall verification approach (figure 3-3), where the developer proves the final IPR between assembly-level implementation and the system-on-a-chip (SoC) using the Knox framework. This framework builds on hybrid symbolic execution [104] and SMT solvers to help the developer prove the *functional-physical simulation* property (section 4.3.4) between the Asm level and the SoC. Knox provides a symbolically-executable semantics for the Asm (section 6.1) and SoC (section 6.2) levels, languages and semantics for writing drivers and emulators (section 6.3), and proof checkers for functional simulation and physical simulation (section 6.4).

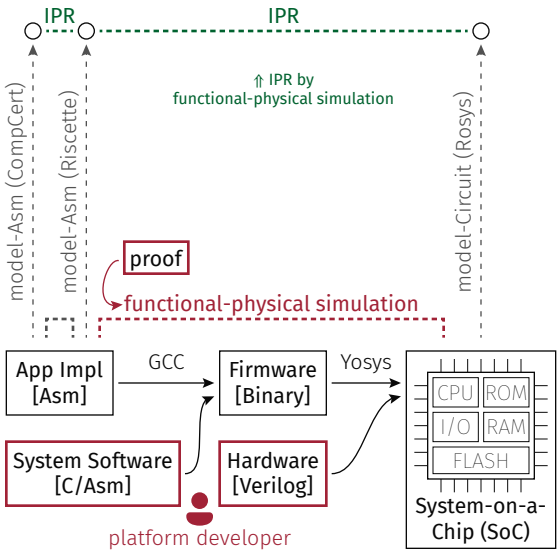


Figure 6-1: The Parfait hardware verification approach. The platform developer, with the support of Parfait's Knox framework, proves the *functional-physical simulation* property between the Asm level and the SoC.

6.1 Assembly-level implementation

Parfait generates the assembly-level implementation App Impl [Asm] using the CompCert compiler, which is written in Coq. CompCert includes a (non-executable) Coq semantics of RISC-V assembly. However, Knox is written using Rosette, a symbolic evaluation library for the Racket programming language. For this reason, on top of Rosette, Knox provides its own executable semantics for CompCert RISC-V assembly, which we call Riscette. This executable semantics closely follows the original CompCert semantics. Furthermore, the Riscette semantics can be single-stepped instruction-by-instruction, for proof purposes.

With this semantics in place, Knox can initialize an abstract machine from assembly code emitted by the CompCert compiler, set up the machine memory and registers to supply a state and input to the `handle` function, and symbolically execute it to produce a final state and an output, similar to how figure 5-6 describes the assembly level's interpretation as a state machine following CompCert semantics.

6.2 Hardware-level implementation

Parfait generates the complete system-on-a-chip using the Yosys synthesis tool; to reason about the circuit in Knox, the framework includes the Rosys tool, which converts the circuit to a symbolically-executable state-machine representation in Rosette. Given a circuit, Rosys

emits:

- An `input` type, with fields for all input wires to the circuit
- An `output` type, with fields for all output wires from the circuit
- A `state` type, with fields for all registers and memories of the circuit, as well as current inputs
- A function `set-input` : `state` × `input` → `state`, to set the values on the input wires
- A function `tick` : `state` → `state`, to execute the circuit for a single clock cycle
- A function `get-output` : `state` → `output`, to get the values on the output wires

Knox relates this Rosys-generated state machine representation of the hardware to the Riscette-generated state machine representation of the assembly-level implementation to prove functional-physical simulation between the two.

6.3 Drivers and emulators

Knox includes languages (and associated semantics) to allow the developer to encode drivers and emulators in the framework.

Driver. The language design and semantics of the driver closely follow the Coq formalization of drivers (section 4.2.2). Because the language is meant to be used for writing drivers for circuits, instead of supporting a general-purpose `Call`, it instead exposes `set-input`, `tick`, and `get-output` directly. Furthermore, instead of supporting the full generality of `Choose`, the language has a `yield` primitive, to model the host yielding and letting the circuit execute for an arbitrary number of cycles—this is useful when modeling asynchronous protocols such as UART.

Figure 6-2 shows an example of a driver for a circuit that communicates over UART: this driver describes how "spec-level operations" (i.e., operations at the level of the assembly-level implementation) translate to wire-level I/O. Specifically, the driver describes how the `handle` function, which takes as input a fixed-length sequence of bytes and returns as output a fixed-length sequence of bytes, is implemented in terms of `set-input`, `tick`, and `get-output`. This driver also demonstrates the use of the `yield` primitive, modeling the host waiting for an arbitrary number of cycles between sending bytes, as is allowed in the asynchronous UART protocol.

Emulator. The language design and semantics of the emulator closely follow the Coq formalization of emulators (section 4.2.3).

```
(define (handle command-bytes)
  (send-bytes command-bytes) ; send command
  (recv-bytes RESPONSE-LEN)) ; read response

(define (wait-until-clear-to-send)
  (while (get-output 'rts))
    (tick))) ; wait a cycle

(define (send-bit bit)
  (set-input 'rx bit)
  (for ([i (in-range BAUD-RATE)])
    (tick)))

(define (send-byte byte)
  (wait-until-clear-to-send)
  (send-bit #b0) ; send start bit
  ;; send data bits
  (for ([i (in-range 8)])
    (send-bit (extract-bit byte i)))
  (send-bit #b1)) ; send stop bit

(define (send-bytes bytes)
  (for ([byte bytes])
    (yield) ; wait for arbitrary number of cycles
    (send-byte byte)))
```

Figure 6-2: A code snippet from a driver for a circuit that communicates over UART. The function corresponding to a spec-level operation (`handle`) is shown in blue. Driver-language primitives are in red.

Parfait provides a strategy that is effective for constructing emulators for the circuit level of abstraction. The emulator runs a fresh instance of the circuit, with dummy data. The emulator does not have access to the data in the real circuit, in particular the read-write persistent memory, but the structure of the circuit and the code in the ROM is common knowledge. The emulator carefully watches the internal state of its instance of the circuit: when the circuit reaches the commit point of an operation, the emulator reads input data out of its circuit's state and translates it into a spec-level input, makes a query to the specification, and injects the result back into its circuit's state, so that the (future) output behavior of its circuit instance matches that of the real circuit. For HSMs that follow the Parfait design, the commit point is the (cycle-level) commit point of the `store_state` function in the

system software (figure 3-4).

Because all of our emulators follow this construction, for convenience, the language semantics has built-in support for constructing a fresh instance of the circuit, reading and writing the instance's state, and executing it for a clock cycle.

6.4 Proof

The developer supplies the driver, an emulator, and a refinement relation relating assembly-level state (a sequence of bytes) to implementation-level state (registers and memories in the circuit). Parfait HSM implementations use a simple journaling strategy for crash safety, using a flag word and toggling between two regions of persistent memory. Figure 6-3 shows a refinement relation for such an implementation.

$$\begin{aligned}\text{Inv}(\text{impl}) \wedge \text{spec} = \ & \texttt{if } \text{impl.mem}[0] == 0 \\ & \texttt{then } \text{impl.mem}[1 : \text{STATE\_LEN} + 1] \\ & \texttt{else } \text{impl.mem}[\text{STATE\_LEN} + 1 : 2 \times \text{STATE\_LEN} + 1]\end{aligned}$$

Figure 6-3: An example of a refinement relation between assembly-level implementation (the spec at this level) and circuit impl. *impl.mem* refers to the persistent memory of the implementation. *Inv* is an invariant on circuit state that holds in between spec-level operations, not shown here.

Using this, Knox proves functional simulation (figure 4-10) and physical simulation (figure 4-11) between the assembly-level implementation and the circuit. Knox includes techniques to handle the challenges that arise when applying symbolic execution and SMT solvers to proving functional and physical simulation. In functional simulation proofs, Knox handles the nondeterminism of `yield` in drivers by automatically finding fixed points (section 6.4.1). In physical simulation proofs, Knox supports reasoning about unbounded-length inputs using an approach we call *guided symbolic model checking* (section 6.4.2). For both, Knox includes performance optimizations: automatic *state synchronization* between assembly and circuit to make SMT queries tractable (section 6.4.3), and support for untrusted *hints* supplied by the proof developer to optimize performance (section 6.4.4).

6.4.1 Nondeterminism

Knox verifies the functional simulation property using symbolic execution of the driver-language program against the HSM implementation, comparing the execution of the driver/cir-

cuit with the assembly-level implementation. However, symbolic execution cannot directly handle the nondeterminism of `yield`, which has the semantics of the driver waiting for an arbitrary number of cycles while the HSM runs.

Knox addresses this by finding a fixed point of the circuit's step function at every yield point. During symbolic execution, the circuit's state is a symbolic term. Stepping the circuit produces a new symbolic term, and so on. At yield points, Knox computes a set of symbolic terms such that the set is closed under the circuit's step function, and it forks symbolic execution for each term in the set.

Closure is defined in terms of symbolic state subsumption. A symbolic term t under a path condition p, written as $t|p$, can be thought of as representing a set of concrete values, $[\![t|p]\!]$, the set of values that t can evaluate to for all possible assignments satisfying p of values to t's symbolic variables. A term t_1 under path condition p_1 is subsumed by a term t_2 under path condition p_2, written as $t_1|p_1 \subseteq t_2|p_2$, if $[\![t_1|p_1]\!] \subseteq [\![t_2|p_2]\!]$. For a set S of symbolic terms paired with path conditions, let $[\![S]\!] = \{[\![t|p]\!] \colon t|p \in S\}$. Finally, call S a fixed point of the step function if $\forall x \in [\![S]\!], \text{step}\,(x) \in [\![S]\!]$.

Knox includes an efficient algorithm for subsumption checks, and fixed points are found through iteratively calling the step function on the symbolic circuit state to build up a set of symbolic terms. Once a fixed-point S is found, symbolic execution proceeds for each of the $t|p \in S$, similar to how branching produces multiple paths to be checked.

Left unchecked, multiple `yield`s can result in an exponential number of cases to check, analogous to the problem of branching resulting in path explosion in symbolic execution. For this reason, Knox uses untrusted `merge` hints in the driver at points where some branches could be merged together. At merge points, Knox uses subsumption checks to automatically find a smaller set of symbolic terms $|S'| \leq |S|$ that still represent all the concrete values included in the original, $[\![S]\!] \subseteq [\![S']\!]$, which addresses case explosion.

6.4.2 Unbounded-length inputs

In Knox, emulators can be symbolically executed with black-box query access to a functional specification. Unlike the functional simulation property which considers a single (spec-level) input, the physical simulation property considers an arbitrary-length sequence of (wire-level) inputs, so Knox can't prove the physical simulation property in the same way. Symbolic execution could verify this property for a fixed-length input, but it cannot directly handle arbitrary-length input.

The standard approach to handling arbitrary-length inputs is to write down an inductive invariant and reason about one step at a time. This approach does not work for large circuits

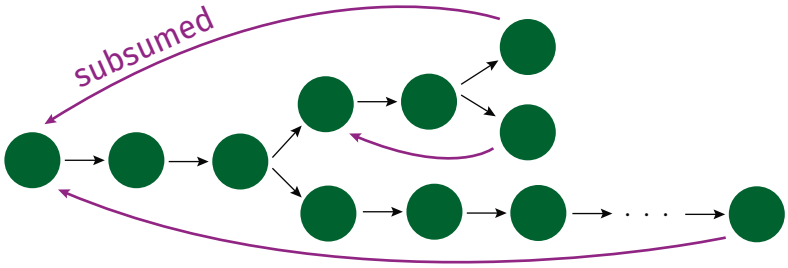


Figure 6-4: An illustration of *guided symbolic model checking* exploring a state space. Each green circle is a symbolic term representing a set of states. Black arrows show STEP invocations and purple arrows show SUBSUMED invocations.

because of the infeasibility of manually writing down the inductive invariant. It would have to include an invariant of circuit execution, capturing which states are reachable and which are not, and it is infeasible to manually write down exactly how CPU microarchitectural registers, peripheral registers, RAM state, etc. are related to each other at every cycle of execution of the software.

Instead, Knox uses an approach that we describe as *guided symbolic model checking*. At a high level, Knox uses a model-checking-style approach to start from the initial states of the circuit and emulator in the definition of physical simulation, explore all reachable states, and ensure that the circuit's behavior matches the emulator's behavior and the recovery condition holds at every step. Exploration starts out at a circuit state c_1, an emulator state e_0 (the initial emulator state), and functional spec state f_1, where both f_1 and c_1 are symbolic terms, and a refinement relation R is assumed to relate f_1 and c_1. Knox can step the circuit and step the emulator, given the same symbolic input, and check that their outputs match. Knox repeats this process until it has explored all reachable states.

This model-checking process involves guidance from the developer in the form of a proof script. Knox provides two primitives that allow the developer to guide exploration of the state space:

- STEP steps the circuit and the emulator/spec (with the same symbolic input) for one cycle and verifies the output equivalence and recovery properties for that single cycle
- SUBSUMED checks that the state currently under consideration is subsumed by a state that was explored earlier, "tying the knot" and finishing a branch of the exploration

Figure 6-4 illustrates how STEP and SUBSUMED let the developer guide the model checker to explore the state space. In addition to these primitives, the developer uses additional *hints* (section 6.4.4) to safely manipulate symbolic terms and help the model checker efficiently explore the state space.

An alternative view of this process is that it incrementally builds up the induction hypothesis that would have been used in an induction-based approach. Once model checking has explored all reachable states, it has visited a set of states S that includes the initial circuit/emulator/spec state where R holds, and the set S has the property of being closed under the circuit/emulator step functions, and the property of matching outputs for a single cycle holds for every state in S. The induction hypothesis is that the state is contained in S.

The proof script is untrusted, and Knox checks that the state space is fully explored. At worst, an incorrect proof script can result in poor performance or Knox reporting that the state space has not been fully explored.

6.4.3 State synchronization

A key challenge for Knox's functional-physical simulation proof is that, in practice, SMT solvers are unable to prove the equivalence of assembly-level and circuit-level executions after many cycles of execution. In particular, for sophisticated applications like Parfait's ECDSA HSM, the app assembly code can take tens of millions of instructions to execute in the SoC, corresponding to a single "step" of the assembly-level specification state machine (section 6.1). The functional-physical simulation proof involves showing that the app assembly transforms the state/command buffers in a way that corresponds with how the SoC hardware updates its buffers. While Knox can symbolically execute both the spec and the implementation, and express this correspondence, SMT solvers are unable to directly prove equivalence of how these buffers are transformed, because the symbolic expressions describing the two are extremely complicated, and not identical.

Instead of waiting to prove equivalence of final states/outputs at the end of executing an entire HSM operation, Knox uses a strategy of incrementally executing the spec and periodically synchronizing the spec and implementation. Although the app assembly level is modeled as a whole-command state machine that executes a command in a single transition, Riscette computes that transition by symbolically executing instruction-by-instruction. Knox makes use of this per-instruction stepping to simplify equivalence checking. When the hardware is in the middle of executing the `handle` function, there's a close correspondence between the hardware's cycle-by-cycle execution and single-stepping through CompCert Asm-level instructions.

To do this synchronization, Knox uses a mapping from CompCert Asm abstract machine state to hardware-level state (registers and memories) provided by the platform developer. During the proof, Knox applies this mapping to line up the states and attempts to prove

equivalence component-wise. If the equivalence check succeeds, it replaces both symbolic expressions with the same symbolic variable. This way, the solver does not get a large hard-to-prove query at the end of execution. Instead, it proves many simpler equivalences throughout the execution. Knox has built-in heuristics for when to synchronize, and for what should be synchronized in which situations.

Knox uses a "best effort" strategy for synchronization. Occasionally the developer-provided mapping or heuristics for alignment are not quite right and an equivalence check fails. When this happens, Knox does not unify the symbolic expressions. Instead, it continues symbolically executing and tries to check for equivalence later. The result is that the solver might end up with a slightly harder query at the next synchronization point.

Synchronization for the Ibex SoC. The remainder of this section describes in more detail the mapping and synchronization heuristics, using the platform mapping for the Ibex-based SoC used in one of our case studies (chapter 8) as an example.

State correspondence. The platform developer supplies the correspondence between the CompCert Asm abstract machine state (fixed by the framework) and the hardware (implemented by the platform developer). As illustrated by examples in figure 6-5, the platform developer supplies:

- **Register mapping**: for each architectural register in abstract state, what is the Verilog register to which it corresponds? For example, in the Ibex processor used in the case study HSM, `x1` corresponds to `cpu.gen_regfile_ff.register_file_i.g_rf_flops[1].rf_reg_q`.
- **Pointer mapping**: for each concrete pointer (e.g., `0x20000008`), what is the Verilog memory and index to which it corresponds? For the Ibex SoC, this pointer maps to Verilog memory `ram.ram`, with an offset of 2 words.
- **Encoding of next RISC-V instruction**, including whether or not this is valid (it may be invalid if the execute stage of the processor pipeline is stalled). This is what Knox uses to synchronize execution between Asm and hardware (rather than mapping program counter addresses). In our case study SoC, the instruction about to be executed by the ID/EX stage of the pipeline is found in `cpu.u_ibex_core.if_stage_i.instr_rdata_id_o`, and the signal that indicates whether this instruction is valid is found in `cpu.u_ibex_core.if_stage_i.instr_valid_id_q`.

These mappings are specified in about 10 lines of proof code.

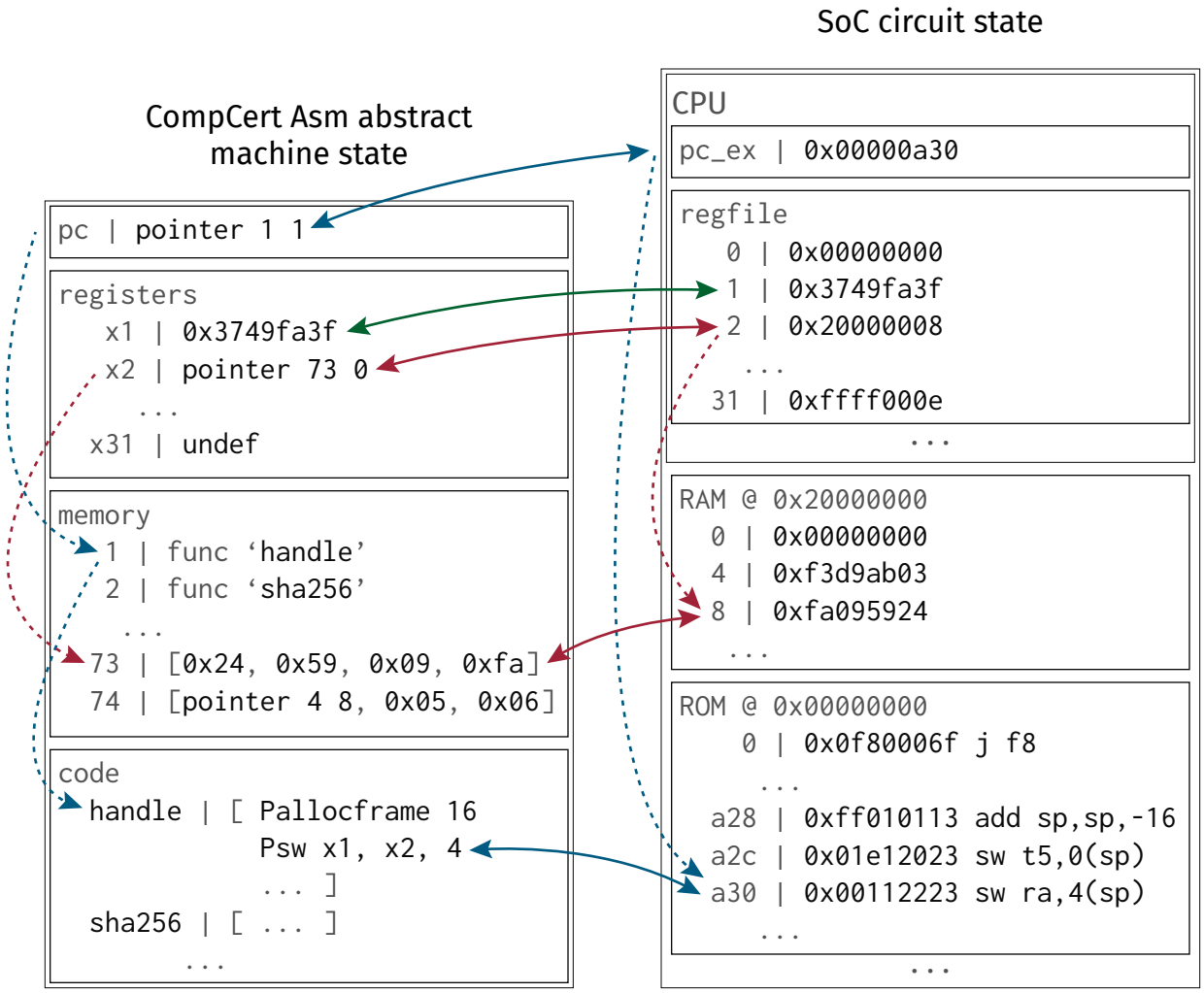


Figure 6-5: Correspondence between Asm machine state and circuit state. The figure illustrates an example register mapping in green, pointer mapping in red, and instruction mapping in blue.

Data type correspondence. The Asm model has its own data type of values that are stored in registers/memory: bitvectors (32-bit words in registers, 8-bit bytes in memory), pointers (there is a native pointer type in CompCert Asm, they are not just represented as ints), and undef. In the SoC, everything is a bitvector. Knox synchronizes spec state registers with circuit state registers as follows:

- **Bitvectors**: this represents data, and the Asm and hardware are generally lined up, so these values should be equal; Knox invokes an SMT solver to prove that the spec register and implementation register have the same value, and replaces both with the same symbolic variable.
- **Pointers**: when a spec register has a pointer value, Knox guesses that the value in

the hardware register is also a pointer, and points to flat memory. In this case, Knox synchronizes the *contents* of the memories, and leaves the pointers in the registers untouched (a CompCert `pointer` in the spec, and a 32-bit bitvector in the implementation). Synchronizing the memory contents is similar to synchronizing registers. Knox knows the bounds of the allocation thanks to CompCert's structured memory model. Knox uses the SMT solver to prove the chunk of memory equal between spec and implementation, doing this byte-by-byte, and synchronizing values that are equal.

- **Undef**: when a spec register is undefined, Knox leaves the implementation register as-is.

All the examples in figure 6-5 are shown with concrete values, but when used in Knox for verification, most of the registers and memory contain symbolic expressions.

When to synchronize. Because synchronization involves multiple SMT queries, it is too expensive to do at every cycle; moreover, some CompCert Asm instructions take multiple cycles to execute in SoC hardware. Instead, Knox uses a number of heuristics to decide when to synchronize, as shown in figure 6-6. Knox watches the instruction being executed in the spec machine, and the instruction being executed in the implementation machine (thanks to the developer-written mapping that provides the next-executing instruction). Knox steps each machine up until the next synchronization point, and then does the synchronization. In some cases, there's direct correspondence between Asm instructions and hardware instructions, in other cases, it's a more complex mapping. Figure 6-6 shows CompCert Asm instructions and their corresponding RISC-V assembly instructions or instruction sequences. Knox synchronizes either register values only (only the bitvectors, not the memory contents pointed to by registers containing pointer values), buffer values only, or both, depending on the kind of synchronization point.

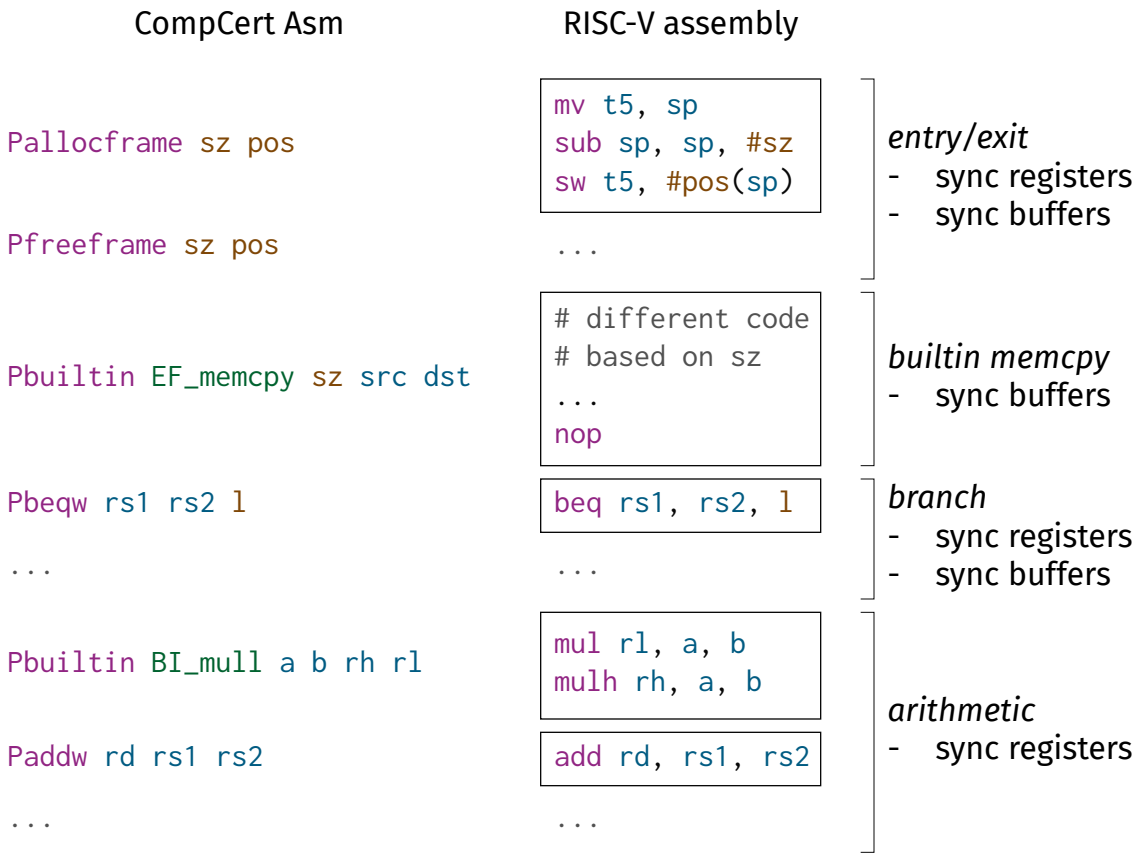


Figure 6-6: Knox synchronization points and corresponding actions.

6.4.4 Hints

In both functional simulation proofs and physical simulation proofs, relying only on hybrid symbolic execution and state synchronization still results in an explosion in term size and queries that make the SMT solver time out, related to the execution of the system software by the circuit (execution of `handle` is not a problem thanks to state synchronization).

Knox addresses this with untrusted (solver-checked) human guidance called *hints*. Knox has 8 primitive hints:

- CASE-SPLIT performs case analysis
- CONCRETIZE invokes the solver to prove that a symbolic term is concrete and replaces it with the concrete value
- OVERAPPROXIMATE replaces a term with a fresh variable
- WEAKEN weakens the current path condition
- REPLACE rewrites or simplifies terms
- REMEMBER, SUBSTITUTE, and CLEAR effectively allow marking terms as opaque to symbolic execution and substituting in their values later

Furthermore, Knox supports writing higher-level *tactics* that can reflect on the current state of symbolic execution and invoke primitive hints (or other tactics). A tactic might, for example, analyze the state of the circuit to determine if a CPU is about to branch, and in that situation, it can invoke a CASE-SPLIT hint with the appropriate cases constructed based on analyzing the symbolic circuit state.

All invocations of hints are verified by the Knox framework with a call to the SMT solver when necessary. Hints are untrusted: at worst, hints can be incorrect and fail (e.g., when attempting to replace a term with an unequal term), which will result in an error message to the user, or the given hints can be inadequate to ensure good performance, in which case verification will be slow or fail to terminate.

6.5 Discussion

The Parfait approach and Knox framework aim to minimize developer effort to prove IPR. Knox's use of symbolic execution and SMT solvers along with Knox's performance optimizations allow for bridging the gap directly from assembly-level execution to circuit-level execution without the need to introduce additional levels of abstraction. Knox doesn't require that the developer write an ISA specification, or that the developer verify a processor to be correct against an ISA specification; this saves considerable effort compared to prior work on hardware/software verification [40, 42]. Furthermore, thanks to automation, Knox is able to support the use of off-the-shelf processors that were not purpose-built for verification, as we demonstrate in chapter 8.

6.6 Limitations

Like the software verification assumes that the KaRaMel C semantics coincide with the CompCert C semantics (section 5.5), Parfait's hardware verification assumes that the CompCert RISC-V Asm semantics coincide with the Riscette semantics. We have written the Riscette semantics by closely following the CompCert Coq code, so we expect these to align.

Chapter 7

Implementation

The Parfait framework consists of the three components shown in table 7-1. We have open-sourced all of these components, along with HSMs verified using Parfait (table 1-1).

Table 7-1: Components that comprise the Parfait framework. The Knox framework includes Rosys and Riscette. Lines of code counts include tests (for Knox) and proofs (for IPR theory).

Component	Language	Lines of code
IPR theory	Coq [103]	3,000
Starling framework	F* [102]	100
Knox framework	Racket [43] / Rosette [104]	8,000

Software toolchain. A developer using Parfait writes app specifications in F* and app implementations in Low*. The developer uses the Starling framework to verify the software (chapter 5). F* relies on the Z3 SMT solver [79] to discharge verification conditions. Parfait uses the KaRaMeL compiler [94] to extract Low* to C and the CompCert compiler [67] to compile C code to RISC-V Asm. Parfait forks the CompCert compiler and adds 450 lines of code to append a `nop` instruction to compiler-expanded memcpy builtins to aid in synchronization and to dump the RISC-V Asm AST to a `.json` file before emitting the final `.s` file.

Hardware toolchain. The developer writes the system software in a combination of C and assembly code, and they write the hardware implementation in Verilog. The developer uses Knox to verify the hardware (chapter 6). To achieve good performance, Knox implements a number of techniques beyond those described in this thesis, including symbolic state serialization, term subsumption, fixpoint finding, state merging, a new algorithm for

symbolic subsumption checking based on a disjoint-set data structure, and a equivalent-up-to-renaming query cache in front of the SMT solver that reduces queries by 90%.

Parfait uses GCC [101] to compile the system software and link it with the app assembly to form the complete firmware binary. Parfait uses the Yosys [113] synthesis tool and its SMT-LIB backend to dump a `.smt2` file. To interpret the app assembly as a state machine, Riscette reads the CompCert-generated Asm AST JSON dump and produces a deep embedding in Rosette. Riscette implements a single-steppable executable semantics for this deeply-embedded language. To interpret the circuit as a state machine, Rosys reads the Yosys-generated SMT-LIB representation, and using a collection of Racket macros, transforms it into a shallow embedding in Rosette.

Knox builds on Rosette, which relies on SMT solvers to discharge verification conditions. Rosette supports multiple backends; Parfait uses Z3.

Trusted computing base. The trusted computing base (TCB) of the Parfait framework consists of: (1) the Coq definition of IPR, (2) the Starling framework's F* encoding of the lockstep property (needs to match the Coq definition), and (3a) the Knox framework's Riscette semantics (needs to match CompCert), (3b) the Knox framework's Rosys conversion of a circuit (Verilog/software) into Rosette, and (3c) the Knox checker that verifies that the functional-physical simulation property is satisfied (needs to encode/check the Coq definition).

Parfait also inherits the TCB of the verification tools it uses: the TCB of Coq (including the Coq proof checker kernel), the TCB of F* (including the Z3 SMT solver), and the TCB of Rosette (including the Z3 SMT solver). Parfait inherits the TCB of KaRaMeL and CompCert because the overall IPR proof relies on the correctness of these compilers. GCC is *not* part of the TCB.

Chapter 8

Verifying HSMs using Parfait

This chapter qualitatively demonstrates that Parfait enables verification of realistic HSM implementations, ensuring they are free from leakage bugs. We first describe the HSMs that we have developed on top of Parfait as case studies, and then we discuss how Parfait catches security bugs that an HSM implementation may have.

8.1 Case studies

We implement HSMs for two applications: ECDSA certificate signing and password hashing. The software builds on top of specifications, implementations, and proofs for cryptographic algorithms from the HACL* [122] library. We run these applications on two hardware platforms: one based on the Ibex processor [72] from the OpenTitan project [73] and one based on the PicoRV32 processor [114].

Application 1: ECDSA certificate signing. An ECDSA certificate-signing app, described in figure 3-1, was the running example through this thesis. The complete F* specification (figure A-1) is about 40 lines of code. The specification uses the HACL* verified cryptography library, re-using its specifications of the HMAC-SHA256 and ECDSA-P256 algorithms. At initialization time, the user calls `Initialize` to configure the HSM with an ECDSA signing key and the key for the HMAC pseudorandom function (used for generating signing nonces). The HSM also exposes a `Sign` command that takes a message as input and returns a signature on it. There is no method to retrieve the signing key or the PRF key from the HSM.

The Low* implementation of the `handle` function along with the IPR proof consists of 500 lines of code, not including library code/proof used from HACL*. The complete implementation extracts to 2,000 lines of C code.

For Knox verification to go through, the app implementation must not leak information through timing. HACL* code is already intended to be constant time, and verification confirmed that library functions indeed execute in constant time on our hardware, so we did not need to modify any library code. In the implementation, we needed to take care to ensure that other operations do not leak information through timing. For example, the `ecdsa_p256` spec/implementation return an error if the nonce or signing key is not less than the prime field order, and also return an error if $r = 0$ or $s = 0$ in the signature algorithm. The HSM spec does not distinguish between any of these errors (the caller just receives `Signature None`), and so IPR requires that the implementation also not reveal any information beyond this. The implementation computes a signature unconditionally, and then applies a mask to the buffer (`0xff` or `0x00`) based on whether all the checks passed or not; this way, the entire computation is constant-time.

Application 2: Password hashing. As a second case study, we implemented a password-hashing HSM. Figure 8-1 shows the core of the specification. The complete F* specification (figure A-2) is about 30 lines of code. The HSM implementation is a thin wrapper around the HMAC/Blake2S implementation from HACL*.

```
let step (st:state_t) (cmd:command_t): state_t & response_t =
  match cmd with
  | Initialize secret ->
    { secret = secret }, Initialized

  | Hash message ->
    let digest = hmac Blake2S st.secret message in
    st, Hashed digest
```

Figure 8-1: The transition function from the specification for a password-hashing HSM. The definition of `hmac` is used directly from HACL*.

Hardware platform 1: Ibex-based SoC. Our main hardware platform is a stripped-down version of the OpenTitan open-source hardware root of trust. The CPU is the Ibex, a two-stage pipelined RISC-V processor written in 13,000 lines of SystemVerilog. This is a pre-existing open-source CPU not purpose-built for verification. The HSM uses a UART peripheral to communicate with the host machine. In addition, the SoC contains a RAM, a ROM,

and ferroelectric RAM (FRAM) as persistent memory. Aside from the CPU, the rest of the components are written in 500 lines of Verilog.

The system software for this platform—main loop, drivers, I/O code, and persistence code—is written in 300 lines of C and assembly.

Our implementation makes two changes to the CPU: we remove async resets because Rosys doesn't support them, and we replace the Ibex's multiplier with a simple full-width Verilog multiplication of operands.

We changed the multiplier such that it does not affect circuit-level timing, but the change speeds up verification. Some hardware designs, such as the Ibex, include an implementation of a multiplier made up of smaller multipliers. The original implementation uses the Verilog $*$ operator on 16-bit operands, and it uses several such multipliers and several cycles of execution to implement a 32-bit multiply. In Knox execution, this results in a slightly larger gap between assembly-level execution and circuit-level execution, which slows down verification. A common practice when writing hardware implementations is to use the Verilog $*$ operator on full-width operands and leave it to the synthesis tool (e.g., Yosys) to infer an optimal implementation of the multiplier. This often results in better FPGA code, because, for example, the synthesis tool can implement the multiplier using the largest DSP blocks available on the hardware target. For these reasons, we swapped the Ibex's original multiplier with a full-width multiplier.

Hardware platform 2: PicoRV32-based SoC. We also verify and run the case-study applications on a second CPU, the PicoRV32, which is a size-optimized RISC-V CPU. As with the Ibex, we removed async reset from the implementation. This platform uses the same system software as the Ibex-based SoC. We use the PicoRV32 SoC to quantify the development effort required to port to a new platform, in section 9.1.

8.2 Security discussion

Attacks handled by Parfait. Parfait proves IPR between the app's functional specification and the SoC running the app and system software, so the verification process catches all possible bugs that are captured by IPR: hardware bugs, software bugs, and timing side channels. Here, we give several examples of possible bugs, and explain what part of the verification process prevents those bugs.

- Software logic bug (e.g., integer overflow leading to nonce reuse): Starling will catch this when verifying the postcondition for the Low* implementation, which ensures

that the output and final state of the implementation match the output and final state of the specification.

- Software-level timing leak from branching on a secret: although this is a "software bug," Parfait does not introduce any notions of timing until the SoC level—earlier level execute HSM operations in a single state-machine step. Knox will catch this because the emulator's behavior will not match the circuit's behavior.
- Compiler-introduced timing leakage: if a compiler optimization introduces a timing bug, such as returning early from `memcmp`, Knox will catch this bug at the SoC level, just as in the above example.
- Hardware-level timing leak from a variable-latency arithmetic instruction executed on secret data: Knox will catch this, as the emulator's behavior will not match the circuit. The emulator doesn't have access to the secret data, so it computes over dummy data instead; the real circuit will take a different amount of time, not matching the emulator.
- Buffer overflow or use-after-free: Low* verification prevents these memory safety bugs. In particular, type checking in the `Stack` effect ensures memory safety.
- Stack overflow: Parfait uses an abstract memory model up to and including the app assembly level (including in the Riscette semantics), with an unbounded size stack and frames addressed by mathematical integers. The SoC level introduces a bounded stack. To rule out stack overflows, Knox relates the SoC with a bounded stack to the app assembly with an unbounded stack and shows that they correspond.
- I/O peripheral driver bug in system software (e.g., incorrectly encoding the output or setting the wrong UART baud rate): Knox will catch this bug when verifying functional simulation.
- Pipeline hazard in CPU implementation: caught during Knox verification. If this occurs during the execution of the app code, this will show up as a mismatch between the app assembly execution (which uses the Riscette instruction-by-instruction execution semantics) and the hardware execution.

Out-of-scope attacks. Parfait does not handle physical attacks on the HSM. While Parfait's threat model allows arbitrary digital I/O, Parfait assumes that the adversary cannot violate the digital abstraction, such as sending 5V signals to a circuit expecting 3.3V logic. Parfait

also does not handle physical side channels such as radiation [4], temperature [58], and power [75]. Finally, Parfait cannot handle attacks that exploit mistakes in the specification.

Chapter 9

Evaluation

This chapter answers two key questions:

- Section 9.1: What is the developer effort to verify HSMs with Parfait?
- Section 9.2: What is the performance of HSMs verified with Parfait?

9.1 Developer effort

Table 9-1 summarizes the lines of code required to specify and implement each of the four HSMs from chapter 8 (two apps on two platforms). As the table shows, verification in Parfait relates a hardware/software stack that takes over ten thousand lines of code to implement to a state-machine-style application specification that comprises only tens of lines of code.

Table 9-1: Lines of code for case studies. The specification covers the hardware and software stack. Spec LoC counts the lines the HSM developer writes (and doesn't include HACL*, Low*, or F* library/language code). Driver LoC counts the total lines of driver code across the software and hardware levels.

HSM	**Spec**	**Driver**	**Platform**	**Implementation**	
				Software	Hardware
ECDSA signer	40 LoC	100 LoC	Ibex	2,300 LoC	13,500 LoC
			PicoRV32	2,300 LoC	3,000 LoC
Password hasher	30 LoC	100 LoC	Ibex	1,000 LoC	13,500 LoC
			PicoRV32	1,000 LoC	3,000 LoC

Table 9-2 shows, for each case-study app, the number of lines of proof required to prove the lockstep property between the app's F* specification and its Low* implementation. We co-developed the ECDSA-signer app with Starling, so we cannot report the verification

effort for the ECDSA-signer app on its own. Once the Starling framework was in place, we implemented the password hasher app as a second case study. Implementing and verifying this new app took two hours. Machine verification of these proofs runs in less than a minute.

Table 9-2: Software verification effort. Verifying a second application with Parfait required only two additional developer-hours of effort.

App	Proof	Dev time
ECDSA signer	500 LoC	-
Password hasher	200 LoC	Δ 2 hours

Table 9-3 shows the number of lines of proof required to verify the two platforms with Knox. We co-developed the Ibex platform with the approach and framework; as a second case study, we modified the platform and swapped the Ibex CPU with the PicoRV32 CPU. Porting to this new platform took two hours and involved writing 10 lines of new proof to map PicoRV32 CPU state to CompCert Asm abstract machine state, while the rest of the proof remained unchanged, because the system software and rest of the hardware platform (such as peripherals) remained unchanged.

Table 9-3: Hardware verification effort and verification time, showing both total wall-clock time (single threaded) and symbolic circuit simulation speed. Porting the platform to use a different CPU took just two hours of developer time and 10 lines of changed proof code.

	Proof size (LoC)				Verification			
					ECDSA signer		Password hasher	
Platform	Emulator	Hints	Mapping	**Dev time**	Time	Cycles/s	Time	Cycles/s
Ibex	50	250	10	-	80 hrs	304	0.10 hrs	289
PicoRV32			10	Δ 2 hours	100 hrs	671	0.14 hrs	588

The "Verification" columns of table 9-3 show Knox's verification performance for each combination of app and platform, benchmarked on a machine with an Intel Xeon Gold 5420+ processor. It is possible to swap in new apps and hardware platforms with no changes required to proof code on either side. After such a change, the only requirement is to run Knox on the new software/hardware combination.

Knox verification is highly automated (section 6.4), though it can take up to 100 core-hours of computation to verify our most complicated application. Verifying the ECDSA HSM requires symbolically executing the hardware for tens of millions of cycles and issuing hundreds of millions of SMT queries, leading to the long verification time. In contrast, verifying the password hasher takes only a few minutes because the code is much simpler

and only runs for hundreds of thousands of cycles. Verification throughput (cycles per second) is higher for the PicoRV32, because simulating each SoC execution cycle is faster on the simpler hardware. Total verification wall-clock time is higher for the PicoRV32 because apps require more cycles to run on the non-pipelined processor, requiring Knox to simulate more SoC execution cycles.

Development cycle. If the Low* implementation has a timing bug when executed by the circuit, Knox verification will fail with a mismatch between the real circuit's execution and the emulator's execution. Usually, this will be caused by secret data (on which timing should not depend) entering the control state of the circuit; Knox can print out user-requested debugging information such as the program counter when this occurs. From this, the user can look at the assembly listing and then determine the C code corresponding to the program counter value. This will generally reveal code that does something leaky, such as `if (secret) { ... }` or `x / secret`. Going from the C code to the source Low* is easy because a design goal of Low* is to translate straightforwardly to C.

Because hardware verification takes hours, one trick we use to identify failures faster is reducing loop bounds. For example, if the implementation contains code that does `for (int i = 0; i < 80; i++)`, we can manually change the loop bound from 80 to 2 in the C code and try verifying that the hardware securely executes this code. Even though this is no longer computing the "correct" functionality, timing leakage is usually not affected by reducing loop bounds in this way, so we can catch issues faster. We revert to the original code for the final verification.

9.2 Performance

Parfait's use of the CompCert compiler introduces run-time overhead, because CompCert emits less performant code than GCC does. Table 9-4 measures this performance penalty, showing that several commercial HSMs have ECDSA-signing throughput that is within $12\times$ the throughput of HSMs built with Parfait. This is not an apples-to-apples comparison — the different HSMs use different CPUs, have different ISAs, run at different clock speeds, and run different software.

For some applications, such as client-side HSMs, run-time performance may not be the primary concern. For example, a U2F token that takes 1 second to sign users in, but has a formal proof of correctness/security/non-leakage, is valuable. For server-side applications, performance is important in some but not all applications. Parallelization across multiple low-cost HSMs could be used to overcome the performance gap. For example, Let's Encrypt

Table 9-4: Run-time performance comparison of HSMs. The Ibex processor is clocked at 100 MHz, which is the OpenTitan reference clock.

HSM	Compiler	ECDSA sig/s	Improvement
Parfait ECDSA/Ibex	CompCert -O1	1.1	-
	GCC -O2	8.1	7x
Nitrokey HSM 2 [85]		12.5	11x
YubiHSM 2 [119]		13.7	12x

uses a fleet of HSMs for certificate signing [1]. With Parfait's ECDSA HSM, for example, a certificate authority could instantiate multiple HSMs with the same private key but different PRF seeds, and then run them all in parallel.

The primary run-time performance penalty of Parfait comes from CompCert; as the research community makes advances in verified compilers, a better CompCert would be a drop-in replacement. Furthermore, the HACL* developers consider CompCert as an important target, so future versions of HACL* could be tuned to aid CompCert in producing more performant code. This would also be a drop-in replacement.

Ibex is a realistic (e.g., used in OpenTitan) but relatively simple HSM CPU. It is similar in complexity to some production HSMs (e.g., many simpler devices use the ARM Cortex-M3 series processor). One could use a slightly more complex CPU, such as the biRISC-V [108] (a 6/7-stage pipelined CPU), that could provide better performance. Such CPUs are compatible with Rosys [14], so the Parfait approach likely could be applied to HSMs with such CPUs.

Chapter 10

Conclusion

This thesis develops an approach to verify security and leakage-freedom for hardware and software systems, relating an application specification to a circuit-level implementation. This thesis applies this approach to verifying hardware security modules to be correct, secure, and free of timing side-channel vulnerabilities. The contributions of this thesis include foundational theories (*information-preserving refinement*), a verification approach and framework (*Parfait*, *Starling*, and *Knox*), and several verified HSMs.

10.1 Discussion

Combining verification tools. Parfait uses three different verification tools—Coq, F*, and Rosette—for IPR theory, software verification, and hardware verification, respectively. This way, Parfait is able to use the best tool for each job. We heavily depended on the interactive proof mode of Coq when verifying the IPR proof strategies. Parfait HSMs build on top of the HACL* specifications, implementations, and proofs, and we benefited from the rich support for verifying C-like programs in Low* using the F* language. Knox performs symbolic execution of circuits for millions of cycles to verify IPR, and its performance critically depends on Rosette's hybrid symbolic evaluation. The hardware and software proofs use vastly different strategies, and it would have been near-impossible to use F* to verify the hardware implementation or use Rosette to verify the software.

Our experience with this approach has been positive: the results presented in this thesis would likely not have been possible to produce in a comparable number of person-years if we were restricted to using a single verification tool.

Circuit-level verification without a verified processor. Parfait sidesteps some of the challenges traditionally faced by hardware/software verification efforts. Parfait's goal is

to prove that a circuit satisfies an application-level specification, which doesn't necessitate having formalized intermediate specifications. In particular, this doesn't require an ISA specification and a verified-correct processor that provably implements this spec, such as the Kami processor [31]. Parfait does not verify that a processor is correct; instead, it verifies IPR, which captures that the processor correctly and securely executes the specific application being verified, though this does require running Knox to verify each software/hardware combination.

Verifying off-the-shelf real-world hardware. Parfait is able to verify IPR for implementations that use off-the-shelf processors that were not purpose-built for verification. This is thanks to a focus on verifying IPR with respect to the specification (not not verifying processor correctness in general), along with a high degree of automation in Knox through its reliance on symbolic execution and SMT solvers in addition to many performance optimizations.

10.2 Future work

Scaling up Parfait. This thesis verifies HSMs that use simpler embedded-class CPUs (such as the two-stage-pipelined Ibex processor used in the OpenTitan) and I/O peripherals (such as UART). Extending Parfait to more complex and powerful processors will require new techniques; for example, Knox's synchronization between assembly and circuit (section 6.4.3) would likely need to be extended to support superscalar or out-of-order processors. Supporting more sophisticated I/O peripherals and protocols, such as USB or PCIe, would likely require additional techniques to reconcile these stateful protocols with the strict non-leakage requirements of the IPR definition.

Parfait requires a deterministic cycle-precise description of behavior to verify circuits. This is why our HSMs use ferroelectric RAM (FRAM) as persistent memory, because its specification describes cycle-precise timing behavior (completing word-level reads/writes in a single cycle). Supporting components like flash memory, where datasheets do not pin down cycle-precise timing behavior, will require reconciling nondeterminism with IPR's strict non-leakage requirements.

Horizontal and vertical composition of IPR. Parfait leverages transitivity to make proofs of IPR for HSMs manageable. However, we have not defined or formalized a notion of horizontal or vertical composition of state machines that are verified with IPR. Compositionality could help enable:

- Verifying a host machine application, such as certificate authority server software, that builds on top of an HSM verified with IPR, such as the ECDSA signature HSM (vertical composition)
- Verifying an HSM with an SoC that contains some black-box components, such as a closed-source processor for which an IPR-style specification is verified by the CPU designer and assumed by the software developer (horizontal composition)

Hardware accelerators. Some HSMs use hardware accelerators for cryptography. Our work on verifying HSMs directly with Knox [13] supports hardware cryptographic accelerators but doesn't scale to sophisticated software. The Parfait approach presented in this thesis supports sophisticated software such as public-key cryptography but does not support hardware accelerators.

Extending the IPR definition. The definition of information-preserving refinement could be extended to capture non-leakage for HSM-like devices that:

- Support multiple applications, running one-at-a-time like Notary [11] or supporting true multitasking
- Use true random number generators, as explored in Karatroc [120]
- Have hardware in multiple clock domains, as explored in Kronos [77]
- Use a real-time clock, to implement functionality such as rate limiting
- Handle trusted input, such as the confirmation button of U2F tokens
- Communicate with peripherals such as a GPS
- Execute multiple operations concurrently or in parallel

10.3 Final remarks

This thesis develops a definition of non-leakage that captures that a circuit implementation leaks no more information than a specification allows. We believe that the definition of information-preserving refinement and the ideas developed in this thesis are applicable beyond hardware security modules. We hope that IPR can serve as the foundation of future security definitions focused on capturing non-leakage.

Approaches and tools for formal verification are growing increasingly capable of verifying complex and realistic systems. The techniques in this thesis show that it is possible to prove security properties by symbolically executing an entire system-on-a-chip, including its software, at the circuit level for millions of cycles. We hope that Parfait inspires practical applications of formal verification of hardware and software systems.

Appendix A

Code listings

```
let prf_key_t = lseq uint8 16
let signing_key_t = lseq uint8 32
let message_t = lseq uint8 32
let signature_t = lseq uint8 64

noeq type state_t = {
  prf_key:prf_key_t; prf_counter:uint64; signing_key:signing_key_t;
}

let state_init:state_t = {
  prf_key = create 16 (uint 0);
  prf_counter = uint 0;
  signing_key = create 32 (uint 0);
}

noeq type command_t =
| Initialize: new_prf_key:prf_key_t
            -> new_signing_key:signing_key_t
            -> command_t
| Sign: message:message_t -> command_t

noeq type response_t =
| Initialized: response_t
| Signature: maybe_signature:option signature_t -> response_t

let step (st:state_t) (cmd:command_t) : state_t & response_t =
  match cmd with
  | Initialize prf_key signing_key ->
    { prf_key = prf_key; prf_counter = uint 0; signing_key = signing_key },
    Initialized

  | Sign msg ->
    if uint_v st.prf_counter < maxint U64 then
      let data = uint_to_bytes_be st.prf_counter in
      let nonce = hmac SHA2_256 st.prf_key data in
      let sig = ecdsa_signature_agile NoHash _ msg st.signing_key nonce in
      { st with prf_counter = incr st.prf_counter }, Signature sig
    else
      st, Signature None
```

Figure A-1: The complete F* specification for the ECDSA certificate-signing HSM, including type definitions for the state, commands, and responses, and the description of the initial state. See chapter 3 for an overview of the role of functional specifications in Parfait.

```
let secret_t = lseq uint8 20
let message_t = lseq uint8 32
let digest_t = lseq uint8 32

noeq type state_t = {
  secret:secret_t;
}

let state_init:state_t = {
  secret = create 20 (uint 0);
}

noeq type command_t =
| Initialize: secret:secret_t -> command_t
| Hash: message:message_t -> command_t

noeq type response_t =
| Initialized: response_t
| Hashed: digest:digest_t -> response_t

let step (st:state_t) (cmd:command_t) : state_t & response_t =
  match cmd with
  | Initialize secret ->
    { secret = secret }, Initialized

  | Hash message ->
    let digest = hmac Blake2S st.secret message in
    st, Hashed digest
```

Figure A-2: The complete F* specification for the password-hashing HSM, including type definitions for the state, commands, and responses, and the description of the initial state. See chapter 8 for details on this HSM.

References

[1] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth Schoen, and Brad Warren. "Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, United Kingdom, Nov. 2019, pages 2473–2487.

[2] Martín Abadi. "Protection in Programming-Language Translations". In: *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pages 19–34.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "On the Power of Simple Branch Prediction Analysis". In: *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Singapore, Mar. 2007, pages 312–320.

[4] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. "The EM Side-Channel(s)". In: *Proceedings of the 2002 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Redwood City, CA, Aug. 2002.

[5] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. "Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices". In: *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Edinburgh, United Kingdom, Aug. 2010, pages 71–85.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, Oct. 2017, pages 1807–1823.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. "Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC". In: *Proceedings of the 23rd International Conference on Fast Software Encryption (FSE)*. Bochum, Germany, Mar. 2016, pages 163–184.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations". In: *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, Aug. 2016, pages 53–70.

[9] Apple, Inc. *Apple Platform Security*. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf. May 2024.

[10] ARM Limited. *ARM Cortex-M3 Processor Technical Reference Manual*. https://developer.arm.com/documentation/100165/latest/. Revision r2p1. Nov. 2016.

[11] 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 (SOSP)*. Huntsville, Ontario, Canada, Oct. 2019, pages 97–113.

[12] Anish Athalye, Henry Corrigan-Gibbs, M. 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 30th ACM Symposium on Operating Systems Principles (SOSP)*. Austin, TX, Nov. 2024.

[13] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Hardware Security Modules with Information-Preserving Refinement". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, July 2022, pages 503–519.

[14] Anish Athalye, M. Frans Kaashoek, Nickolai Zeldovich, and Joseph Tassarotti. "Leakage models are a leaky abstraction: the case for cycle-level verification of constant-time cryptography". In: *Proceedings of the 1st Workshop on Programming Languages and Computer Architecture (PLARCH)*. Orlando, FL, June 2023.

[15] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. "SideTrail: Verifying Time-Balancing of Cryptosystems". In: *Proceedings of the 10th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Oxford, United Kingdom, July 2018, pages 215–228.

[16] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. "SoK: Computer-Aided Cryptography". In: *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. Virtual conference, May 2021, pages 777–795.

[17] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. "Formal Verification of a Constant-Time Preserving C Compiler". In: *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*. New Orleans, LA, Jan. 2020.

[18] Jean-Baptiste Bédrune and Gabriel Campana. *Everybody be Cool, This is a Robbery!* https://donjon.ledger.com/BlackHat2019-presentation/. Aug. 2019.

[19] William R. Bevier, Warran A. Hunt Jr., J. Strother Moore, and William D. Young. "An Approach to Systems Verification". In: *Journal of Automated Reasoning* 5.4 (Dec. 1989), pages 411–428.

[20] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying constant-time implementations by abstract interpretation”. In: *Journal of Computer Security* 27.1 (2019), pages 137–163.

[21] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. “Vale: Verifying High-Performance Cryptographic Assembly Code”. In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada, Aug. 2017, pages 917–934.

[22] Dan Boneh and Glenn Durfee. “Cryptanalysis of RSA with Private Key d Less than $N^{0.292}$”. In: *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Prague, Czech Republic, May 1999, pages 1–11.

[23] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. “WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors”. In: *Proceedings of the 33rd USENIX Security Symposium*. Philadelphia, PA, Aug. 2024.

[24] Billy Bob Brumley and Nicola Tuveri. “Remote Timing Attacks Are Still Practical”. In: *Proceedings of the 16th European Symposium on Research in Computer Security*. Leuven, Belgium, Sept. 2011, pages 355–371.

[25] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *Proceedings of the 12th USENIX Security Symposium*. Washington, DC, Aug. 2003, pages 1–13.

[26] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. “Validation of Side-Channel Models via Observation Refinement”. In: *Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture*. Athens, Greece, Oct. 2021, pages 578–591.

[27] Ran Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. Las Vegas, NV, Oct. 2001, pages 136–145.

[28] 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)*. London, United Kingdom, June 2020, pages 913–926.

[29] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. “Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning”. In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, July 2022, pages 447–463.

[30] Dominic Chell. *Apple iOS Hardware Assisted Screenlock Bruteforce*. https://www.mdsec.co.uk/2015/03/apple-ios-hardware-assisted-screenlock-bruteforce/. Mar. 2015.

[31] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. “Kami: A Platform for High-Level Parametric Hardware Specification and its Modular Verification”. In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Oxford, United Kingdom, Sept. 2017.

[32] 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)*. Santa Barbara, CA, June 2016, pages 648–664.

[33] Filippo Cremonese. *Security Analysis of the Solo Firmware*. https://blog.doyensec.com/2020/02/19/solokeys-audit.html. Feb. 2020.

[34] *CVE-2004-0320*. https://nvd.nist.gov/vuln/detail/CVE-2004-0320. Sept. 2004.

[35] *CVE-2018-6875*. https://nvd.nist.gov/vuln/detail/CVE-2018-6875. Feb. 2018.

[36] *CVE-2019-18671*. https://nvd.nist.gov/vuln/detail/CVE-2019-18671. Nov. 2019.

[37] *CVE-2019-18672*. https://nvd.nist.gov/vuln/detail/CVE-2019-18672. Nov. 2019.

[38] *CVE-2021-31616*. https://nvd.nist.gov/vuln/detail/CVE-2021-31616. Apr. 2021.

[39] *CVE-2024-37880*. https://nvd.nist.gov/vuln/detail/CVE-2024-37880. June 2024.

[40] 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 Conference on Programming Language Design and Implementation (PLDI)*. Virtual conference, June 2021.

[41] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. “Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises”. In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2019, pages 73–90.

[42] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. “Foundational Integration Verification of a Cryptographic Server”. In: *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Copenhagen, Denmark, June 2024.

[43] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. “A Programmable Programming Language”. In: *Communications of the ACM* 61.3 (Mar. 2018), pages 62–71.

[44] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using verification to disentangle secure-enclave hardware from software”. In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017, pages 287–305.

[45] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. “An Empirical Study on the Correctness of Formally Verified Distributed Systems”. In: *Proceedings of the 12th ACM EuroSys Conference*. Belgrade, Serbia, Apr. 2017, pages 328–343.

[46] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ""These results must be false": A usability evaluation of constant-time analysis tools". In: *Proceedings of the 33rd USENIX Security Symposium*. Philadelphia, PA, Aug. 2024.

[47] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. "A Verified, Efficient Embedding of a Verifiable Assembly Language". In: *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*. Cascais, Portugal, Jan. 2019.

[48] Sivanarayana Gaddam, Atul Luykx, Rohit Sinha, and Gaven Watson. "Reducing HSM Reliance in Payments through Proxy Re-Encryption". In: *Proceedings of the 30th USENIX Security Symposium*. Vancouver, Canada, Aug. 2021, pages 4061–4078.

[49] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. "Time Protection: The Missing OS Abstraction". In: *Proceedings of the 14th ACM EuroSys Conference*. Dresden, Germany, Mar. 2019.

[50] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *Proceedings of the 3rd IEEE Symposium on Security and Privacy*. Oakland, CA, Apr. 1982, pages 11–20.

[51] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play Any Mental Game". In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. New York, NY, May 1987, pages 218–229.

[52] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof Systems". In: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*. Providence, RI, May 1985, pages 291–304.

[53] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, Nov. 2016, pages 653–669.

[54] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. Virtual conference, May 2021, pages 1868–1883.

[55] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-End Security via Automated Full-System Verification". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, Oct. 2014, pages 165–181.

[56] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. "ct-fuzz: Fuzzing for Timing Leaks". In: *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. Porto, Portugal, Oct. 2020, pages 466–471.

[57] Mason Hemmel, Jason Meltzer, Thomas Pornin, Keegan Ryan, Javed Samuel, David Wong, Rob Wood, and Greg Worona. *Android Cloud Backup/Restore*. https://research.nccgroup.com/wp-content/uploads/2020/07/Final_Public_Report_NCC_Group_Google_EncryptedBackup_2018-10-10_v1.0.pdf. Oct. 2018.

[58] Michael Hutter and Jörn-Marc Schmidt. "The Temperature Side Channel and Heating Fault Attacks". In: *Proceedings of the 12th Smart Card Research and Advanced Application Conference (CARDIS)*. Berlin, Germany, Nov. 2013, pages 219–235.

[59] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ""They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks". In: *Proceedings of the 43rd IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2022, pages 632–649.

[60] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. "Minerva: The curse of ECDSA nonces (Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces)". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.4 (2020), pages 281–308.

[61] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, Oct. 2009, pages 207–220.

[62] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2019, pages 19–37.

[63] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*. Santa Barbara, CA, Aug. 1996, pages 104–113.

[64] Ivan Krstić. *Behind the Scenes with iOS Security*. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf. Aug. 2016.

[65] Adam Langley. *Checking that functions are constant time with Valgrind*. https://www.imperialviolet.org/2010/04/01/ctgrind.html. Apr. 2010.

[66] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Dakar, Senegal, Apr. 2010, pages 348–370.

[67] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (July 2009), pages 107–115.

[68] Yehuda Lindell. “How to Simulate It — A Tutorial on the Simulation Proof Technique”. In: *Tutorials on the Foundations of Cryptography*. Cham, Switzerland: Springer International Publishing, 2017, pages 277–346.

[69] 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: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, Aug. 2018, pages 973–990.

[70] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy*. San Jose, CA, May 2015, pages 605–622.

[71] 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)*. Phoenix, AZ, June 2019, pages 1041–1053.

[72] lowRISC. *Ibex RISC-V Core*. https://github.com/lowRISC/ibex. 2015.

[73] lowRISC. *OpenTitan: Open source silicon root of trust*. https://opentitan.org.

[74] Nancy Lynch and Frits Vaandrager. “Forward and Backward Simulations – Part I: Untimed Systems”. In: *Information and Computation* 121.2 (Sept. 1995), pages 214–233.

[75] Rita Mayer-Sommer. “Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards”. In: *Proceedings of the 2000 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Worcester, MA, Aug. 2000, pages 78–92.

[76] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. “TPM-FAIL: TPM meets Timing and Lattice Attacks”. In: *Proceedings of the 29th USENIX Security Symposium*. Virtual conference, Aug. 2020, pages 2057–2073.

[77] Noah Moroze. “Kronos: Verifying leak-free reset for a system-on-chip with multiple clock domains”. Master’s thesis. Massachusetts Institute of Technology, Jan. 2021.

[78] 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)*. New York, NY, June 2022, pages 72–86.

[79] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, Mar. 2008, pages 337–340.

[80] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. “seL4: from General Purpose to a Proof of Information Flow Enforcement”. In: *Proceedings of the 34th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2013, pages 415–429.

[81] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. “Noninterference for Operating System Kernels”. In: *Proceedings of the 2nd International Conference on Certified Programs and Proofs*. Kyoto, Japan, Dec. 2012, pages 126–142.

[82] Andrew Myers and Barbara Liskov. “A Decentralized Model for Information Flow Control”. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France, Oct. 1997, pages 129–147.

[83] George C. Necula. “Translation validation for an optimizing compiler”. In: *Proceedings of the 21st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Vancouver, Canada, June 2000, pages 83–94.

[84] Luke Nelson, James Bornholt, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. “Noninterference specifications for secure systems”. In: *ACM SIGOPS Operating Systems Review* 54.1 (Aug. 2020), pages 31–39.

[85] Nitrokey. *Nitrokey HSM 2*. https://shop.nitrokey.com/shop/nkhs2-nitrokey-hsm-2-7.

[86] OASIS PKCS 11 Technical Committee. *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0*. https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/os/pkcs11-curr-v3.0-os.html. June 2020.

[87] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. “Revizor: Testing Black-Box CPUs against Speculation Contracts”. In: *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, Feb. 2022, pages 226–239.

[88] Oleksii Oleksenko, Marco Guarnieri, Boris Kopf, and Mark Silberstein. “Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing”. In: *Proceedings of the 44th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2023, pages 1737–1752.

[89] Marco Patrignani, Amal Ahmed, and Dave Clarke. “Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work”. In: *ACM Computing Surveys* 51.6 (Nov. 2019).

[90] Marco Patrignani and Deepak Garg. “Secure Compilation and Hyperproperty Preservation”. In: *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF)*. Santa Barbara, CA, Sept. 2017, pages 392–404.

[91] Amir Pnueli, Michael Siegel, and Eli Singerman. “Translation Validation”. In: *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lisbon, Portugal, Mar. 1998, pages 151–166.

[92] Thomas Pornin. *BearSSL Constant-Time Mul*. https://bearssl.org/ctmul.html.

[93] 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 Wintersteiger, and Santiago Zanella-Beguelin. "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider". In: *Proceedings of the 41st IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2020, pages 983–1002.

[94] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified Low-Level Programming Embedded in F*". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Oxford, United Kingdom, Sept. 2017.

[95] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats". In: *Proceedings of the 28th USENIX Security Symposium*. Santa Clara, CA, Aug. 2019, pages 1465–1482.

[96] Andrei Sabelfeld and Andrew Myers. "Language-Based Information-Flow Security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pages 5–19.

[97] Thomas Sewell, Magnus Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, June 2013, pages 471–482.

[98] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. "Nickel: A Framework for Design and Verification of Information Flow Control Systems". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, Oct. 2018, pages 287–306.

[99] Robert Sison, Scott Buckley, Toby Murray, Gerwin Klein, and Gernot Heiser. "Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems". In: *Proceedings of the 25th International Symposium on Formal Methods (FM)*. Lübeck, Germany, Mar. 2023, pages 103–121.

[100] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, and Alexei Czeskis. *Universal 2nd Factor (U2F) Overview*. https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf. Implementation draft. Sept. 2016.

[101] Richard M. Stallman. *Using the GNU Compiler Collection*. https://gcc.gnu.org/onlinedocs/gcc/.

[102] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. "Dependent Types and Multi-Monadic Effects in F*". In: *Proceedings of the 43rd ACM*

Symposium on Principles of Programming Languages (POPL). St. Petersburg, FL, Jan. 2016, pages 256–270.

[103] The Coq Development Team. *The Coq Proof Assistant, version 8.17.1*. June 2023.

[104] Emina Torlak and Rastislav Bodik. "A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, June 2014, pages 530–541.

[105] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. "CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests". In: *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture*. Fukuoka, Japan, Oct. 2018, pages 947–960.

[106] Jean-Baptiste Tristan. "Formal verification of translation validators". PhD thesis. Paris Diderot University, France, 2009.

[107] Florian Uekermann. *Buggy OTP slot range check*. https://github.com/Nitrokey/nitrokey-pro-firmware/issues/4. June 2016.

[108] ultraembedded. *biRISC-V - 32-bit dual issue RISC-V CPU*. https://github.com/ultraembedded/biriscv. 2020.

[109] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2/3 (1996), pages 167–188.

[110] 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 30th ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark, Nov. 2023, pages 2128–2142.

[111] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. "CT-wasm: type-driven secure cryptography for the web ecosystem". In: *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*. Cascais, Portugal, Jan. 2019, 77:1–77:29.

[112] *WhatsApp Security Whitepaper: Security of End-to-End Encrypted Backups*. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf. Sept. 2021.

[113] Claire Xenia Wolf. *Yosys Open SYnthesis Suite*. https://github.com/YosysHQ/yosys. 2012.

[114] Claire Xenia Wolf. *PicoRV32 – A Size-Optimized RISC-V CPU*. https://github.com/YosysHQ/picorv32. 2015.

[115] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *Journal of Cryptographic Engineering* 7 (2017), pages 99–112.

[116] *YSA-2015-1*. https://developers.yubico.com/ykneo-openpgp/SecurityAdvisory%202015-04-14.html. Apr. 2015.

[117] *YSA-2018-01*. https://www.yubico.com/support/security-advisories/ysa-2018-01/. Jan. 2018.

[118] *YSA-2020-04*. https://www.yubico.com/support/security-advisories/ysa-2020-04/. July 2020.

[119] Yubico. *YubiHSM 2*. https://www.yubico.com/product/yubihsm-2/.

[120] Katherine Zhao. "Verifying Hardware Security Modules With True Random Number Generators". Master's thesis. Massachusetts Institute of Technology, May 2024.

[121] Yongbin Zhou and Dengguo Feng. *Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing*. Cryptology ePrint Archive, Paper 2005/388. Oct. 2005.

[122] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, Oct. 2017.