Modular Verification of Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation

Anish Athalye[1] Henry Corrigan-Gibbs[1] M. Frans Kaashoek[1] Joseph Tassarotti[2] Nickolai Zeldovich[1]
[1] *MIT CSAIL* [2] *New York University*

Abstract

Parfait is a framework for proving that an implementation of a hardware security module (HSM) leaks nothing more than what is mandated by an application specification. Parfait proofs cover the software and the hardware of an HSM, which catches bugs above the cycle-level digital circuit abstraction, including timing side channels. Parfait's contribution is a scalable approach to proving security and non-leakage by using intermediate levels of abstraction and relating them with transitive information-preserving refinement. This enables Parfait to use different techniques to verify the implementation at different levels of abstraction, reuse existing verified components such as CompCert, and automate parts of the proof, while still providing end-to-end guarantees. We use Parfait to verify four HSMs, including an ECDSA certificate-signing HSM and a password-hashing HSM, on top of the OpenTitan Ibex and PicoRV32 processors. Parfait provides strong guarantees for these HSMs: for instance, it proves that the ECDSA-on-Ibex HSM 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.

1 Introduction

This paper presents an approach for proving the absence of correctness bugs, security bugs, and leakage bugs such as timing side channels, with modular reasoning. The paper applies this approach to verifying hardware security modules (HSMs), which are single-function devices intended to perform security-critical operations such as ECDSA public-key signatures.

Any vulnerability in an HSM's hardware or software can undermine the security of the HSM. These devices have suffered from bugs throughout the hardware/software stack, such as logic bugs, memory corruption, hardware bugs, leakage bugs, and timing side channels [1–8, 18, 24, 27, 41, 49, 69]. These bugs motivate the need for verifying security and non-leakage from application specification to circuit-level implementation, covering the full stack.

A common approach to reasoning about leakage across levels of abstraction is to use leakage models [11, 38], which augment a higher level of abstraction with additional information about what is leaked by a lower level. For example, the HACL* cryptography library [59, 77] assumes that hardware leaks branches and memory addresses, but not, for example, operands of addition and multiplication instructions. Getting leakage models right and aligning them across levels of abstraction such as C code, assembly, and circuit, is challenging [17, 71]. For instance, on the ARM Cortex-M3, the latency of multiply instructions is operand-dependent [12, 57], so HACL*'s ostensibly constant-time code could leak its secrets via timing side channels on this processor. There have been no verified hardware and software systems with top-to-bottom proofs of non-leakage using leakage models.

Knox [14] does provide top-to-bottom proofs of non-leakage, but without using any intermediate levels of abstraction or leakage models. It directly relates a circuit (including its firmware) to an application-level specification using *information-preserving refinement* (IPR), which rules out correctness bugs, security bugs, and timing bugs. The downside of Knox's monolithic approach is that it does not scale: for example, Knox cannot handle software that performs public-key cryptography. This is because the gap between the specification (e.g., abstract mathematical definitions for public-key crypto) and the circuit (e.g., code with sophisticated mathematical optimizations running on a pipelined processor [16, 21, 25, 29]) is too large for an SMT solver to establish correspondence.

This paper presents *Parfait*, a framework for verifying security and non-leakage with modular reasoning. The key contribution of Parfait is formalizing Knox's IPR in a way that allows IPR to be applied transitively. This allows for bridging the gap between an application-level specification and a circuit-level implementation in a modular way with intermediate refinements. To verify our case-study HSMs, we use five levels of abstraction: specification, proof-oriented programming language, C, assembly, and circuit. Given proofs between pairs of successive levels of abstraction, the transitivity of IPR implies end-to-end security and non-leakage of the entire hardware and software system.

Transitive IPR has three additional benefits. First, Parfait can simplify the proof by verifying IPR between levels of abstraction for a specific implementation rather than proving a

This work is licensed under a Creative Commons Attribution International 4.0 License.
Copyright is held by the owner/author(s).
SOSP '24, November 4–6, 2024, Austin, TX
ACM ISBN 979-8-4007-1251-7/24/11.
https://doi.org/10.1145/3694715.3695956

general-case theorem for arbitrary code, such as proving that a processor correctly implements an ISA or that assembly-level and hardware-level leakage models match. Second, Parfait can use different proof techniques for different levels of abstraction. Parfait formalizes in Coq [66] the *IPR by functional-physical simulation* proof technique from Knox and introduces two additional proof techniques for verifying software: *IPR by equivalence* and *IPR by lockstep*. Third, Parfait can reuse existing tools and build on existing proofs for verifying each level of abstraction. Parfait uses F⋆ [65] and encodes IPR into pre/postcondition-style specifications for verifying software, which allows HSM developers to reuse crypto code and proofs from HACL⋆ [77]. For example, our ECDSA HSM builds on the ECDSA code from HACL⋆, including its proof. Parfait uses the verified CompCert compiler [42] for compiling C to assembly. Finally, Parfait uses Rosette [67] and extends Knox [14] with an *assembly-circuit synchronization* technique to make verification tractable for complex HSMs.

To demonstrate the Parfait approach, we implemented a prototype of Parfait, which comprises the *Starling* framework for verifying software and the *Knox2* framework for verifying hardware. We used these frameworks to develop and verify four example HSMs, including a PKCS#11-compatible ECDSA certificate-signing HSM and an HMAC-based password-hashing HSM. Parfait proofs ensure that the I/O behavior of these HSMs, down to the level of digital gates and wires, leaks no information beyond what is allowed by their application-level specifications. This rules out a large class of correctness bugs (logic bugs, memory corruption, etc.) as well as leakage bugs (such as error messages that reveal sensitive information or non-constant-time crypto). Parfait's modularity makes it easy for app developers to build new HSMs and for hardware developers to port HSMs to new hardware platforms. For instance, after verifying our applications on the Ibex-based SoC, we were able to verify the same application on the PicoRV32 processor in only two additional developer hours.

In summary, the contributions of this paper are (1) the Parfait approach for modular verification of security and non-leakage, including a formalization of transitive IPR and proof strategies for IPR, (2) the Starling framework for verifying IPR for software, (3) the Knox2 framework for verifying IPR for hardware, and (4) an evaluation of Parfait verifying four HSMs. A more detailed description of Parfait can be found in the first author's PhD thesis [13]. All source code—IPR theory, verification frameworks, and verified HSMs—is available at: `github.com/anishathalye/{ipr,starling,knox,parfait-hsm}`.

One of the limitations of Parfait is that it focuses on simpler CPU designs used in a large class of HSMs today, such as the OpenTitan's Ibex. New ideas and techniques would be needed to extend Parfait to handle high-performance out-of-order processors such as Intel and AMD x86 CPUs that are used in other HSMs.

2 Overview and HSM developer workflow

HSM design framework. Parfait provides an HSM design framework that helps avoid timing leakage. HSMs following this design, shown in figure 1, run an execution loop that (1) reads a command from the I/O interface, (2) loads state from persistent memory, (3) handles the command to produce an in-memory state update and response, (4) updates persistent state atomically, and (5) sends the response over the I/O interface.

```
uint8_t state[STATE_SIZE];
uint8_t cmd[COMMAND_SIZE];
uint8_t resp[RESPONSE_SIZE];

void main() {
  while (true) {
    read_command(&cmd); // from I/O interface             (1)
    load_state(&state); // from persistent memory         (2)
    handle(&state, &cmd, &resp); // core computation      (3)
    store_state(&state); // to persistent memory, atomic (4)
    write_response(&resp); // to I/O interface            (5)
  }
}
```

Figure 1: The main loop of an HSM in the Parfait design.

Steps (1) and (5) do not compute over the HSM's internal state, and steps (2) and (4) read/write secret state opaquely without computing over secret values, so the developer can write these functions to run in constant time. Step (4) requires some care to implement atomicity.

At the core of the HSM is a `handle` function that implements step (3), implementing command deserialization, core functionality, and response serialization. The developer must write this function such that its execution time (i.e., the number of hardware cycles) depends only on the `cmd` (the serialization of a command and its arguments), not on the internal `state` of the HSM.

Writing code that executes in constant time at the hardware level requires careful programming at the source-code level, a compiler that preserves constant-time behavior, and hardware that executes the code in constant time. Parfait helps developers prove that their HSM is indeed leakage-free.

Developer workflow. Figure 2 gives an overview of the Parfait developer workflow (§2) and verification approach (§3). Parfait supports two largely independent developers: an app developer who writes and verifies the application (§4) and a platform developer who writes and verifies the system software (for persistence, peripheral I/O, etc.) and hardware for running the application (§5).

App development. The app developer writes an implementation of the application functionality in Low⋆ [58] (the App Impl [Low⋆] in figure 2), 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 1, which oper-

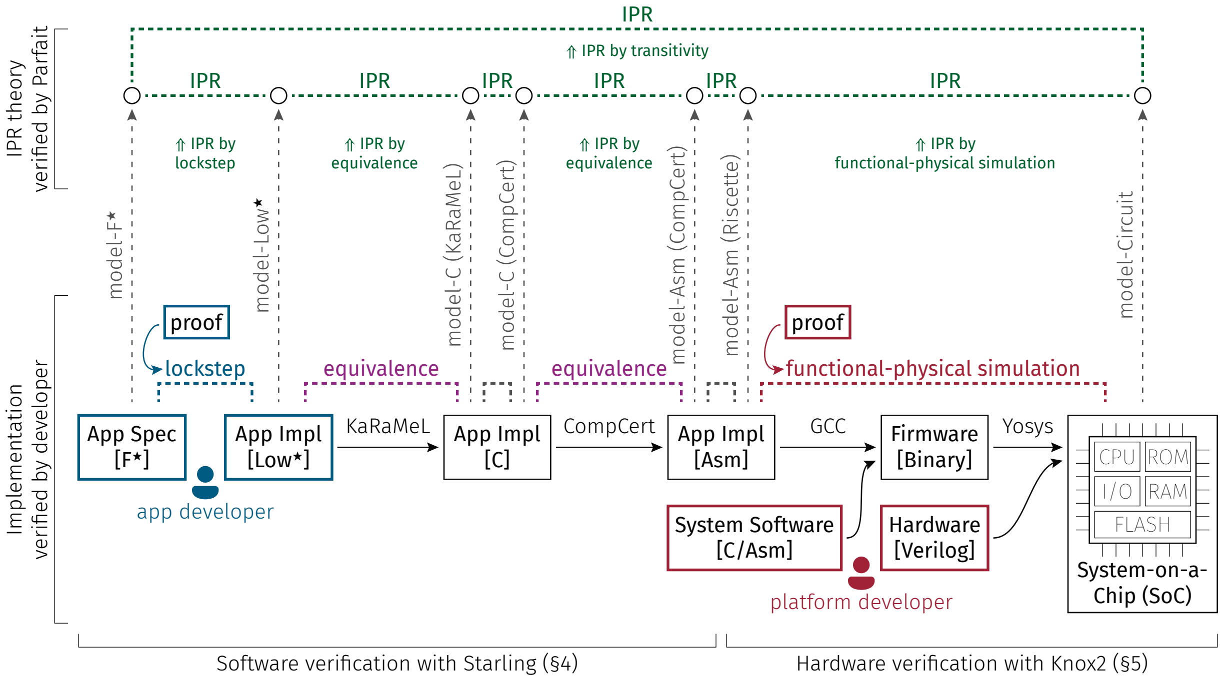


Figure 2: The Parfait developer workflow and verification approach. The app developer writes the App Spec [F*], App Impl [Low*], and a proof of lockstep verified using the Starling framework in F* (§4). Off-the-shelf verified compilers provide the equivalence proofs. The platform developer writes the System Software, Hardware, and a proof of functional-physical simulation verified using the Knox2 framework in Rosette (§5). The Parfait framework provides a theory of IPR that is verified once-and-for-all in Coq (§3). An on-paper argument connects the mechanized proofs written by the HSM developer to the mechanized theory of IPR provided by Parfait by modeling each level of abstraction as a state machine in the formalism of IPR to prove a top-level theorem of IPR between the App Spec [F*] and the System-on-a-Chip (SoC).

ates on in-memory state, command, and response buffers. Off-the-shelf verified compilers turn this code into an assembly-level implementation (the App Impl [Asm] in figure 2).

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 1, and the implementations of `read_command`, `load_state`, `store_state`, and `write_response`. 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 2). 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.

Trusted computing base. Among the code the HSM developer writes, only the App Spec and *driver* (model of how a well-behaved client communicates in IPR, not shown in figure 2), both described in §3, are in the trusted computing base (TCB). The entire implementation—the Low* app code, system software, and hardware—is covered by verification. §6 describes the TCB of the Parfait framework itself.

Threat model. Parfait considers 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 attacks, such as an adversary that compromises the HSM's 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 focuses 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 [76] such as EM radiation [9], temperature [40], and power [48].

3 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 2.

IPR. In Parfait's formalization of IPR, each level of abstraction is a state machine satisfying the interface shown in figure 3. A level of abstraction is defined by the types of commands and responses, the type of state, the initial state, and a step function that describes the behavior of each command. In Parfait, every level of abstraction is *modeled* as a state machine of this form.

```
Record state_machine (command response : Type) := {
  state : Type;
  init : state;
  step : state -> command -> (state * response);
}.
```

Figure 3: The interface of state machines in IPR, written in Coq.

A Parfait developer writes application specifications (the App Spec in figure 2) in explicit state-machine style. For example, figure 4 shows the step function from the specification of an ECDSA-signing HSM. A single HSM operation (such as `Sign msg`) is an atomic step of this state machine, which we call "whole-command."

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

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

Figure 4: The step function for the ECDSA HSM, written in F*. The specifications for hmac and `ecdsa_signature_agile` are used directly from HACL*, a verified crypto library. This HSM specification does not support reading out the signing key or pseudorandom function (PRF) key, and it ensures unique nonces across operations.

In contrast, when modeling the SoC as an IPR state machine, the state is the entire state of the circuit, and there are three commands: (1) `set_input(...)`, which sets signals on input wires to the HSM, (2) `get_output()`, which reads signals on output wires from the HSM, and `tick()`, which waits for the HSM to run for a single clock cycle, whose step function is defined by the SoC's circuit. One logical HSM operation corresponds to millions of state-machine steps at this level.

Figure 5 illustrates IPR, which is defined as an equivalence between a real world and an ideal world. The real world contains the implementation state machine. In the real world, a *driver* describes how to obtain spec-level behavior from the implementation, akin to a device driver, a program mapping spec-level operations to implementation-level I/O. The developer writes the driver, which is part of the TCB. The real world exposes both a spec-level (e.g., figure 4) and an implementation-level interface (e.g., wire-level inputs/outputs of the SoC). A client of the state machine could use the spec-level interface (through the driver), or bypass the driver and perform arbitrary implementation-level I/O operations (representing what an adversary could do).

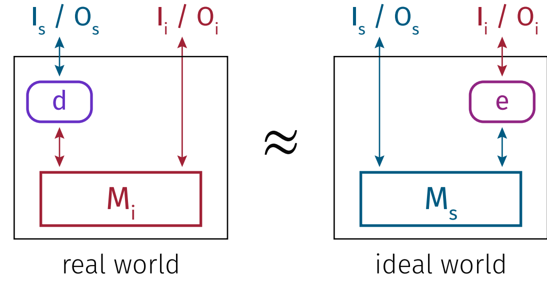


Figure 5: The definition of IPR: an implementation M_i (with command/response types I_i / O_i) is an information-preserving refinement of a specification M_s (with command/response types I_s / O_s) with respect to a driver d, written as $M_i \approx_{IPR[d]} M_s$, if there exists an emulator e such that the real world is observationally equivalent to the ideal world.

The ideal world contains the specification, and the IPR definition states that there must exist an *emulator* (a dual of the driver) in the ideal world. The emulator is a proof artifact that is not part of the TCB. The emulator, which exposes an implementation-level interface, must mimic the real implementation's behavior while given only query access to the specification.

The IPR definition states that the real world must be observationally equivalent [13: §4.1.2] to the ideal world. If the emulator, given access to only the ideal-world spec, can always produce an output that is equivalent to the real-world implementation, then it must be that the real-world implementation and driver cannot introduce any unintended leakage beyond the specification.

Top-level theorem. Parfait's top-level theorem states that the bottom (SoC) level of abstraction securely implements the top (app spec) level of abstraction, meaning that the SoC returns the correct results and does not leak any additional information. Parfait specifies this using IPR:

$$\text{model-Circuit (SoC)} \quad \approx_{IPR[d]} \quad \text{model-F}^\star \text{ (App Spec)}$$

Proof approach: transitive IPR. To prove the top-level theorem—that the SoC is an information-preserving refinement of the app spec—Parfait breaks up the IPR relation into smaller steps. To do this, Parfait proves that IPR, as encoded in Parfait's formalization, is transitive. That is, if we have three levels of abstraction M_1, M_2, and M_3, with IPR between each pair, we can establish that IPR holds between M_1 and M_3:

$$\frac{M_1 \approx_{IPR[d_{12}]} M_2 \qquad M_2 \approx_{IPR[d_{23}]} M_3}{M_1 \approx_{IPR[d_{12} \circ d_{23}]} M_3}$$

Parfait formalizes the definitions of IPR, transitivity, and composition in the Coq proof assistant, described in detail in the first author's PhD thesis [13: §4].

Levels of abstraction. Table 1 shows the five *levels of abstraction* used to verify our case-study HSMs. We chose these levels so that we could reuse existing tools and libraries.

Table 1: The levels of abstraction used to verify our case-study HSMs. This table shows the state, input/output types, and step 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	State	I / O	Step
App Spec [F*]	`state_t`	`command_t` / `response_t`	`step()`
App Impl [Low*]	bytes	bytes	`handle()`
App Impl [C]	bytes	bytes	`handle()`
App Impl [Asm]	bytes	bytes	`handle()`
System-on-a-Chip	registers & memories	wires	cycle step

Parfait's overall approach involves proving IPR between each level of abstraction. At the top level is the app-developer-supplied application spec, such as figure 4. The second level is the app implementation, which implements the core app 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 machine words, 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 logical operation is a single step. The last level introduces cycle-precise timing; verification at this level catches timing bugs.

Between each of these levels of abstraction, a driver describes how inputs/outputs at the higher level of abstraction (e.g., `command_t`/`response_t`) 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 proof techniques. Figure 2 shows how Parfait uses IPR proof techniques. In addition to the transitivity of IPR, Parfait formalizes three proof techniques for IPR:

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 (developer-defined 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 (one 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 supplies 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 observationally equivalent. This applies when using verified compilers, where the state machines given by the corresponding models are equivalent: observational *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 [46] to the IPR setting introduced by Knox [14], which applies when a *functional-physical simulation relation* 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; Parfait provides a formulaic method for constructing emulators that follow the Parfait HSM design framework.

Software and hardware proofs. The developer uses the *Starling* framework to prove *lockstep* between the specification and the app implementation. Starling encodes the lockstep property as a precondition/postcondition for the Low* `handle` function, which allows the developer to reuse existing verified software such as HACL*. From the Low* implementation, Parfait uses verified compilers (KaRaMeL [58] and CompCert [42]) to produce an assembly implementation that is observationally equivalent to the Low* code, so *equivalence* holds between the models of the levels as state machines. §4 describes this software verification approach 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 *Knox2* framework to prove *functional-physical simulation*. §5 describes this hardware verification approach in detail.

Connecting mechanized proofs. Parfait combines these mechanized proofs together with an on-paper argument to provide an 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 models each level of abstraction as a state machine in the formalism of IPR and uses the verified proof strategies, including transitivity, to prove the top-level IPR between App Spec and SoC.

Parfait gives each level (e.g., Asm code) an interpretation as a state machine in the formalism of IPR. This is the connection to the theory mechanized in Coq. For example, figure 8 shows how model-Asm (CompCert) interprets the Asm code as a state machine. This modeling is on-paper, as is the connection between (1) Starling's encoding of lockstep in F⋆ and the Coq definition of lockstep, and (2) Knox2'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 models of the Low⋆ and C as whole-command state machines are equivalent state machines, an on-paper 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 on-paper argument (§4.2).

For compiling C to Asm, Parfait uses CompCert; its (proven-in-Coq) correctness implies that the models of the C and Asm as state machines are equivalent, another on-paper 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 (§5.1).

4 Software verification with Starling

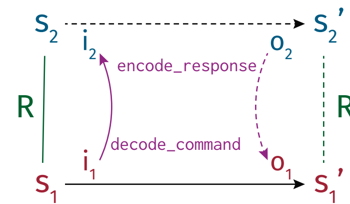
Parfait's *Starling* framework supports the application developer in implementing the HSM software and proving IPR between the specification and the assembly-level code. A key challenge in the software verification component of Parfait is minimizing proof effort and enabling reuse of existing specifications,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

4.1 Low⋆-level proof

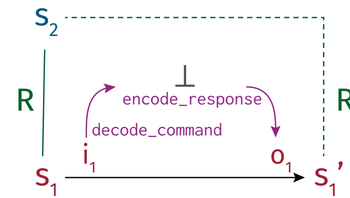
There is a simple correspondence between the F⋆-level spec and Low⋆-level implementation, where one step of the implementation state machine corresponds to a single step of the spec state machine, and vice versa. For this reason, the driver, which translates spec-level operations to implementation-level operations, is simple. It needs only to describe how commands are encoded (an `encode_command` function) and how responses are decoded (a `decode_response` function). The driver has the form (1) serialize the spec-level input (e.g., `Sign msg`) as a buffer, (2) invoke `handle`, and (3) decode the response buffer into a spec-level output (e.g., `Signature sig`). As a result, Starling can use IPR by lockstep to verify the implementation at this level.

IPR by lockstep. The lockstep proof strategy implicitly constructs an emulator based on developer-supplied encode/decode functions that are duals of those comprising the driver: a `decode_command` function that translates bytes to spec-level inputs, and an `encode_response` function that translates spec-level outputs to bytes. The `decode_command` function produces an `option`-typed output to allow for low-level inputs that do not correspond to any high-level input. The `encode_response` function consumes an `option`-typed input to support producing low-level outputs (signaling an error response) for the situation where there is no valid high-level input and hence no valid high-level output.

The lockstep proof strategy requires the developer to supply: (1) the encode/decode functions (which implicitly define an emulator); (2) a proof of correspondence between encoders and decoders (that decode is the inverse of encode); (3) a refinement relation R between spec-level state (`state_t`) and Low⋆-level state (bytes); and (4) a proof of the *lockstep simulation* property shown in figure 6. Together, these imply IPR.



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



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

Figure 6: The two cases of lockstep simulation: the low-level input corresponds to some high-level input (a) or none (b).

Encoding in F⋆. The Starling framework encodes the lockstep property—in particular, the encode/decode correspondences and the lockstep simulation property—into F⋆, and the app developer proves the properties.

Starling provides the developer with signatures for the encode/decode functions for commands and responses; the postconditions of the encode functions ensure that the functions satisfy the required correspondence.

Starling encodes lockstep simulation into the signature of

the Low* handle function, handle_st, as shown in figure 7. Rather than use a refinement relation between states, Starling uses an encode_state 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 handle 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 (figure 6), 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 6a) and the case of invalid low-level input (cmd_spec = None, corresponding to figure 6b).

```
let handle_st (spec: spec_t) ... =
    state:buffer uint8{length state = state_size}
 -> state_spec:erased spec.state_t
 -> command:buffer uint8{length command = command_size}
 -> response:buffer uint8{length response = response_size} ->
 Stack unit
 (requires fun h -> ... /\
  as_seq h state == encode_state state_spec)
 (ensures fun h0 () h1 -> ... /\
  let cmd_spec = decode_command (as_seq h0 command) in
  match cmd_spec with
  | Some v ->
    let (state_spec', resp_spec) = spec.step state_spec v in
    as_seq h1 state == encode_state state_spec' /\
    as_seq h1 response == encode_response (Some resp_spec)
  | None ->
    as_seq h1 state == as_seq h0 state /\
    as_seq h1 response == encode_response None)
```

Figure 7: Starling's encoding of lockstep simulation in F* as the signature of handle.

The combination of encode/decode correspondences and the postcondition of handle guarantee non-leakage. The intuitive reason is:

- When the command can be decoded as a spec-level command (i.e., Some v), then the behavior matches the spec: the final state matches the encoding of the final spec state, and the value in the response buffer matches the encoding of the spec-level response. 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.

4.2 C and assembly-level proofs

Parfait compiles Low* to assembly code using a stack of verified compilers. KaRaMeL [58] 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 [42] 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 8 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. The step function takes as inputs a state buffer and command buffer and returns a new state buffer and response buffer as outputs. The model-Low*, model-C (KaRaMeL), and model-C (CompCert) interpretations are analogous.

```
type state_t = bytes[STATE_SIZE]
type command_t = bytes[COMMAND_SIZE]
type response_t = bytes[RESPONSE_SIZE]

def step(state, command) -> (state_t, response_t):
  m = compcert_asm_abstract_machine("AppImpl.asm.json")

  # copy state and command into machine memory
  state_ptr = m.alloc(STATE_SIZE)
  m.storebytes(state_ptr, state)
  command_ptr = m.alloc(COMMAND_SIZE)
  m.storebytes(command_ptr, command)

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

  # 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)
  response = m.loadbytes(response_ptr)
  return (new_state, response)
```

Figure 8: Pseudocode describing how model-Asm (CompCert) interprets the CompCert Asm as a state machine according to the CompCert RISC-V Asm 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 models

each level as a state machine, observes that the state machines are observationally equivalent due to using verified compilers, and applies *IPR by equivalence* to obtain IPR between these levels.

This approach has two limitations. Although KaRaMeL semantics are intended to coincide with CompCert C [58], there is no mechanized connection between the two. Parfait assumes that the state machines from model-C (KaRaMeL), interpreting the C code according to the KaRaMeL C semantics, and model-C (CompCert), interpreting the C code according to the CompCert C semantics, are observationally equivalent. Parfait does not need to assume that the semantics perfectly coincide (e.g., stepwise correspondence between the small-step operational semantics of KaRaMeL C and CompCert C), only that the state machines coincide, which boils down to assuming that the final values computed 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.

5 Hardware verification with Knox2

Parfait's *Knox2* framework supports the platform developer in implementing the HSM hardware and proving IPR between the assembly-level code and its circuit-level implementation. Knox2 builds on top of the Knox framework [14] and makes two contributions to make Knox compatible with transitive IPR and to scale up to more sophisticated HSMs: *Riscette*, a RISC-V assembly semantics in Rosette (§5.1), and the *assembly-circuit synchronization* technique (§5.4).

5.1 Assembly semantics in Rosette

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. Knox is written using Rosette, a symbolic evaluation library for the Racket programming language, so it cannot directly use the CompCert semantics. For this reason, on top of Rosette, Knox2 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, Knox2 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 8 describes the assembly level's interpretation as a state machine following CompCert semantics.

The assembly level is a whole-command state machine, so the execution of a single logical HSM operation occurs in a single step, but at the circuit level, it takes many state-machine steps (tens of millions, for the ECDSA HSM) to execute a single logical operation. Proving IPR between these levels of abstraction ensures that the circuit's wire-level behavior does not leak any information.

5.2 Driver

To relate the assembly and circuit levels of abstraction in IPR, the platform developer defines an driver that translates from the inputs/outputs at the assembly level (buffers) to interaction with the state machine at the hardware level (signals on wires). The driver describes the I/O protocol that the client implements: how to send a buffer over the wire (which the HSM reads in `read_command`), and how to read the response (which the HSM sends in `write_response`). This is like a device driver, in the form of a program built on the circuit-level primitives: `set_input`, `get_output`, and `tick` [13: §6.3].

5.3 Proof strategy

To prove IPR between the app assembly (which serves as the specification for hardware verification) and the circuit level of abstraction, Knox2 reuses the same proof strategy implemented in Knox: IPR by functional-physical simulation. This strategy requires the developer to supply: (1) a refinement relation between assembly-level state and circuit-level state; (2) an emulator; and (3) a proof of *functional-physical simulation*. Together, these imply IPR. These components are largely independent of the HSM app logic.

The refinement relation relates the state at the app assembly level, which is a buffer, with the state at the circuit level, which includes the registers and memories of the circuit, including persistent memory. These states are closely related, but the mapping is not one-to-one, because the system software implements atomicity and crash safety (in `load_state` and `store_state`). Our case-study HSMs use a simple journaling strategy, a single flag word (which is atomically writable) to toggle between two copies of state stored in persistent memory, so the refinement relation relates the assembly-level state to the active region of memory (based on this flag), as shown in figure 9.

```
Inv(impl) /\
spec = if impl.storage[0] == 0
       then impl.storage[1 : STATE_SIZE+1]
       else impl.storage[STATE_SIZE+1 : 2*STATE_SIZE+1]
```

Figure 9: An example of a refinement relation between assembly-level implementation (the `spec` for the circuit level) and circuit `impl`. `impl.storage` 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.

Parfait provides a template for constructing an emulator 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 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 framework, the commit point is the (cycle-level) commit point of the `store_state` function in the system software (figure 1). To use this template, the platform developer supplies (1a) a function that identifies when the circuit is about to begin execution of `handle`, (1b) a function that computes the spec-level command (bytes) from the circuit state, (2a) a function that identifies when the circuit is at the commit point of the `store_state` function, and (2b) a function that injects the spec-level response (bytes) into the circuit state.

The Knox2 proof for IPR between the app assembly and SoC depends on the system software and hardware, but not on the app software itself, aside from being parameterized by a few values, like the length of the state encoding, `STATE_SIZE`. The reason Knox2 is able to bridge this large gap, to hardware, is that (1) the app assembly, which is the specification for this IPR proof, is already in terms of "shuffling bits around," which the hardware does the same way as the assembly; and (2) Knox2 uses automation powered by symbolic execution and SMT solvers to automate proving that the app assembly and hardware execution correspond, as we explain next.

5.4 Assembly-circuit synchronization

A key challenge for Knox2'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 cycles to execute in the SoC, corresponding to a single step of the assembly-level state machine (§5.1). 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 Knox2 can symbolically execute both the assembly and the circuit, 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, Knox2 uses a strategy of incrementally executing the assembly and periodically synchronizing the assembly and circuit. Although the app assembly level is modeled as a whole-command state machine that executes a command in a single step, Riscette computes that step by symbolically executing instruction-by-instruction. Knox2 makes use of this per-instruction stepping to simplify equivalence checking. When the hardware is in the middle of executing the `handle` function, there is a close correspondence between the hardware's cycle-by-cycle execution and single-stepping through CompCert Asm-level instructions.

To do this synchronization, Knox2 uses a mapping between CompCert Asm abstract machine state and hardware-level state (registers and memories) provided by the platform developer. During the proof, Knox2 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. Knox2 has built-in heuristics for when to synchronize, and for what should be synchronized in which situations.

Knox2 uses a best-effort strategy for synchronization. Occasionally the developer-provided mapping or heuristics for alignment are incorrect and an equivalence check fails. When this happens, Knox2 does not unify the symbolic expressions. Instead, it continues symbolically executing and tries to check for equivalence later. The result is that the solver will 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 (§7) as an example.

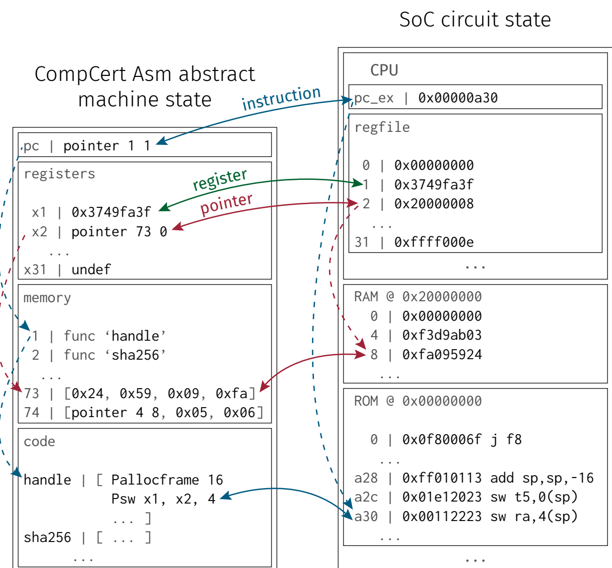


Figure 10: Correspondence between Asm machine state and circuit state. The figure illustrates an example instruction mapping, register mapping, and pointer mapping.

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 10, 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 any concrete pointer value (e.g., `0x20000008`), what is the Verilog memory and index to which it corresponds? For example, in the Ibex SoC, the mapping describes that pointers correspond to the Verilog memory `ram` and that the index (e.g., 2 machine words) is obtained by subtracting the base address of the RAM (`0x20000000`) from the pointer value.
- **Encoding of next RISC-V instruction**, including whether or not it is valid (it may be invalid if the execute stage of the processor pipeline is stalled). This is what Knox2 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.

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. Knox2 synchronizes assembly 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; Knox2 invokes an SMT solver to prove that the assembly register and circuit register have the same value, and replaces both with the same symbolic variable.
- **Pointers**: when an assembly register has a pointer value, Knox2 guesses that the value in the hardware register is also a pointer, and points to flat memory. In this case, Knox2 synchronizes the *contents* of the memories, and leaves the pointers in the registers untouched (a CompCert `pointer` in the assembly, and a 32-bit bitvector in the circuit). Synchronizing the memory contents is similar to synchronizing registers. Knox2 knows the bounds of the allocation thanks to CompCert's structured memory model. Knox2 uses the SMT solver to prove the chunk of memory equal between assembly and circuit, doing this byte-by-byte, and synchronizing values that are equal.
- **Undef**: when an assembly register is undefined, Knox2 leaves the circuit register as-is.

All of the examples in figure 10 are shown with concrete values, but when Knox2 is used for verification, many 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, Knox2 uses a number of heuristics to decide when to synchronize, as shown in figure 11. Knox2 watches the instruction being executed in the assembly-level machine, and the instruction being executed in the circuit-level machine (thanks to the developer-written mapping that provides the next-executing instruction). Knox2 steps each machine up until the next synchronization point, and then does the synchronization. In some cases, there is a direct correspondence between Asm instructions and hardware instructions, in other cases, it is a more complex mapping. Figure 11 shows CompCert Asm instructions and their corresponding RISC-V assembly instructions or instruction sequences. Knox2 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.

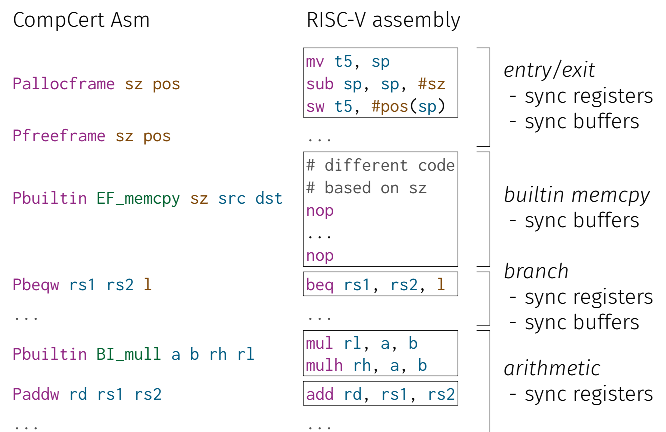


Figure 11: Knox2 synchronization points and corresponding actions.

6 Implementation

The Parfait framework consists of IPR theory formalized in 3000 lines of Coq [66], the Starling framework written in 100 lines of F* [65], and the Knox2 framework written in 3000 lines of code on top of Knox [14], Rosette [67] and Racket [32]. The Knox2 framework includes Riscette, a single-steppable symbolically executable semantics of CompCert RISC-V Asm in Rosette.

The Parfait approach relies on a number of other tools. Parfait uses the KaRaMeL compiler [58] to extract Low* to C and the CompCert compiler [42] to compile C code to RISC-V Asm. Parfait forks the CompCert compiler and adds 450 lines of code to annotate compiler-expanded memcpy builtins with `nop` instructions to aid in synchronization and to dump the RISC-V Asm AST to a `.json` file before emitting the final `.s` file. Compiling the system software and linking into a

firmware binary uses GCC [64]. Synthesizing the Verilog code and extracting a step model for the circuit uses Yosys [73]. Both F* and Rosette use the Z3 SMT solver [28]. The case studies use the HACL* [77] verified crypto library.

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 Knox2 framework's Riscette semantics (needs to match CompCert), (3b) the Knox2 framework's conversion of a circuit (Verilog/software) into Rosette, and (3c) the Knox2 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 Z3). 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.

7 Verifying HSMs using Parfait

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

7.1 Case studies

We implemented 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* [77] library. We run these applications on two hardware platforms: one based on the Ibex processor [45] from the OpenTitan project [44] and one based on the PicoRV32 processor [74].

Application 1: ECDSA certificate signing. An ECDSA certificate-signing app, described in figure 4, is the running example through this paper. The complete F* specification 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 (PRF) 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 Knox2 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. Our 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, which defends against offline brute-force attacks on stolen password databases [22]. Figure 12 shows the core of the specification. The complete F* specification 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 12: The step 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 based on the Ibex processor from the OpenTitan hardware root of trust. The Ibex is a two-stage pipelined RISC-V processor written in 13,000 lines of SystemVerilog. This open-source CPU is not purpose-built for verification. Our platform includes a peripheral for a wire-level I/O interface for the HSM: 4-wire UART with flow control. 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.

We wrote system software for this platform—main loop, I/O code, and persistence code—in 300 lines of C and assembly.

Our implementation makes two changes to the CPU: we remove async resets because Knox does not support them, and we replace the Ibex's multiplier with a simple full-width Verilog multiplication of operands, leaving it to the synthesis tool to infer an optimal implementation [13: §8.1].

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 §8.1.

7.2 Attack discussion

Parfait proves IPR between the application 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 examples of possible bugs and explain what part of the verification process prevents those bugs:

- *Software logic bug (e.g., integer overflow of the PRF counter)*: Starling will catch this when verifying the postcondition for the Low⋆ implementation, which ensures that the final state of the implementation matches final state of the specification.
- *Buffer overflow or use-after-free*: Low⋆ verification prevents these memory safety bugs. In particular, type checking in the `Stack` effect will catch these memory safety bugs.
- *Software-level leakage (e.g., returning different error codes for PRF counter overflow versus invalid curve point)*: Starling will catch this when verifying the postcondition for the Low⋆ implementation, which ensures that the output of the implementation corresponds to a deterministic function of the output of the specification.
- *Timing leakage from branching on a secret*: although this is a "software bug," Parfait does not introduce any notion of timing until the SoC level—higher levels of abstraction execute HSM operations in a single state-machine step. Knox2 will catch this because the emulator's behavior will not match the circuit's behavior: the emulator does not 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.
- *Compiler-introduced timing leakage*: if a compiler optimization introduces a timing bug, such as returning early from `memcmp`, Knox2 will catch this bug at the SoC level, just as in the above example.
- *Hardware-level timing leakage from a variable-latency arithmetic instruction executed on secret data*: Knox2 will catch this bug, just as in the above example.
- *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. Knox2 will catch stack overflow bugs when it relates the SoC with a bounded stack to the app assembly with an unbounded stack.
- *I/O code bug in system software (e.g., incorrectly encoding the output or setting the wrong UART baud rate)*: Knox2 will catch this bug when verifying functional-physical simulation.
- *Pipeline hazard in CPU implementation*: Knox2 will catch this, because if this occurs while executing app code, there will be a mismatch between the app assembly execution (which uses the Riscette instruction-by-instruction execution semantics) and the hardware execution.

8 Evaluation

This section answers two key questions: what is the developer effort to verify HSMs with Parfait (§8.1), and what is the performance of HSMs verified with Parfait (§8.2).

8.1 Developer effort

Table 2 summarizes the lines of code required to specify and implement each of the four HSMs from §7 (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 2: Lines of code for case studies. The specification covers the hardware and software stack. Spec LoC counts the lines the HSM developer writes (and does not 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 (LoC)	Driver (LoC)	Platform	Implementation Software (LoC)	Implementation Hardware (LoC)
ECDSA signer	40	100	Ibex	2,300	13,500
			PicoRV32	2,300	3,000
Password hasher	30	100	Ibex	1,000	13,500
			PicoRV32	1,000	3,000

Table 3 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 3: 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 4 shows the number of lines of proof required to verify the two platforms with Knox2. 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 4: 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 hrs	100 hrs	671	0.14 hrs	588

The "Verification" columns of table 4 show Knox2'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 the other side. After such a change, the only requirement is to run Knox2 on the new software/hardware combination.

Knox2 verification 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 Knox2 to simulate more SoC execution cycles.

Development cycle. If the Low* implementation has a timing bug when executed by the circuit, Knox2 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; Knox2 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 non-constant-time code, such as `if (secret) { ... }` or `x / secret`. Tracing the issue from the KaRaMeL-generated C code to the developer-written Low* source code is straightforward 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.

8.2 Performance

Parfait's use of the CompCert compiler introduces run-time overhead, because CompCert emits less performant code than GCC does. Table 5 measures this performance penalty, showing that two commercial HSMs have ECDSA-signing throughput that is within 12× 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.

Table 5: Run-time performance comparison of HSMs, in ECDSA signatures per second. The Ibex processor is clocked at 100 MHz, which is the OpenTitan reference clock.

HSM	**Compiler**	**Sig/s**	**Speedup**
Parfait ECDSA/Ibex	CompCert -O1	1.1	-
	GCC -O2	8.1	7×
Nitrokey HSM 2 [55]		12.5	11×
YubiHSM 2 [75]		13.7	12×

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 in Parfait.

9 Related work

Hardware/software verification. Because bugs anywhere in an HSM can compromise security, Parfait's proofs span both software and hardware. Several prior works have developed verified systems with proofs that cover software and hardware in an integrated way [10, 19, 30, 31, 47]. In contrast to Parfait, proofs for these systems only establish a form of functional correctness and do not rule out information leaks.

Athalye et al. [14] introduced IPR and the Knox framework for proving IPR. Knox carries out proofs in a monolithic way by symbolically executing hardware at the register-transfer level (RTL) and checking refinement from an app specification. Parfait uses transitivity of IPR to compose proofs for different levels of abstraction of an HSM. This modularity enables

scaling to HSMs that are more complex than the examples verified using Knox.

Verifying non-leakage. Several tools exist for checking constant-time behavior of software [11, 15, 20, 72]. These tools do not account for leakage at the hardware level, so their soundness depends on whether their assumed leakage model of the hardware is accurate. In contrast, Parfait proofs model the hardware at the RTL level in a cycle-accurate way, which allows verification to rule out a large class of hardware-related bugs as well.

Other works have sought to verify the correctness of leakage models. CompCert-CT [17] extends the verified CompCert C compiler [42] to show that leakage specified at the level of C programs is preserved by compilation to assembly. Because some transformations performed by CompCert can potentially introduce timing leaks in certain programs, CompCert-CT uses modified versions of these passes. Parfait instead uses standard CompCert, which ensures IPR for whole-command state machines. Afterward, in order to establish IPR at the circuit level, Knox2 checks that the program running on an HSM cannot have timing leaks, including any that could have been introduced by CompCert.

On the hardware side, LeaVe [71] verifies at the RTL level that an ISA-level leakage model is accurate for a processor. LeaVe has been used to verify leakage models for simple RISC-V processors, including variants of the Ibex processor used in Parfait case studies. To simplify verification, LeaVe carries out analysis under an assumption that a processor is functionally correct. In contrast, Parfait does not assume functional correctness of the HSM's processor.

For more complex hardware, formulating leakage models that can account for microarchitectural state and speculative execution is challenging and the focus of much recent work [23, 38, 50]. The Ibex and PicoRV32 processors used in the example HSMs verified with Knox do not have these kinds of features. The overall Knox2 approach is agnostic to such features, but more complex processors would make it more challenging to describe state correspondence and perform synchronization during symbolic execution.

Noninterference. IPR shows that an HSM's implementation leaks no more information than its specification state machine. However, the specification may have bugs that allow for information leakage. Prior work has developed techniques that could be used to rule out such bugs. For example, noninterference [35] is a property ensuring that secret data cannot influence public outputs. A range of formal methods have been developed for proving noninterference and analyzing information flow [39, 51, 54, 60, 70]. These approaches are complementary to Parfait and the guarantees provided by IPR.

Translation validation. Parfait does not verify the correctness of SoC hardware in general, or prove that a compiler always preserves non-leakage. Instead, Parfait checks that an SoC, a particular circuit containing a particular firmware binary, is related by IPR to a specification. This is a form of translation validation [53, 56, 61, 68], which is traditionally used for checking that a particular output of a compiler refines it source. 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 captured by IPR and validates execution down to the level of hardware.

Process isolation. Parfait targets HSMs running a single application. Other work addresses the problem of leakage between processes in systems with multitasking.

Prior verified hypervisors have proved noninterference and other information flow properties between processes [26, 33, 36, 37, 43, 51, 52, 54, 62]. However, these proofs do not cover leakage through lower-level microarchitectural state or timing.

Ge et al. [34] have extended the verified seL4 kernel with mechanisms to prevent microarchitectural leakage between security domains, by among other things, using instructions to reset such state on domain switches. Sison et al. [63] have formalized the security guarantees provided by this approach under an abstract model of OS and hardware behavior.

10 Conclusion

Parfait uses transitive information-preserving refinement (IPR) to verify that the combined hardware and software implementation of an HSM correctly implements its specification and leaks nothing more than what is required by the spec. Case studies demonstrate that transitive IPR enables Parfait to scale to sophisticated HSMs, such as those implementing public-key cryptography; that Parfait specifications are succinct compared to the implementation (40 lines of specification for the ECDSA-signing HSM, compared to 2,300 lines of code and 13,500 lines of Verilog for the implementation); and that Parfait requires modest effort to port a new app or hardware platform (2 hours of developer time each for the password-hasher app and the PicoRV32 platform).

Acknowledgments

We would like to thank Nikhil Swamy, Jonathan Protzenko, and the F* community for their guidance on working with F*. This paper has been improved thanks to feedback from Alexandra Henzinger, Baltasar Dinis, Derek Leung, Hannah Gross, Joshua Gancher, Noah Moroze, Sanjit Bhat, Stella Lau, Upamanyu Sharma, the anonymous reviewers, and our shepherd, Manos Kapritsos. This research was supported by NSF award CNS-2225441.

References

[1] CVE-2004-0320. https://nvd.nist.gov/vuln/detail/CVE-2004-0320, September 2004.

[2] YSA-2015-1. https://developers.yubico.com/ykneo-openpgp/SecurityAdvisory%202015-04-14.html, April 2015.

[3] CVE-2018-6875. https://nvd.nist.gov/vuln/detail/CVE-2018-6875, February 2018.

[4] YSA-2018-01. https://www.yubico.com/support/security-advisories/ysa-2018-01/, January 2018.

[5] CVE-2019-18671. https://nvd.nist.gov/vuln/detail/CVE-2019-18671, November 2019.

[6] CVE-2019-18672. https://nvd.nist.gov/vuln/detail/CVE-2019-18672, November 2019.

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

[8] CVE-2021-31616. https://nvd.nist.gov/vuln/detail/CVE-2021-31616, April 2021.

[9] 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, August 2002.

[10] 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)*, pages 71–85, Edinburgh, United Kingdom, August 2010.

[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Security Symposium*, pages 53–70, Austin, TX, August 2016.

[12] ARM Limited. ARM Cortex-M3 processor technical reference manual. https://developer.arm.com/documentation/100165/latest/, November 2016. Revision r2p1.

[13] Anish Athalye. *Formally Verifying Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation*. PhD thesis, Massachusetts Institute of Technology, August 2024.

[14] 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)*, pages 503–519, Carlsbad, CA, July 2022.

[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)*, pages 215–228, Oxford, United Kingdom, July 2018.

[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*, pages 777–795, Virtual conference, May 2021.

[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, January 2020.

[18] Jean-Baptiste Bédrune and Gabriel Campana. Everybody be cool, this is a robbery! https://donjon.ledger.com/BlackHat2019-presentation/, August 2019.

[19] William R. Bevier, Warran A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[20] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27(1):137–163, 2019.

[21] Brett Boston, Samuel Breese, Josiah Dodds, Mike Dodds, Brian Huffman, Adam Petcher, and Andrei Stefanescu. Verified cryptographic code for everybody. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV)*, pages 645–668, Los Angeles, CA, July 2021.

[22] Helena Brekalo, Raoul Strackx, and Frank Piessens. Mitigating password database breaches with Intel SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX)*, Trento, Italy, December 2016.

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

[24] Dominic Chell. Apple iOS hardware assisted screenlock bruteforce. https://www.mdsec.co.uk/2015/03/apple-ios-hardware-assisted-screenlock-bruteforce/, March 2015.

[25] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 software. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 299–309, Scottsdale, AZ, November 2014.

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

[27] Filippo Cremonese. Security analysis of the Solo firmware. https://blog.doyensec.com/2020/02/19/solokeys-audit.html, February 2020.

[28] 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)*, pages 337–340, Budapest, Hungary, March–April 2008.

[29] 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*, pages 73–90, San Francisco, CA, May 2019.

[30] 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.

[31] 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.

[32] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, March 2018.

[33] 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)*, pages 287–305, Shanghai, China, October 2017.

[34] 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, March 2019.

[35] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982.

[36] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 595–608, Mumbai, India, January 2015.

[37] 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)*, pages 653–669, Savannah, GA, November 2016.

[38] 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*, pages 1868–1883, Virtual conference, May 2021.

[39] 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)*, pages 165–181, Broomfield, CO, October 2014.

[40] 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)*, pages 219–235, Berlin, Germany, November 2013.

[41] 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). *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4): 281–308, 2020.

[42] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[43] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the Arm confidential compute architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 465–484, Carlsbad, CA, July 2022.

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

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

[46] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

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

[48] 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)*, pages 78–92, Worcester, MA, August 2000.

[49] 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*, pages 2057–2073, Virtual conference, August 2020.

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

[51] 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*, pages 126–142, Kyoto, Japan, December 2012.

[52] 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*, pages 415–429, San Francisco, CA, May 2013.

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

[54] Luke Nelson, James Bornholt, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Noninterference specifications for secure systems. *ACM SIGOPS Operating Systems Review*, 54(1):31–39, August 2020.

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

[56] 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)*, pages 151–166, Lisbon, Portugal, March–April 1998.

[57] Thomas Pornin. BearSSL constant-time mul. https://bearssl.org/ctmul.html.

[58] 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, September 2017.

[59] 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*, pages 983–1002, San Francisco, CA, May 2020.

[60] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[61] 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)*, pages 471–482, Seattle, WA, June 2013.

[62] 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)*, pages 287–306, Carlsbad, CA, October 2018.

[63] 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)*, pages 103–121, Lübeck, Germany, March 2023.

[64] Richard M. Stallman. Using the GNU compiler collection. https://gcc.gnu.org/onlinedocs/gcc/.

[65] 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)*, pages 256–270, St. Petersburg, FL, January 2016.

[66] The Coq Development Team. *The Coq Proof Assistant, version 8.17.1*, June 2023. URL https://doi.org/10.5281/zenodo.8161141.

[67] 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)*, pages 530–541, Edinburgh, United Kingdom, June 2014.

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

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

[70] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[71] 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)*, pages 2128–2142, Copenhagen, Denmark, November 2023.

[72] 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)*, pages 77:1–77:29, Cascais, Portugal, January 2019.

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

[74] Claire Xenia Wolf. PicoRV32 – a size-optimized RISC-V CPU. https://github.com/YosysHQ/picorv32, 2015.

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

[76] 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, October 2005.

[77] 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, October–November 2017.