Verifying Hardware Security Modules with Information-Preserving Refinement

Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

Knox is a new framework that enables developers to build hardware security modules (HSMs) with high assurance through formal verification. The goal is to rule out all hardware bugs, software bugs, and timing side channels.

Knox's approach is to relate an implementation's wire-level behavior to a functional specification stated in terms of method calls and return values with a new definition called *information-preserving refinement (IPR)*. This definition captures the notion that the HSM implements its functional specification, and that it leaks no additional information through its wire-level behavior. The Knox framework provides support for writing specifications, importing HSM implementations written in Verilog and C code, and proving IPR using a combination of lightweight annotations and interactive proofs.

To evaluate the IPR definition and the Knox framework, we verified three simple HSMs, including an RFC 6238-compliant TOTP token. The TOTP token is written in 2950 lines of Verilog and 360 lines of C and assembly. Its behavior is captured in a succinct specification: aside from the definition of the TOTP algorithm, the spec is only 10 lines of code. In all three case studies, verification covers entire hardware and software stacks and rules out hardware/software bugs and timing side channels.

1 Introduction

A powerful approach for building secure computer systems is to factor out the core security functionality onto a separate device. For example, on the server side, certificate authorities use hardware security modules (HSMs) to store their signing key and sign certificates [10, 58]; credit card networks use HSMs for pin translation, secure re-encryption of payment requests during routing; cloud providers use HSMs to safeguard PIN-protected backup keys [9, 43, 47]; and some tax authorities require the use of an HSM to timestamp invoices. On the client side, the iPhone uses its secure enclave processor to enforce PIN guessing limits for unlocking the phone [15]; and users often rely on USB security keys to protect their authentication private key in the face of a compromised computer [65]. For simplicity, this paper refers to all of these types of devices as HSMs. These devices are in widespread use; e.g., there are hundreds of millions of deployed secure enclaves and security keys.

This approach defends against a broad class of attacks where an adversary gains access to any host computer that the HSM might be connected to, regardless of the specific attack vector (exploiting a buffer overflow, missing access control checks, or even gaining access to the administrator's SSH key). As long as the security of the overall system is rooted in the device, an adversary that controls the host cannot undermine the security of the overall system. Of course, the device must be correctly implemented to make sure that the adversary cannot compromise it, which in practice means that the device must provide simple, well-defined functionality.

Although HSMs are relatively simple, any vulnerability in their hardware or software can undermine their security. HSMs have suffered from bugs throughout the hardware/software stack, such as logic bugs, memory corruption, hardware bugs, and timing side channels [1–8, 21, 31, 45, 51, 68]. This paper presents an approach for ruling out such bugs through formal verification, with a particular focus on eliminating leakage through timing side channels.

Our approach is to relate the behavior of the HSM 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 shows the implementation of a simplified PIN-protected backup HSM, which we use as a running example through the paper. The host connects to this HSM via two input wires and two output wires, which the host can read/write at every cycle. Figure 2 shows the functional specification for this HSM. It exposes two operations, `store` and `retrieve`. The specification does not have an operation for reading back the PIN, and it enforces a guess limit on PINs.

We relate a physical implementation to a functional specification with a new definition called *information-preserving refinement (IPR)*, inspired by definitions of zero-knowledge proofs in cryptography [40, 41]. IPR captures the notion that the implementation 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 computer can follow to get correct results from the HSM, describing how each spec-level operation translates to wire-level I/O with the HSM. The driver is a part of the specification (and is trusted). Its dual, an *emulator*, is a proof artifact that describes how wire-level behavior can be explained in terms of spec-level operations. 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), its outputs reveal no more information than the specification.

Applied to HSMs, IPR can capture subtle security bugs: for example, Figure 3 shows code that is correct and even crash safe but has a subtle bug involving persistence and

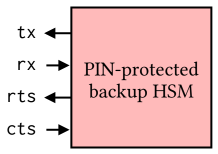

Figure 1: The physical implementation of the PIN-protected backup HSM. It connects to the host via 4 wires, speaking UART with flow control.

```
var bad_guesses = 0, secret = 0, pin = 0

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

def retrieve(guess):
  if bad_guesses >= 10:
    return 'No more guesses'
  if guess == pin:
    bad_guesses = 0
    return secret
  bad_guesses = bad_guesses + 1
  return 'Incorrect PIN'
```

Figure 2: A functional specification for a PIN-protected backup HSM. The spec doesn't support reading out the PIN, and retrieval of the secret requires supplying the correct PIN. Limiting guesses prevents brute-forcing.

```
// return error if PIN guess limit exceeded
// ...

// check PIN guess and update bad_guesses accordingly
if (!constant_time_cmp(&entry->pin, guess)) {
    entry->bad_guesses++;
    uart_write(ERR_BAD_PIN);
    return;
}
entry->bad_guesses = 0;

// output secret
// ...
```

Figure 3: Code snippet from an insecure `retrieve` implementation. `entry` points to persistent memory. The commit point depends on whether the PIN guess is correct.

timing. The way this code gets compiled, the circuit takes longer to persist the incremented guess count, in the case of an incorrect guess, than it takes to zero the guess count, in the case of a correct guess (it takes longer to take the branch than to fall through). This can be abused to reset the guess count by repeatedly guessing every possible PIN and powering off the device after just enough cycles to reset the guess count in the case that the guess is correct (but not waiting long enough to persist if the guess is incorrect). Verifying IPR caught this bug in our implementation (§7.1.1). The buggy implementation doesn't enforce guess limits, which leaks more information than the specification, and this is prohibited by IPR.

Existing security definitions like noninterference or declassification either do not apply or are insufficient to capture the security of wire-level observations and arbitrary wire-level I/O as in the Knox setting (§9).

To be able to verify HSMs with IPR, we developed the Knox framework. Developers using Knox write HSM implementations using standard languages (i.e., Verilog and C code), write specifications in Knox DSLs, and use a combination of lightweight annotations and interactive proofs to show that the implementation is an information-preserving refinement of the specification.

To demonstrate that IPR and the Knox framework can be applied to HSMs and catch bugs in their implementations, we developed and verified three HSMs: a PIN-protected backup HSM, a password-hashing HSM, and an RFC 6238-compliant TOTP token [54]. The Knox HSMs do not have the implementation complexity of commercial HSMs: for example, the RISC-V processor they use is simpler than the ARM Cortex-M series embedded processors ubiquitous in security tokens such as SoloKeys. Still, the HSMs demonstrate many of the hardware and software complexities present in real HSMs. They all use an embedded processor (a RISC-V CPU) and interface with the host via digital I/O (UART), and the password hasher and TOTP token include hardware cryptographic accelerators. All three run application-specific C code, with some including cryptographic functionality, such as HMAC in the TOTP token. Knox proofs are end-to-end, encompassing hardware and software and showing that the implementation is free of exploitable hardware bugs, software bugs, and timing side channels.

In summary, this paper makes the following contributions:

- The definition of information-preserving refinement (IPR), which relates a physical implementation to a functional specification and captures that it: (1) implements the specification, and (2) leaks no additional information
- The Knox framework for proving that an HSM implementation satisfies its specification under the IPR definition
- An evaluation of the IPR definition and Knox's application to three simple HSMs

This paper applies IPR to HSMs, but we believe the definition is broadly applicable to other contexts for capturing non-leakage properties.

This paper has several limitations. The three HSMs verified using Knox are relatively simple: for example, they do not use public-key signatures, which are common HSM operations, because it is difficult to scale up proofs in Knox to handle sophisticated arithmetic needed for public-key implementations. Relatedly, for cryptographic operations such as public-key signatures, IPR requires the emulator to be efficient. Knox currently relies on a manual audit to ensure that the emulator does not brute-force secrets or run in exponential time (§8.1). Finally, IPR does not support true random number generators (TRNGs) — the functional specification has to be deterministic. We believe that a pseudo-random number generator is a reasonable workaround that fits into IPR (§8.2).

2 Threat model and security goal

This paper considers a powerful 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 computer 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 computer is well-behaved.

Our threat model is focused on remote compromise of the host machine, one of the primary attacks that HSMs aim to defend against, so it does not include physical attacks on the HSM. While the adversary can perform arbitrary digital I/O to the HSM through a compromised host, remote compromise is unlikely to allow the adversary to violate the HSM's electrical specifications (e.g., supply 5V into an input wire expecting 3.3V logic or supply current to an output pin of the HSM) or observe analog characteristics of the I/O interface (e.g., measure analog voltage on a pin).

While the threat model includes (digital) timing side channels due to the level at which we model the host-HSM interface (wire-level I/O at every cycle), the threat model does not include arbitrary side channels [73] such as electromagnetic radiation [12], temperature [44], and power [49], because a remote attacker is unlikely to able to make such observations.

The goal of a Knox HSM is to be as secure as its specification. A host machine should be able to follow an I/O protocol to invoke spec-level operations on the HSM and obtain the correct outputs, but the host machine should not be able to abuse the wire-level interface to subvert the HSM and bypass its API or cause it to leak secrets.

3 Information-preserving refinement

The goal of information-preserving refinement (IPR) is to define what it means for an implementation with a wire-level physical interface to implement a functional specification and leak no additional information. IPR achieves this by establishing a bi-level correspondence between implementation and specification, at both the level of the functional interface (spec-level operations) and the physical interface (wire-level I/O). Illustrated in Figure 4, IPR is defined as an indistinguishability between two worlds: the real world, and an ideal world that is correct and secure by construction.

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 a *driver* that is part of the specification. The driver translates spec-level operations to wire-level I/O, describing how the host invokes

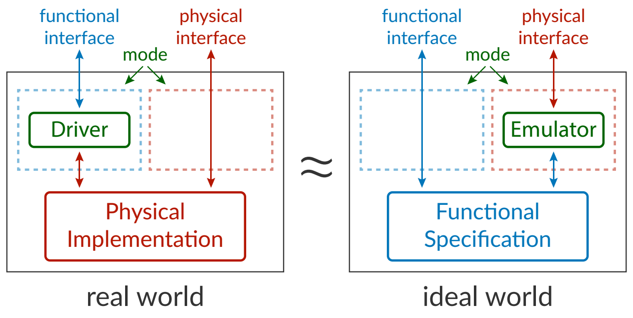

Figure 4: Information-preserving refinement (IPR), defined as an indistinguishability between a real world and an ideal world that is correct and secure by construction.

the operation and reads the return value by interacting with the HSM over its wire-level interface.

The ideal world is set up to provide the same interface as the real world but be correct and secure by construction. In the ideal world, a host machine that takes a functional view of the device invokes operations directly on the specification. To provide a physical view, an *emulator* mimics wire-level behavior, given only query access to the functional specification. The emulator is a dual of the driver; it translates wire-level I/O into spec-level operations. Unlike the driver, the emulator is merely a proof artifact. The ideal world can be instantiated with *any* emulator, and it remains secure by construction. IPR is defined to hold if there exists some emulator such that the real and ideal worlds are indistinguishable.

The host can switch 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). When switching views from physical to functional, in the real world, the driver is re-initialized; when switching from functional to physical, in the ideal world, the emulator is re-initialized.

The ideal world is correct and secure by construction. 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 mimic 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, maps to some sequence of operations invoked on the specification.

Due to the indistinguishability that IPR requires between real and ideal worlds, any execution in the real world also maps to some sequence of operations invoked on the specification. In other words, when IPR holds, any attack that an adversary could execute on the real device could be transformed into an attack on the specification itself: the adversary could run the emulator and then execute the original attack using the emulator, which matches the implementation's wire-level behavior, given only query access to the specification. Indistinguishability between real and ideal worlds guarantees that the implementation is as secure as the specification.

Definition (Information-preserving refinement)**.** A physical implementation is an *information-preserving refinement* of a functional specification with respect to a driver if there exists an emulator such that the real world is indistinguishable from the ideal world as illustrated in Figure 4. ■

3.1 Applying IPR to HSMs

IPR, without explicitly talking about hardware, software, or timing side channels, captures exploitable bugs in all of those. If there were such exploitable bugs, IPR would not be satisfied: when there are implementation behaviors that can't be explained in terms of the specification, there does not exist an emulator that makes the real world and ideal world indistinguishable.

IPR relates any wire-level interaction with the HSM to an interaction with the specification. For example, suppose that the host machine follows the driver to perform a number of spec-level operations, and then it gets compromised, at which point it begins performing arbitrary I/O in an attempt to subvert the HSM. IPR, by requiring indistinguishability between the real and ideal worlds, says that this scenario corresponds to some sequence of spec-level operations, and that the arbitrary I/O reveals no more information than those spec-level operations do. Furthermore, IPR says that after the HSM is moved to an uncompromised host, normal operation can resume (as the host follows the driver), and that the behavior of the device will reflect any specification state changes that were a result of queries made by the emulator (any operations that were effectively invoked during arbitrary wire-level I/O).

The definition directly addresses host machine compromise by an adversary while the host is in between spec-level operations. It might seem like IPR only addresses arbitrary I/O that begins between these operations; however, a compromise in the middle of an operation can be thought of as a compromise that happens slightly earlier, at the start of the operation, and IPR covers this case.

Information-preserving refinement transfers both cryptographic and non-cryptographic security properties from the specification to the implementation. For example, the PIN-protected backup specification limits PIN guesses, and so IPR implies that the implementation enforces the guess limit as well. If it didn't limit guesses, it would reveal more information than the specification (through subsequent retrieve operations), which IPR prohibits. This rules out the subtle bug shown in Figure 3, even though the information disclosure manifests after the buggy code executes. If a specification computed signatures without revealing a key, then IPR would imply that the implementation also doesn't leak the key, including through its timing behavior.

4 Proving IPR

Knox models the specification and the implementation (§4.1) as state machines, relates the two with a refinement relation, and proves three properties: an initialization property (§4.2), functional equivalence (§4.3, indistinguishability of the functional view), and physical equivalence (§4.4, indistinguishability of the physical view), tying together these properties with the refinement relation. Together, these properties imply IPR.

4.1 Physical implementation

Knox models HSM implementations with a cycle-accurate description of their wire-level I/O behavior, covering hardware and software. Figure 1 shows the interface of a circuit implementing PIN-protected backup. The HSM interface allows for: (1) setting input wires, (2) reading output wires, and (3) waiting for the HSM to execute for a clock cycle of the HSM's internal clock.

In the case of the PIN-protected backup HSM, the UART `rx` and `cts` wires can be set and the `tx` and `rts` wires can be read at every cycle. The baud rate is independent of the HSM clock frequency; the IPR formalism itself has no notion of a serial port or baud rate, only wires and hardware-level clock cycles. The three main Knox case studies use UART, but simpler Knox examples use different I/O protocols.

The HSM model comprises the circuit state, a step function describing behavior for a single cycle, the initial state of the HSM (contents of non-volatile memory, such as ROM containing code and read-write persistent memory being zero-initialized), and a description of the power-on / reset behavior of the circuit (losing the contents of volatile memory).

4.2 Refinement relation and initialization

Knox uses a refinement relation R, a proof artifact supplied by the developer, to relate the state of the implementation to the state of the specification *in between spec-level operations*. That is, it is not required to hold at arbitrary steps of the circuit, only before/after spec-level operations, or after switching from the physical view to the functional view, which involves re-initializing the driver (which in our implementations, resets the circuit). Use of a common R connects functional equivalence and physical equivalence.

R relates states and usually includes an invariant that captures circuit quiescence (it holds in between spec-level operations). Figure 5 shows the refinement relation used in

$$\begin{aligned}
\text{spec.bad\_guesses} &= \text{swap32}(\text{impl.fram}[0..3]) & \wedge \\
\text{spec.pin} &= \text{impl.fram}[4..9] & \wedge \\
\text{spec.secret} &= \text{impl.fram}[10..19] & \wedge \\
&\text{Inv}(\text{impl})
\end{aligned}$$

Figure 5: A simplified version of the refinement relation used in the proof of PIN-protected backup. *impl.fram* refers to the persistent memory of the implementation. *swap32* performs a byte order swap. *Inv* is the invariant (not shown here).

```
(define (store secret pin)
  (send-byte #x02) ; command number
  (send-bytes pin)
  (send-bytes secret)
  (recv-byte)) ; wait for ack

(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: A code snippet from the PIN-protected backup driver. The function corresponding to a spec-level operation is shown in blue. Driver-language primitives are in red.

the proof of the PIN-protected backup HSM. It relates each variable in the specification to the persistent memory of the circuit.

Knox requires that the initial implementation state is related by R to the initial specification state.

4.3 Functional equivalence

Functional equivalence states that spec-level behavior is obtained from the implementation's wire-level interface by following the I/O protocol described by the driver. The driver is a program, written in Knox's driver language, that is part of the specification of the HSM. For every spec-level operation, the driver has a corresponding function that describes how the host invokes the operation on the HSM over its wire-level I/O interface.

For example, Figure 6 shows the driver for the PIN-protected backup HSM. The driver exposes a function corresponding to each spec-level function, such as `(store ...)`, implemented in terms of driver-language primitives for interacting with the implementation: `(set-input ...)` and `(get-output ...)` write the input wires and read the output wires, respectively; `(tick)` waits for the HSM to execute for a single cycle; `(yield)` models situations where the host is allowed to wait for an arbitrary number of cycles, e.g., in between sending bytes in an asynchronous protocol.

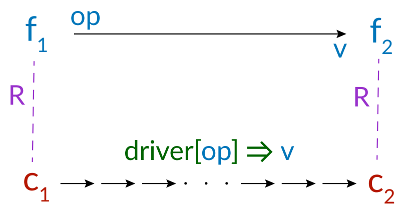
$$\begin{tikzcd}
f_1 \arrow[rrrrrrr, "op"] \arrow[d, dash, dashed, "R"'] & & & & & & & f_2 \arrow[d, dash, dashed, "R"] \\
c_1 \arrow[r] & \cdot \arrow[r] & \cdot \arrow[r] & \cdots & \cdot \arrow[r] & \cdot \arrow[r] & \cdot \arrow[r] & c_2
\end{tikzcd}$$

Figure 7: Functional equivalence: for all implementation states c_1 and spec states f_1 that are related by R, and for all spec-level operations op:
(1) the spec-level output v matches the driver output
(2) the final states c_2 and f_2 are related by R

Figure 7 defines functional equivalence: starting from circuit/spec states related by R, invoking an operation on the specification gives the same result as running the corresponding driver function against the circuit, and the final circuit/spec states continue to be related by R.

The HSM runs asynchronously from the host: its clock keeps ticking even if there is no operation to perform. To model this, the driver also describes a spec-level no-op: e.g., in the case of the PIN-protected backup HSM, the host sets the `rx` line high, indicating that it has nothing to transmit. Functional equivalence also covers this no-op case.

4.4 Physical equivalence

Physical equivalence states that wire-level behavior matching the real circuit's behavior can be obtained by running an emulator (with query access to the specification), capturing the notion that the circuit leaks no more information than the specification. The emulator in IPR is a dual of the driver: it is a program, written in Knox's emulator language, that implements wire-level interactions in terms of spec-level operations. Unlike the driver, the emulator is a proof artifact: if there exists an emulator that mimics circuit behavior, then physical equivalence holds.

An emulator exposes a function corresponding to each wire-level interaction: setting the input, getting the output, and running for a cycle. These are implemented in terms of emulator-language primitives for invoking spec-level operations (e.g., `(store ...)` and `(retrieve ...)`, for the PIN-protected backup). Besides the ability to make black-box queries to the functional specification, the emulator can maintain auxiliary state across emulating multiple cycles; the auxiliary state is initialized to a null value whenever the emulator is re-initialized.

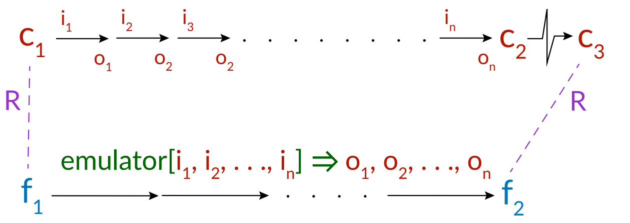

Figure 8: Physical equivalence: for all spec states f_1 and implementation states c_1 that are related by R, and for all wire-level inputs $i_1 \ldots i_n$:
(1) the circuit outputs $o_1 \ldots o_n$ match the emulator outputs
(2) the final states f_2 and c_3 (c_2 after a reset) are related by R

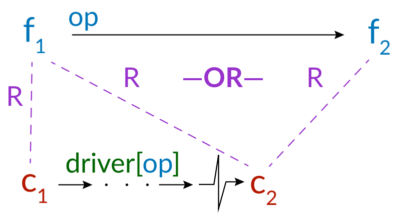

Figure 9: Crash safety: for all implementation states c_1 and spec states f_1 that are related by R, and for all spec-level operations op: if the driver is interrupted at any point, the post-reset state of the circuit c_2 is related by R to *either* f_1 or f_2.

Figure 8 defines physical equivalence: starting from circuit/spec states related by R, any wire-level I/O behavior exhibited by the circuit is matched by the emulator, which makes queries to the specification as it runs. Furthermore, the final specification state is related by R to the final circuit state (after the circuit is reset).

IPR is satisfied as long as there exists some emulator such that the real and ideal worlds are indistinguishable. Proofs of physical equivalence in Knox involve constructing an emulator (i.e., writing a program in the emulator language) that satisfies the definition of physical equivalence. Because the emulator is merely a proof artifact, the details of the construction do not matter, as long as the program satisfies the definition. The Knox case studies (§7) describe the techniques used in practice to write emulators.

4.5 Crash safety

Physical equivalence already covers the case of an interrupted spec-level operation, because an interrupted protocol-following execution can be viewed as a case of arbitrary I/O: physical equivalence guarantees that any wire-level I/O corresponds to *some* sequence of spec-level operations. However, we can state an additional property that is stronger: when the HSM is interrupted in the middle of an operation while the host is following the driver, the implementation is crash safe, acting either as if the operation never started or as if the operation completed successfully. Figure 9 defines this crash-safety property.

5 The Knox framework

The Knox framework uses hybrid symbolic execution [67] and SMT solvers to help developers prove IPR. Knox includes techniques to handle the challenges that arise when applying symbolic execution for proving functional equivalence and physical equivalence. In functional equivalence proofs, Knox handles the nondeterminism of `yield` in drivers by automatically finding fixed points (§5.1). In physical equivalence proofs, Knox supports reasoning about unbounded-length inputs using an approach we call *guided symbolic model checking* (§5.2). In both, Knox allows the proof developer to supply *hints*, untrusted guidance where the framework invokes the solver as necessary to ensure soundness (§5.3).

5.1 Nondeterminism

Knox verifies the functional equivalence property using symbolic execution of the driver-language program against the HSM implementation, comparing the execution of each driver operation against the corresponding spec operation. 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 $[\![S]\!] = \{[\![t|p]\!] : t|p \in S\}$. 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, i.e., $[\![S]\!] \subseteq [\![S']\!]$, which addresses case explosion.

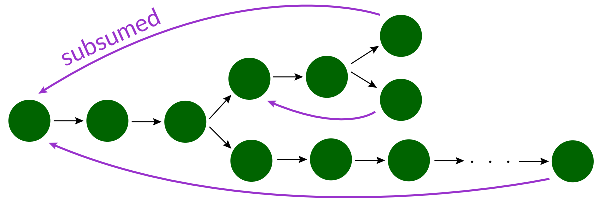

Figure 10: An illustration of *guided symbolic model checking* exploring a state space. Each green circle is a symbolic term representing a set of states. Black arrows show STEP invocations and purple arrows show SUBSUMED invocations.

5.2 Unbounded-length inputs

In Knox, emulators can be symbolically executed with black-box query access to a functional specification. Unlike the functional equivalence property which considers a single (spec-level) input, the physical equivalence property considers an arbitrary-length sequence of (wire-level) inputs, so Knox can't prove the physical equivalence 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 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 equivalence, 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 auxiliary state, null), and functional spec state f_1, where both f_1 and c_1 are symbolic terms, and 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 10 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* (§5.3) 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.

5.3 Hints

In both functional equivalence proofs and physical equivalence proofs, relying only on hybrid symbolic execution quickly results in an explosion in term size, and in the case of HSMs involving cryptography, queries that make the SMT solver time out.

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 an 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 Implementation

The Knox framework builds on top of Racket [37] and the Rosette solver-aided programming language [67], and it relies on the Z3 SMT solver [32]. To compile circuits to a shallow embedding in Rosette, Knox uses GCC and its RISC-V backend to compile C code, Yosys [69] and its SMT-LIB backend to process Verilog, and `#lang yosys` (700 LOC of Racket and Rosette) from Notary [16] to convert the SMT-LIB output into a Rosette model.

The Knox framework's core — the semantics for the driver and emulator languages, and the tools for verifying functional equivalence and physical equivalence — is implemented in 3000 lines of Racket and Rosette. Achieving good verification performance required many optimizations and some new techniques, including symbolic state serialization, term substitution, fixpoint finding, state merging, and a new algorithm for symbolic subsumption checking based on a disjoint-set data structure.

The case studies are implemented in Verilog, C, and assembly (summarized in Figure 16). The case studies run on a $65 1BitSquared iCEBreaker development board, which has a Lattice iCE40UP5K FPGA, and use an open-source FPGA toolchain: the Yosys synthesis tool, the nextpnr place and route tool [72], and Project IceStorm [70] to create the bitstream and flash the FPGA. The FPGA connects using an FTDI cable to a host computer running Linux, for which we wrote client libraries for the three HSMs.

Figure 11 shows an overview of the different components that the developer writes when using Knox to verify an HSM. The functional specification, physical implementation, and refinement relation *R* are common inputs, used when verifying both functional equivalence and physical equivalence. When verifying functional equivalence, Knox takes as additional input the driver, along with hints to guide symbolic execution. When verifying physical equivalence, Knox takes as additional input the emulator and a proof script. The functional specification and the driver, highlighted in green, comprise the code written by the developer that is trusted. Other components, the HSM implementation and proof artifacts, are verified by the framework. Similar to other tools based on symbolic execution, when verification in Knox fails, the framework can provide a concrete counterexample, aiding the developer in debugging the implementation or the proof.

Source code for Knox and the case studies is available at `https://github.com/anishathalye/knox`.

7 Evaluation

To evaluate information-preserving refinement and the Knox framework, we ask the following questions:

- Can IPR and Knox be applied to HSM hardware/software?
- What types of bugs does verification prevent?
- What is the performance of the Knox framework?
- What is the performance of HSMs verified with Knox?

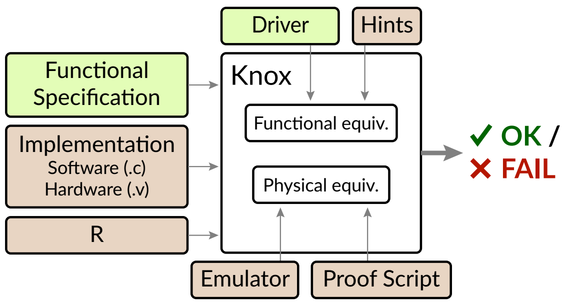

Figure 11: An overview of the Knox workflow. Trusted inputs are shown in green.

Methodology. We evaluate the first two questions through case studies (§7.1) that formally verify three HSMs with different types of specification and implementation complexity: a PIN-protected backup HSM, a password-hashing HSM, and an RFC 6238-compliant TOTP token [54]. To answer questions related to verification performance and the performance of the HSM implementations, we report on measurements (§7.2).

7.1 Case studies

7.1.1 PIN-protected backup HSM

Specification. A simplified PIN-protected backup HSM (Figure 2) was a running example through this paper; we verified an HSM with additional functionality: storing multiple secrets, each protected by its own PIN, and indexed by a slot number. The specification exposes four functions: `status`, `store`, `retrieve`, and `delete`. The specification demonstrates support for non-cryptographic security properties, such as the guess limit on PINs.

Implementation. Figure 12 shows a schematic of the implementation. It uses the PicoRV32 RISC-V CPU and the SimpleUART peripheral from the PicoSoC [71] with minimal modifications: we removed asynchronous reset from the CPU and added hardware flow control to the UART. The HSM uses ferroelectric RAM (FRAM) for persistent storage. Knox requires cycle-accurate models of the entire hardware, and FRAM has simple cycle-precise behavior, supporting durable word-level writes in a single cycle. For convenience, to avoid wiring an external chip, the prototype uses FPGA Block RAM in place of FRAM for the experiments. In total, the HSM hardware is described in 2670 lines of Verilog.

The software is written in a combination of C and assembly. To simplify verification, the HSM uses a strategy inspired by Notary [16] to minimize variation in the states that the hardware can be in. The HSM uses a reset-based design: the SoC is held in an "embryo" state until the host is ready to perform an operation, and after the HSM performs a single operation, it enters the embryo state again until the host begins the next operation. This is done through a combination of hardware and software: the HSM's `cts` input doubles as a signal that the host is ready to perform an operation, and a

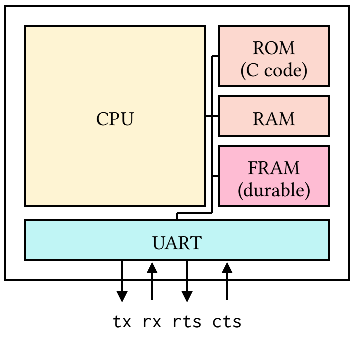

Figure 12: A schematic of the PIN-protected backup HSM.

small amount of logic implemented in hardware holds the rest of the circuit in a reset state until the host is ready to perform an operation. After the HSM performs a single operation, it signals this power management hardware to reset the SoC and return to the embryo state.

The software on the HSM includes a driver for the memory-mapped UART peripheral, along with code to implement each of the operations. The `main` function of the HSM reads a command and arguments from the UART, calls the appropriate handler, and then shuts down. The code for the HSM is written in 150 lines of C and 40 lines of assembly.

Verification and bugs caught. Knox physical equivalence proofs are constructive. We designed the emulator for the PIN-protected backup as follows. The emulator runs a copy 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 copy of the circuit: when the circuit is about to perform 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 output behavior of the circuit copy matches that of the real circuit. For example, for the retrieve operation, when the emulator sees that its circuit copy has just completed the equality comparison, the emulator extracts the slot number and PIN guess from the circuit copy's RAM, makes a `retrieve` query to the specification, and injects the result (match or no match) back into its copy of the circuit, also injecting the secret into the appropriate location in memory if the guess indeed matched. All of the emulators for the case studies follow this general construction. Through the physical equivalence proof, we show that all implementation-level behavior can be explained with spec-level behavior, proving that the implementation leaks no more than the spec.

Verifying physical equivalence catches classic security bugs, such as a bug where the implementation's timing behavior leaks how many bytes of the PIN guess matches due to using `strcmp`. This information is not revealed by the spec — which only reveals whether or not the guess is correct (and the secret, when the guess is correct), not how many bytes of the guess match the PIN — so IPR prevents the implementation from leaking it. For such a buggy implementation, an emulator satisfying the IPR definition doesn't exist: the emulator doesn't have direct access to the true PIN (only query access through the specification), so its behavior can't match the real circuit's leaky behavior.

Verifying physical equivalence caught a subtle security bug involving persistence and timing. Figure 3 shows a code snippet from the insecure implementation. The compiler happens to compile this code using a branch instruction, branching in the case where the guess is incorrect, and the CPU implementation takes longer to take the branch than to fall through to the next instruction. This has the effect that the circuit takes longer to write the updated `bad_guesses` value to persistent memory in the case of an incorrect PIN guess (where it's executing `entry->bad_guesses++`) than in the case of a correct PIN guess (where it's executing `entry->bad_guesses = 0`). An attacker can abuse this to reset the guess count by guessing a PIN, powering off the device after just enough cycles to reset the guess count in the case that the guess is correct (but not waiting long enough that `entry->bad_guesses++` has a chance to run, in the case that the guess is incorrect), and repeating this process for every possible PIN. This is not a correctness or even a crash-safety bug: this insecure implementation is both correct and crash safe. However, physical equivalence prevents this security bug in the implementation. The bug is fixed by using constant-time code to make the commit point of the operation independent of whether the PIN guess is correct.

```
var secret = 0

def config(new_secret):
  secret = new_secret

def hash(password):
  return sha256(password || secret)
```

Figure 13: The functional specification for the password-hashing HSM.

7.1.2 Password-hashing HSM

Specification. Figure 13 outlines the specification for the password-hashing HSM. It includes a specification of SHA256 (not shown) that follows FIPS 180-4 [57]. The password hasher is configured with a secret, and then it computes salted hashes using the stored secret. There is no function to retrieve the secret after it is stored. The specification also guarantees that future operations cannot leak past inputs, because the secret cannot be read back, and passwords are not stored.

Implementation. The hardware is similar to that of the PIN-protected backup HSM. This HSM adds a hardware SHA256 cryptographic accelerator (about 300 lines of Verilog), which implements the SHA256 block function. We originally used an off-the-shelf SHA256 core [66], but we switched to a custom implementation to minimize FPGA area so the design would fit on our low-cost board. The software includes a driver for the SHA256 peripheral, which implements the message padding function and drives the memory-mapped SHA256 peripheral, one block at a time. The software is written in 200 lines of C code and 40 lines of assembly.

The HSM needs to be crash safe, and the specification has operations that require updating multi-word values (the secret, in our specification, is 20 bytes), but the FRAM only supports word-level (32-bit) atomic writes. For this reason, the HSM uses a simple journaling strategy where it keeps two copies of the state in persistent memory and uses a flag to determine which one is active. To do an atomic write, the HSM writes to the inactive region and then toggles the flag.

Verification and bugs caught. The functional equivalence proof caught a bug in the hardware implementation, in the integration of the SHA256 peripheral into the SoC. The memory-mapped SHA256 peripheral's chip select input was being set based on the CPU's `mem_addr` bus, but the `mem_addr` is uninitialized on reset (it becomes stable after a couple cycles), so the SHA256 hardware could receive an unintended command right at boot. This bug would be difficult to catch through testing because it is only triggered in rare cases, when the uninitialized address bus contains a particular value on reset. The bug was fixed by adding an additional condition that `mem_valid` was also asserted (which all the other peripherals did, but the SHA256 peripheral didn't when it was first integrated).

Verification caught a security bug in the software, where the code branched on the flag indicating which region of persistent memory was active, and so there was observable timing variation where the circuit leaked more than the spec. Leaking which region of memory was active effectively leaked the parity of the total number of `hash` operations that the HSM had processed, which the specification does not expose. We fixed this by writing more careful C code that GCC compiled without branches so that there was no leakage.

7.1.3 TOTP token

Specification. Figure 14 outlines the specification for the TOTP token. It includes a specification of the TOTP algorithm (not shown) that follows RFC 6238 [54], which relies on HOTP [53], HMAC [46] and SHA1 [57]. The spec doesn't support reading back the secret after it has been set. It allows computing TOTP values given a timestamp supplied by the host machine, but it doesn't allow rewinding the timestamp. It supports an `audit` function to get the last timestamp value, to be able to identify if the HSM was ever abused to compute future TOTP values.

Implementation. The hardware is similar to that of the password-hashing HSM, except this token uses a hardware SHA1 cryptographic accelerator. Its software includes a driver for the SHA1 peripheral that implements message padding, along with a software implementation of HMAC and the TOTP algorithm. Part of the TOTP algorithm is implemented in assembly, carefully written to prevent timing side channels. In one situation, we had to modify C code to be more amenable to symbolic execution, avoiding symbolic memory addresses in favor of fixed addresses and bit-twiddling tricks, as shown in Figure 15. The software for the TOTP token comprises 300 lines of C code and 60 lines of assembly.

Verification and bugs caught. The TOTP token uses a strategy matching the password hasher for achieving atomic state updates. Verifying functional equivalence caught a crash-safety bug where a struct field was missing a `volatile` qualifier and the compiler re-ordered a commit point (toggling the flag) before a write that should happen first (updating state in the inactive region of memory).

The emulator for the TOTP token follows the same basic construction as the others. One interesting detail: the `get_totp` implementation branches based on whether the timestamp is less than the last seen timestamp value; because the timestamp is a 64-bit value and PicoRV32 uses a 32-bit architecture, this turns into a number of comparisons/branches. The emulator, in order to make sure its behavior matches the real circuit's timing behavior, calls the `audit` function to retrieve the real last timestamp value and inject it in place of the

```
var secret = 0, last_timestamp = 0

def set_secret(new_secret):
  secret = new_secret

def get_totp(timestamp):
  if timestamp < last_timestamp:
    return 'Cannot rewind timestamp'
  last_timestamp = timestamp
  return totp(secret, timestamp)

def audit():
  return last_timestamp
```

Figure 14: The functional specification for the TOTP token.

```
/* old implementation:
uint32_t s = (buf[offset]   & 0x7f) << 24
           | (buf[offset+1] & 0xff) << 16
           | (buf[offset+2] & 0xff) <<  8
           | (buf[offset+3] & 0xff);
*/
uint32_t s = 0;
for (int i = 0; i < 0x10; i++) {
    uint32_t match = ((i != offset) - 1);
    s += ((buf[i]   & 0x7f) & match) << 24;
    s += ((buf[i+1] & 0xff) & match) << 16;
    s += ((buf[i+2] & 0xff) & match) <<  8;
    s += ((buf[i+3] & 0xff) & match);
}
```

Figure 15: Rewriting TOTP dynamic truncation to avoid symbolic memory addresses.

HSM	Spec		Driver	HW	SW	Proof
	core	total				
PIN backup	32	60	110	2670	190	470
PW hasher	5	150	90	3020	240	650
TOTP	10	180	80	2950	360	830

Figure 16: Lines of code for case studies. Lines of code for the spec are broken down into "core" and "total", where core is the main HSM functionality and doesn't include boilerplate or definitions of functions like SHA1, HMAC, and TOTP.

dummy data in the circuit copy, so that the timing behavior matches the real circuit. Also, the commit point for the TOTP operation is right after saving the new timestamp value, before the actual call to the TOTP function, so the emulator calls the functional specification's `get_totp` operation at the commit point in order to satisfy the recovery condition, stashes the output in auxiliary state, and when the circuit copy gets to the point where it's returning from its call to the TOTP function (computing with dummy data), injects the cached return value in place of the dummy value in the circuit.

The physical equivalence proof caught an issue with the TOTP implementation: it was using the C modulus (`%`) operator to compute the final mod 10^6 operation, but this operation had variable latency dependent on its input, which leaks information that is not available in the functional specification. The spec doesn't reveal the output of the HMAC or dynamic truncation, only the final 6-digit code. The fix was to implement this functionality in constant time, which we did in assembly code using `sltu` and bitwise/arithmetic instructions.

7.1.4 Summary

IPR and Knox can be applied to simple HSM hardware and software. A design goal of the Knox HSMs, the implementations are minimal, using simple hardware throughout the SoC (e.g., a small RISC-V processor, simpler than the ARM Cortex-M found in many security tokens). Still, the Knox HSMs have implementation features found in real-world HSM hardware (e.g., microprocessor, I/O peripheral, persistent memory, cryptographic accelerator) and software (e.g., peripheral drivers, cryptography, crash safety), and Knox verification covers all of these.

Knox specs are succinct and proofs are manageable. Figure 16 shows lines of code in the spec and driver, implementation (hardware and software), and proof required for verifying each HSM. We break down spec lines of code into "core" and "total", where core doesn't include boilerplate or definitions of functions like SHA1, HMAC, and TOTP. Knox specifications are as short as their pseudocode: for example, aside from the definition of the TOTP algorithm as specified in RFC 6238, the core of the TOTP token specification in Knox is only 10 lines of code, as shown in Figure 17.

```
(struct state (secret last-ts))

(define s0 (state (bv 0 160) (bv 0 64)))

(define ((set-secret secret) s)
  (result #t (state secret (state-last-ts s))))

(define ((get-otp ts) s)
  (if (bvult ts (state-last-ts s)) (result (bv 0 32) s)
      (result (totp (state-secret s) ts)
              (state (state-secret s) ts))))

(define ((audit) s)
  (result (state-last-ts s) s))
```

Figure 17: The core of the Knox specification for the TOTP token. The definition of `totp`, not show here, is a pure function that follows the spec in RFC 6238.

HSM	FE-N	FE-N+C	FE	FE+C	PE
PIN backup	1	10	**209**	962	**8**
PW hasher	1	6	**74**	238	**4**
TOTP	3	8	**44**	141	**8**

Figure 18: Time taken (in minutes) for verification by Knox. FE is functional equivalence; the -N variation disables nondeterminism in the driver; +C adds verification of crash safety. PE is physical equivalence. The two bolded columns, FE and PE, together imply IPR.

Knox catches bugs throughout hardware/software. Verification caught bugs across hardware (e.g., SHA256 peripheral initialization, in the password hasher) and software (e.g., compiler re-ordering a commit point, in the TOTP token), including timing side channels (e.g., variable-time modulus, in the TOTP token) and subtle bugs involving hardware, software, timing, and persistence (e.g., commit point dependent on the PIN guess being correct, in the PIN-protected backup).

7.2 Performance

Verification performance. Figure 18 shows Knox's verification performance, evaluated on a 2014-era Intel i7-5930K. The implementation is currently single-threaded. Most of the time in functional equivalence proofs is due to nondeterminism (yield and merge) or verifying crash safety. The relatively low performance of verifying PIN-protected backup is due to performing case analysis on the slot number, which causes many paths to be explored independently.

When developing functional equivalence proofs, we usually begin by disabling driver nondeterminism and verification of crash safety. This significantly reduces verification time, and the tighter feedback loop speeds up the initial proof development process. After verification completes successfully in this simplified setting, we add back complexity and fix up the implementation and proof as needed.

Implementation performance. The case studies showed that hardware or software may need to be modified to satisfy

Metric	Baseline	Verified	
FPGA LUTs	3966	3962	(−0%)
Max clock freq	20.01 MHz	20.53 MHz	(+3%)
Code size	2412 B	2592 B	(+7%)
TOTP op latency	0.73 ms	0.83 ms	(+14%)

Figure 19: Overhead of modifications to TOTP token.

the strict definition of IPR and simplify verification. The TOTP token required the most modifications among the case studies. Figure 19 shows the impact of these modifications on hardware (FPGA area and maximum clock frequency) and software (code size and performance). Code performance was measured at a clock of 12 MHz and baud rate of 2M for the `totp` operation. Most of the slowdown results from the modification to the dynamic truncation code (Figure 15).

The verified TOTP token can perform TOTP operations with a latency of 0.83 ms, which is fast enough for interactive use [50]. The other HSMs have similar per-operation latency.

8 Discussion

This section elaborates on some of the design decisions made in IPR and Knox and discusses their implications.

8.1 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, information-preserving refinement captures an information-theoretic notion of information preservation, rather than a computational one.

The Knox framework does not fully formalize or mechanically verify emulator efficiency. Instead, the proofs rely on a manual audit of the emulator code. The emulators we construct are simple, so the efficiency property is easy to check. In fact, the Knox 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.

8.2 Randomness

Functional specifications in IPR are deterministic, so IPR cannot be used to verify HSMs that use true random number generators (TRNGs). As an alternative, 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.

8.3 Allowed leakage

IPR enforces that the implementation leaks no more than the specification; sometimes, it is desirable to allow the implementation to leak some non-sensitive information, e.g., the current `bad_guesses` count in the PIN-protected backup. This fits in to IPR: the leakage can be expressed as a spec-level `leak` operation. Knox supports leakage specifications using this strategy, and it allows the user to skip proving functional equivalence for `leak` operations (a well-behaved host does not need to invoke this operation; it is only relevant for modeling leakage as part of physical equivalence).

8.4 Monolithic end-to-end verification

Knox performs monolithic end-to-end verification, which has some benefits over modular verification. There is no need to define intermediate specifications and prove that layers satisfy these specs; there is no distinction between hardware and software, or a notion of, e.g., an instruction set architecture. Knox simply reasons about the cycle-accurate behavior of the entire circuit. If the circuit happens to contain a CPU that runs some software, the software is "inlined" into hardware (the initial contents of a ROM, for example).

Knox uses symbolic execution, and due to performing symbolic execution end-to-end across software and hardware, symbolic execution can be kept as concrete as possible, which improves performance. For example, Knox doesn't attempt to prove a CPU correct (that it executes *any* program correctly); this would require reasoning about symbolic instructions/programs. Instead, a proof of a Knox HSM only shows (indirectly) that the CPU executes the HSM's particular software correctly, which is an easier task.

Lack of modularity could be a challenge when scaling up Knox to more sophisticated HSM implementations, because end-to-end symbolic execution across hardware and software will perform poorly as hardware gets more sophisticated, and the proof developer might have trouble with non-modular reasoning as complexity increases. But modular reasoning about security properties is also challenging: e.g., proving software correct with respect to an ISA specification is inadequate for proving absence of timing side channels.

A potential approach to this problem could involve structuring the HSM implementation to delay responses until a worst-case execution time bound, and then specializing the verification framework to reason about HSMs following such a design. With such a structure, precise Knox-style reasoning about software executing on hardware may not be necessary, making it possible to separately reason about correctness (following standard approaches) and then reason about worst-case execution time bounds of software executing on hardware to show a top-level property like IPR.

9 Related work

Noninterference. Noninterference [39] captures confidentiality properties in systems where high-sensitivity inputs should not affect low-sensitivity outputs, which are separate from high-sensitivity outputs. seL4 [55], mCertiKOS [30], Komodo [36], and Nickel [64] verify noninterference properties such as process isolation. HACL* [62, 74], Vale [25, 38], EverCrypt [63], and Jasmin [14] phrase freedom from timing side channels in crypto code as noninterference by defining a leakage trace (the low-sensitivity output) that captures adversary observations, such as every program counter value, and showing that two executions that have matching public inputs but differing secrets (high-sensitivity inputs) produce identical leakage traces.

The Knox setting does not have separate low/high-sensitivity outputs: there is just one output, the logic levels on the output wires at every cycle, and this is what the adversary observes. Noninterference does not hold in the Knox setting: the output can and will be secret-dependent. Instead, IPR says that the output does not leak more information than the spec, which is not a noninterference property.

Declassification. Noninterference with declassification [56] 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 [42] uses this style of security definition; the proofs cover only software, not hardware, and do not rule out timing side channels.

The Knox setting does not separate low and high-sensitivity inputs. There is just one input, the logic levels on the input wires at every cycle. 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.

Ironclad contains a PassHash app similar to the Knox password hasher. PassHash generates a secret internally, from a computer's TPM, and it services network requests: given a password, it returns a hash of the password salted with the secret. The secret is a high-sensitivity input (from the TPM) and the password is low-sensitivity input (from the network); the security definition is phrased as noninterference with declassification, allowing the final output to the network to depend on the secret in a controlled way. In the Knox HSM, the secret is not generated internally but received from the host, and both secrets and passwords are received over the same input wires (there are no separate public and secret inputs). A declassification-style definition does not apply. IPR says that the implementation's behavior can't leak more information than the specification, so for example, after a host sets the secret, the HSM can't leak the secret. IPR also gives the same property with passwords: the HSM can't leak passwords that were input earlier. In contrast, the noninterference property for PassHash doesn't prevent the implementation from leaking passwords, because they are low-sensitivity inputs.

Hardware/software verification. A long line of work performs end-to-end verification of functional correctness properties for hardware/software systems, with an emphasis on modular verification [13, 23, 35, 48]. Proving functional correctness does not rule out timing side channels, while addressing side channels is a central goal of IPR and Knox.

Knox uses Notary's toolchain to convert C/Verilog to Rosette models, and Knox uses Notary's idea of reset-based design for simplifying verification [16, 52]. Knox solves a different problem than Notary. IPR is a new security definition for HSMs that captures the notion that a hardware/software implementation satisfies a functional specification and leaks nothing more, and Knox is a framework for proving this property, including support for writing specifications, encoding drivers and emulators, and proving correctness and security. Notary's focus is a hardware/software architecture for better isolation between multiple mutually-distrustful agents running on the same device, and Notary only verifies a simple (but key) property of an embedded system and its boot code, that all internal state is cleared after reset.

Simulation-based definitions of security. IPR is inspired by simulation-based definitions of security for multiparty computation (MPC) and universal composability (UC) [28, 40, 41]. The Knox emulator is similar to the MPC simulator, which formalizes the notion of zero knowledge in an MPC protocol. Knox uses this concept to define non-leakage for a hardware/software system.

Verified cryptography. Some tools [18, 19, 24, 61] verify cryptographic properties of functional specifications and protocols. These are complementary to Knox, as illustrated by work that formally analyzes HSM interfaces [26, 33].

Other works prove functional correctness of crypto implementations [17, 22, 27, 29, 34]. Some of these provide side-channel resistance with cryptographic constant-time code, which can be compiled to machine code while preserving constant-time [20], but the security property does not go down to the hardware / wire I/O level.

Verifying efficiency. Cryptographic proofs generally reason about efficient adversaries, and some frameworks for verified cryptography support proving polynomial bounds on programs' running time [18, 61]. Knox could follow their approach, adding a cost semantics for the emulator language, to formally reason about emulator efficiency.

Secure compilation. In Knox, the circuit is not derivable 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 [11] preserve and reflect observational equivalence from the source to the target language. Some security properties can be stated as program equivalences [60], but IPR's non-leakage property is not captured by this type of definition. In fact, some Knox specifications such as the password hasher 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 [59] 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 Knox specifications such as the password hasher have no instances that are trace-equivalent but not equal, so trace-equivalence preservation at the circuit level would be vacuous.

10 Conclusion

Information-preserving refinement (IPR) is a new security definition that captures the idea that a circuit-level implementation should implement its logical-level specification and leak nothing more. Knox demonstrates that IPR is useful in practice for ruling out bugs in an HSM's hardware and software. We believe that IPR is applicable beyond HSMs and hope that it can serve as a foundation of future security definitions.

Acknowledgments

We would like to thank Henry Corrigan-Gibbs and Joseph Tassarotti for insightful discussions that helped us develop the IPR definition. We are grateful to Emina Torlak and Xi Wang for their guidance on working with Rosette. This paper has been improved thanks to feedback from many individuals, including Akshay Narayan, Alexandra Henzinger, Ariel Szekely, Derek Leung, Kyle Hogan, Ralf Jung, Robert Morris, Sacha Servan-Schreiber, Stella Lau, Tej Chajed, Thomas Bourgeat, Yun-Sheng Chang, the anonymous reviewers, and our shepherd, George Candea. This research was supported by NSF award CNS-1812522 and by Google.

References

[1] CVE-2004-0320. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0320`, Sept. 2004.

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

[3] CVE-2018-6875. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6875`, Feb. 2018.

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

[5] CVE-2019-18671. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18671`, Nov. 2019.

[6] CVE-2019-18672. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18672`, Nov. 2019.

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

[8] CVE-2021-31616. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31616`, Apr. 2021.

[9] WhatsApp security whitepaper: Security of end-to-end encrypted backups. `https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf`, Sept. 2021.

[10] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. Schoen, and B. 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)*, pages 2473–2487, London, United Kingdom, Nov. 2019.

[11] M. Abadi. *Protection in Programming-Language Translations*, pages 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48749-4.

[12] D. Agrawal, B. Archambeault, J. R. Rao, and P. 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.

[13] E. Alkassar, W. J. Paul, A. Starostin, and A. 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, Aug. 2010.

[14] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 1807–1823, Dallas, TX, Oct.–Nov. 2017.

[15] Apple, Inc. Apple platform security. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf, May 2021.

[16] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 97–113, Huntsville, Ontario, Canada, Oct. 2019.

[17] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-aided cryptography. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, pages 777–795, Virtual conference, May 2021.

[18] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 90–101, Savannah, GA, Jan. 2009.

[19] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual International Cryptology Conference (CRYPTO)*, pages 71–90, Santa Barbara, CA, Aug. 2011.

[20] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. 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.

[21] J.-B. Bédrune and G. Campana. Everybody be cool, this is a robbery! https://donjon.ledger.com/BlackHat2019-presentation/, Aug. 2019.

[22] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Security Symposium*, pages 207–201, Washington, DC, Aug. 2015.

[23] W. R. Bevier, W. A. Hunt Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, Dec. 1989.

[24] B. Blanchet. A computationally sound mechanized prover for security protocols. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 140–154, Oakland, CA, May 2006.

[25] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the 26th USENIX Security Symposium*, pages 917–934, Vancouver, Canada, Aug. 2017.

[26] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 260–269, Chicago, IL, Oct. 2010.

[27] B. Boston, S. Breese, J. Dodds, M. Dodds, B. Huffman, A. Petcher, and A. 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.

[28] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, Las Vegas, NV, Oct. 2001.

[29] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang. Verifying Curve25519 software. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 299–309, Scottsdale, AZ, Nov. 2014.

[30] D. Costanzo, Z. Shao, and R. 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.

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

[32] L. de Moura and N. 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, Mar.–Apr. 2008.

[33] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 331–344, Pittsburgh, PA, June 2008.

[34] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. 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.

[35] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. 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)*, June 2021.

[36] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. 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, Oct. 2017.

[37] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

[38] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. 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.

[39] 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, Apr. 1982.

[40] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, New York, NY, May 1987.

[41] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–304, Providence, RI, May 1985.

[42] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. 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, Oct. 2014.

[43] M. Hemmel, J. Meltzer, T. Pornin, K. Ryan, J. Samuel, D. Wong, R. Wood, and G. 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.

[44] M. Hutter and J.-M. 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, Nov. 2013.

[45] J. Jancar, V. Sedlacek, P. Svenda, and M. 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.

[46] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Network Working Group, Feb. 1997.

[47] I. Krstić. Behind the scenes with iOS security. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf, Aug. 2016.

[48] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. 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.

[49] R. 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, Aug. 2000.

[50] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the AFIPS 1968 Fall Joint Computer Conference*, pages 267–277, San Francisco, CA, Dec. 1968.

[51] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *Proceedings of the 29th USENIX Security Symposium*, pages 2057–2073, Aug. 2020.

[52] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich. rtlv: push-button verification of software on hardware. In *Proceedings of the 5th Workshop on Computer Architecture Research with RISC-V (CARRV)*, Virtual conference, June 2021.

[53] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-based one-time password algorithm. RFC 4226, Network Working Group, Dec. 2005.

[54] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-based one-time password algorithm. RFC 6238, Network Working Group, May 2011.

[55] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. 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.

[56] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–147, Saint-Malo, France, Oct. 1997.

[57] National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards (FIPS) 180-4, U.S. Department of Commerce, Washington, DC, Aug. 2015.

[58] 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.

[59] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF)*, pages 392–404, Santa Barbara, CA, Sept. 2017.

[60] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 51(6), Nov. 2019.

[61] A. Petcher and G. Morrisett. The Foundational Cryptography Framework. In *Proceedings of the 4th International Conference on Principles of Security and Trust*, pages 53–72, Apr. 2015.

[62] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. 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.

[63] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. 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.

[64] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. 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, Oct. 2018.

[65] S. Srinivas, D. Balfanz, E. Tiffany, and A. Czeskis. Universal 2nd Factor (U2F) overview. https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf, Sept. 2016.

[66] J. Strömbergson. sha256. https://github.com/secworks/sha256, 2013.

[67] E. Torlak and R. 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] F. Uekermann. Buggy OTP slot range check. https://github.com/Nitrokey/nitrokey-pro-firmware/issues/4, June 2016.

[69] C. X. Wolf. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys, 2012.

[70] C. X. Wolf. Project IceStorm — Lattice iCE40 FPGAs bitstream documentaion. https://github.com/YosysHQ/icestorm, 2015.

[71] C. X. Wolf. PicoRV32 – a size-optimized RISC-V CPU. https://github.com/YosysHQ/picorv32, 2015.

[72] C. X. Wolf, gatecat, D. Gisselquist, S. Bazanski, M. Milanovic, and E. Hung. nextpnr. https://github.com/YosysHQ/nextpnr, 2018.

[73] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Report 2005/388, Oct. 2005.

[74] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.