Verifying Hardware Security Modules with Information-Preserving Refinement

Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich

Presented by Russell Bentley

Stony Brook

2024

Introduction

- Presenting [1] (2022)
- Hardware Security Modules (HSM)
- Information Preserving Refinement
- Knox Framework
- Case studies

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.

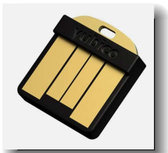
Our approach is to relate the behavior of the HSM implementation at the wire level interface — the ground truth of what the host machine controls and observes at the digital level, which captures timing channels at a cycle-accurate level — to a functional specification of the methods that the HSM exposes. 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

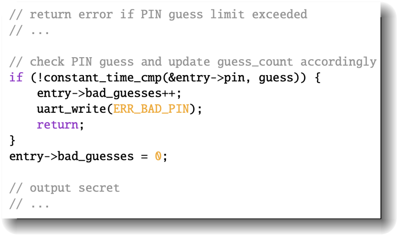
Hardware Security Modules (HSM)

- Dedicated hardware for core security functionality
  - Certificate signing
  - Password hashing
  - Token generation
- High Value Targets
- Hardware + Software
  - May include general purpose core
  - Communicates with host over wire interface



Threat Model

- Remote compromise of host
- *digital* (mis)use of interface
- Ensure no extra information leaks
  - Bug in implementation
  - Timing Attacks $\rightarrow$
  - Crash Safety
- Does not consider physical access
  - Overvoltage, tampering
  - Secure as the specification



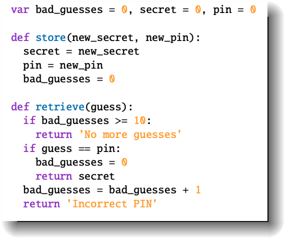
```c
// return error if PIN guess limit exceeded
// ...

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

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

Information Preserving Refinement (IPR)

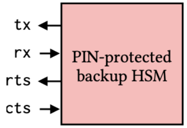
- Start with functional specification
- Device implements this spec
- Wire-level interactions $\Longleftrightarrow$ spec. level operations
- Implementation States $\Longleftrightarrow$ spec. level states
- Based on dual view
  - Real world (uncompromised host)
  - Ideal world (compromised host



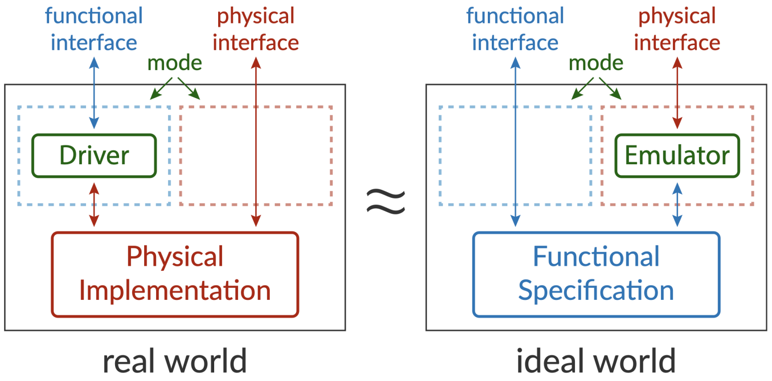
```
var bad_guesses = 0, secret = 0, pin = 0

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'
```



Information Preserving Refinement (IPR)



Proving IPR

- Developer provides R, relates
  - State of spec state machine
  - State of implementation
- Developer also provides
  - Driver
  - Emulator

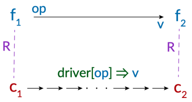


Figure 7: Functional equivalence: for all implementation states c_1 and spec states f_1 that are related by R, and for all spec-level operations op:
(1) the spec-level output v matches the driver output
(2) the final states c_2 and f_2 are related by R

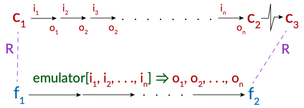


Figure 8: Physical equivalence: for all spec states f_1 and implementation states c_1 that are related by R, and for all wire-level inputs $i_1 \ldots i_n$:
(1) the circuit outputs $o_1 \ldots o_n$ match the emulator outputs
(2) the final states f_2 and c_3 (c_2 after a reset) are related by R

Implementation: Knox

- Knox, monolithic verification
  - Source on Github
  - Hybrid Symbolic Execution [3]
- Targets Rosette [4]
  - Solver-aided programming language
- Generate SMT Constraints
  - Solve with Z3 [2]

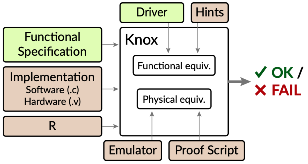


Figure 11: An overview of the Knox workflow. Trusted inputs are shown in green.

Symbolic Execution

- Functional Equivalence - `yeild`
  - How to handle Nondeterminism?
  - Find closure of circuit states
  - Fork on elements of set
  - Exponential growth!
  - *merge* hints shrink space
- Physical Equivalence - input size
  - Arbitrary sequence of wire inputs
  - Intractable to write circuit invariant
  - *Guided symbolic model checking*
- Knox allows variety of hints

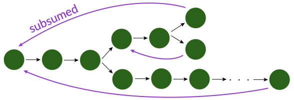
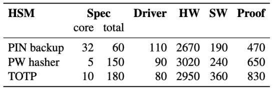


Figure 10: An illustration of *guided symbolic model checking* exploring a state space. Each green circle is a symbolic term representing a set of states. Black arrows show STEP invocations and purple arrows show SUBSUMED invocations.

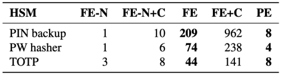
Case Studies

- Three Case Studies
  - PIN-protected backup
  - Password hashing
  - TOTP Token
- Source on Github



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.

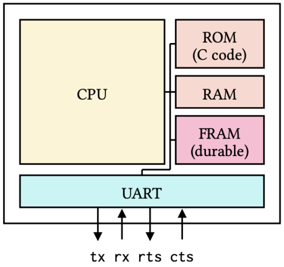


HSM	FE-N	FE-N+C	FE	FE+C	PE
PIN backup	1	10	**209**	962	**8**
PW hasher	1	6	**74**	238	**4**
TOTP	3	8	**44**	141	**8**

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.

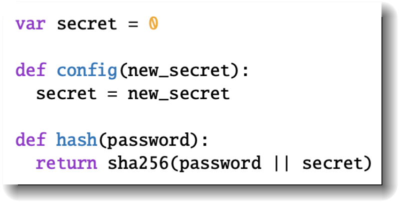
Case Study: PIN-protected backup HSM

- Runs on RISC-V CPU
- Features
  - Multiple Secrets
  - Enforce Attempts Limit
- Emulator Implementation
  - Constructive
  - Inject spec output into circuit
- Bugs Found
  - Timing attack with `strcmp`
  - Guess timing attack



Case Study: Password Hashing HSM

- Includes `sha256`
  - Needs spec
  - Verilog implementation
- Can't retrieve secret
- Crash Safe
  - Multiword writes
  - Caught timing bug
- Caught `mem_addr` bug



```
var secret = 0

def config(new_secret):
  secret = new_secret

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

Case Study: TOTP Token

- Host provides timestamp
- Can't rewind timestamp
- Includes `SHA1`
- Issue with symbolic execution
- Timing attack with %

```
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.

Conclusion

- Today we covered:
  - HSMs
  - Information Preserving Refinement
  - Knox
- Questions?

References I

[1] Anish Athalye, M Frans Kaashoek, and Nickolai Zeldovich. "Verifying hardware security modules with {Information-Preserving} refinement". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 503–519.

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

[3] Emina Torlak and Rastislav Bodik. "A lightweight symbolic virtual machine for solver-aided host languages". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 530–541.

[4] Emina Torlak and Rastislav Bodik. "Growing solver-aided languages with rosette". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 135–152.