Verifying Hardware Security Modules With True Random Number Generators

by

Katherine Zhao

S.B. in Computer Science and Engineering and Mathematics, Massachusetts Institute of Technology (2024)

Submitted to the the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Katherine Zhao Department of Electrical Engineering and Computer Science May 10, 2024
Certified by:	Anish Athalye Doctoral Candidate, Thesis Supervisor
Certified by:	Nickolai Zeldovich Professor, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Verifying Hardware Security Modules With True Random Number Generators

by

Katherine Zhao

Submitted to the the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Hardware security modules (HSMs) are powerful tools in building secure computer systems, allowing developers to factor out security-critical code to separate devices. Because HSMs usually work with sensitive data, it is crucial that we are able to verify that they are secure. Many HSMs today also include true random number generators (TRNGs) as part of their architecture to seed cryptographic functions for generating keys, creating nonces, padding, and more. This thesis presents a definition of *Information-Preserving Refinement with Randomness* (IPRR) that captures the idea that a HSM with a TRNG is correct and is secure from timing side channel attacks. We additionally construct a strategy to prove IPRR, and develop Karatroc, a tool for verifying that a HSM satisfies IPRR. Through the creation and evaluation of Karatroc, we demonstrate the ability to verify HSMs with TRNGs without incurring significant added cost in performance and proof length as compared to existing proof methods.

Thesis supervisor: Anish Athalye
Title: Doctoral Candidate

Thesis supervisor: Nickolai Zeldovich
Title: Professor

Acknowledgments

Thank you to Anish Athalye, for being a great mentor and an inspiring role model. Your thoughtful advice, expertise, and prior work were fundamental to the creation of this thesis. You were always there to help when I needed it, and I am so grateful for that.

Thank you to Professor Nickolai Zeldovich, for your insightful guidance and feedback throughout my journey to writing this thesis.

Thank you to my friends, for providing me with invaluable memories that I'll never forget.

Finally, thank you to my parents, for your unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

When building secure computer systems, a useful method is to factor out key security-critical functionality onto hardware security modules (HSMs). These HSMs are separate devices with special purpose software and hardware that perform high security tasks such as storing keys, signing certificates, and encrypting data. For instance, Google's Meet devices contain HSMs that manage firmware rollback and protect user keys [1], and Apple stores users' iCloud keys on HSMs in its data centers [2]. Because these devices often contain sensitive information, it is crucial that they remain secure even if the host computer they are connected to is compromised.

Ensuring that a HSM is correct, secure, and free of timing side channel vulnerabilities is difficult, but a promising approach is through formal verification. With formal verification, we can prove that a HSM satisfies a given specification and that it leaks no additional information. Prior work has formalized this into a definition called the Information-Preserving Refinement (IPR) and developed the Knox framework [3] to verify that a HSM satisfies IPR. Even so, current tools are limited in the types of HSMs that they can verify, and they cannot support HSMs that use true random number generators (TRNGs).

However, many real world HSMs use TRNGs for a variety of security applications and cryptographic protocols such as key generation, initialization vectors, and secure communication [4]. Many popular TRNGs on the market today have built-in TRNGs, such as the Thales Luna Network HSM [5] and the Yubico YubiHSM [6]. Because of the prevalence of TRNGs in HSMs, it is crucial that we are able to verify security for these HSMs.

Acknowledging these factors, this thesis presents a definition of *Information-Preserving Refinement with Randomness* (IPRR) that captures the idea that a HSM is correct and secure from timing side channel attacks. We furthermore develop a framework, Karatroc, for verifying that a HSM satisfies IPRR.

More specifically, IPRR defines a relationship between the specification and the implementation. The specification is a description of the operations that the HSM should perform, written in a high-level programming language. For instance, the specification of a password-hasher HSM is shown in Figure 1.1, containing a `config` function to set up the secret key with a new random value by sampling random bits through the `get_rand_bit()` function. It also contains a `hash` function that computes the hash of the password concatenated with the secret.

```
var secret = 0
var SECRET_LENGTH = 160

def config():
    secret = 0
    for i in range(SECRET_LENGTH):
        secret = (secret << 1) + get_rand_bit()

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

Figure 1.1: The specification of the password-hashing HSM.

The implementation of the HSM refers to the physical circuit, which gets random values through a physical TRNG.

If the implementation and specification satisfy IPRR, then it ensures two key aspects.

Firstly, the implementation outputs correct values through a defined sequence of steps. This also accounts for random values, so for instance the password-hasher implementation would need to sample random values from the TRNG in order to correctly reflect the `secret` variable. Secondly, IPRR asserts that the implementation cannot leak any information beyond what is revealed by the specification. The specification for the password-hasher has no way of recovering the secret after it is set. Therefore, the implementation has no way of leaking the secret either, in both its outputs and its timing behavior. This means that IPRR would not only rule out circuits that output the secret during execution, but also circuits that take a longer or shorter time to execute depending on the value of the secret, preventing the adversary from learning any additional information.

To formalize the definition of IPRR, we use a real/ideal world model, where the real world is based on the implementation and the ideal world is based on the specification and is correct and secure by construction. IPRR asserts that for all values that the random numbers can take, the real and ideal worlds are indistinguishable to the host computer.

1.2 Threat Model

This thesis focuses on powerful adversaries capable of remotely compromising a host machine and gaining access to the wire-level interface of the HSM, allowing them to observe signals on output wires and write to input wires. These adversaries can exploit digital-level timing side channels. For example, if a circuit takes a different number of cycles to compute an operation depending on a stored secret, the adversary can observe this behavior and potentially deduce the value of the secret.

However, we do not consider adversaries that can access arbitrary side channels such as temperature, power, and electromagnetic radiation, as we are primarily concerned with the scenario where the attacker is remote. Furthermore, the attacker cannot observe the internal behavior of the HSM, and therefore cannot see wire-level behavior between the HSM and

the TRNG.

1.3 Goals and Challenges

The security goal of Karatroc is to ensure that a HSM that uses a TRNG correctly implements its specification and leaks no additional information. We formalize this idea with a definition called Information-Preserving Refinement with Randomness (IPRR). Subsequently, we develop a tool to verify that a HSM with a TRNG satisfies IPRR.

We aim for our definition and implementation to support realistic TRNGs, similar to those currently in use. We want to ensure that our model of the TRNG accurately reflects the interfaces and the operation of real TRNGs; one, for usability of our tool in the real world, and two, so that we can make sure the adversary cannot attack the HSM by using properties of the TRNG that were not captured in the model.

A key challenge of this thesis is to represent randomness in a way that allows us to capture correctness and security of a HSM. Randomness is traditionally modeled through nondeterminism, where one state can transition to multiple states. This is usually sufficient to prove correctness in settings where randomness in the specification means the variable in the implementation can take any value, by ensuring that every end state satisfies the specification. However, in our setting, simply using nondeterminism is not enough: we need a way to make sure that the randomness used in the implementation reflects the randomness used in the specification, and that no information is leaked through randomness. For instance, consider the password-hasher example. The `secret` is specified to be random, but it is not enough to simply accept implementations that generate any value as the secret. Doing so could lead to the acceptance of insecure schemes, such as one that always outputs 0 as the secret. As another example, consider a HSM that implements a function to output a new random word on every invocation. We do not want to accept implementations where the outputted word depends on secret state in the HSM, or implementations that reuse the same random word

every time. Therefore, we need to make sure that the implementation and the specification use randomness in the same way: if we specify that a variable is random, then that variable must correspond to some values sampled from the TRNG. As a consequence, we also need our model to reflect equivalence in randomness between the specification and implementation. Randomness from the specification's `get_rand_bit` function should be represented as equivalent to the implementation's new TRNG output, but different from all other sources.

1.4 Thesis Contributions

This thesis makes the following contributions:

1. The definition of *Information-Preserving Refinement with Randomness (IPRR)*, which captures the idea that a HSM that uses a TRNG correctly implements its specification and leaks no additional information.

2. A proof strategy to verify that a given HSM satisfies IPRR.

3. Karatroc, a tool that allows developers to verify that their HSMs are secure using the aforementioned proof strategy.

4. Verified examples of HSMs using Karatroc. This includes a toy example of a random byte generator and a more complex example of a password hashing HSM that uses a TRNG. We then use these HSMs to evaluate Karatroc on proof complexity and performance.

The source code for Karatroc is available at https://github.com/katherinezhao02/karatroc, and the verified examples are available at https://github.com/katherinezhao02/karatroc-hsm.

1.5 Thesis Outline

The rest of this thesis is structured according to the following outline. Section 2 provides background information on HSMs and TRNGs. Chapter 3 introduces the definition of IPRR, Chapter 4 describes the strategy for proving IPRR, and Chapter 5 describes the implementation of Karatroc. Chapter 6 presents two case studies for verifying HSMs using Karatroc, and Chapter 7 evaluates Karatroc using metrics obtained from the two case studies. Chapter 8 discusses the rationale behind, implications, and limitations of some design decisions made in IPRR and Karatroc. Chapter 9 presents areas for future work, as well as preliminary progress made for verifying security under a different TRNG model. Finally, Chapter 10 details related works, and Chapter 11 concludes the paper.

Chapter 2

Background

2.1 Hardware Security Modules

A hardware security module (HSM) is a physical device that is specialized for security-critical operations. HSMs allow developers to factor out sensitive functions, such as key generation and authentication, to a machine separate from the main codebase. This allows for stronger confidence in its security properties, as it is much simpler to reason about a smaller system, and there are a lot fewer factors that can interfere with its operation.

Some existing examples of HSMs include the Yubico YubiHSM [6] and the Thales Luna Network HSM [5]. These HSMs are both able to perform several cryptographic algorithms such as key generation, encryption, and authentication. Both also include built-in TRNGs that are used to seed functions. Furthermore, major software companies use HSMs to protect sensitive data. For instance, Google Meet hardware devices come with HSMs that manage firmware rollback and protect user keys [1], and Apple iCloud uses HSMs in its datacenters to store user information [2].

Although HSMs can provide strong security guarantees, they are still susceptible to bugs and side channel attacks. Especially considering the fact that HSMs usually work with security-critical data, it is important that we are able to verify that HSMs are secure.

2.2 True Random Number Generators

A true random number generator (TRNG) is a device that generates random numbers based on a physical process that produces entropy. These sources include radioactive decay, Brownian motion, multi-phase oscillators, flash memory cells, and others [7], [8].

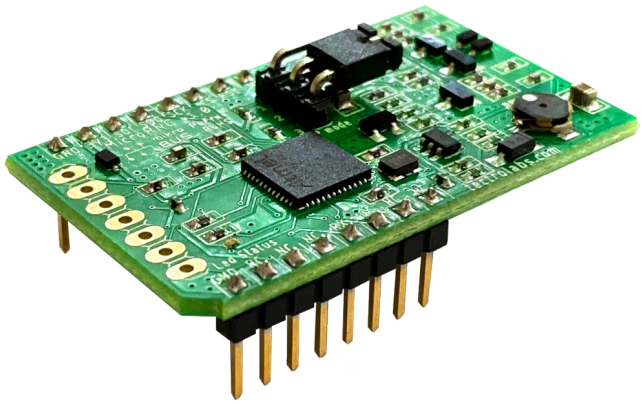

Figure 2.1: The TectroLabs MicroRNG [9]

TRNGs come in a variety of forms and interfaces, but in this thesis we are primarily concerned with those can interact directly with a circuit. Commercial TRNGs of such kind provide two main types of interfaces. The first type is one that outputs random bits at a fixed rate that is often synchronized with a clock, such as the TRNG-P200 [8] and the RPG100 [10]. Figure 2.2 shows the Verilog module interface of a simplified version of these TRNGs.

```
module trng (
    input clk,
    input enable,
    output trng_bit);
```

Figure 2.2: A simplified Verilog module interface of a TRNG that outputs random bits at a fixed rate. Every cycle of `clk`, if `enable` is true, the TRNG outputs a new random bit to `trng_bit`. Output is undefined when `enable` is false.

The other type is one that outputs a longer random word, but having some delay before

the random word is ready, such as the TectroLabs MicroRNG [9], which uses the UART protocol, and the neoTRNG [11], which has a flag that goes high when the random word is ready. An example Verilog interface for this type of TRNG is displayed in Figure 2.3.

```
module trng (
    input clk,
    input reset,
    input enable,
    output data,
    output valid);
```

Figure 2.3: A Verilog module interface of a TRNG that outputs a random word after some delay, modeled after the neoTRNG [11]. When a random word is ready and `enable` is true, the `valid` wire will go high and the user can sample the random bit from the `data` output. Output is undefined when `enable` is false.

In this thesis, we will mainly focus on the first TRNG interface, but we will discuss more about how to perform verification with TRNGs of the other interface in the Chapter 9.

Chapter 3

Information-Preserving Refinement with Randomness (IPRR)

In this chapter we provide the definition of *Information-Preserving Refinement with Randomness* (IPRR). IPRR relates a HSM implementation with a specification that describes the functions that the HSM should perform. For instance, the specification for the password-hasher is shown in Figure 1.1, detailing its two operations. An example architectural model of an implementation for the password-hasher is shown in shown in Figure 3.1, consisting of C code that is run on a CPU.

IPRR asserts that the implementation correctly implements the functional specification and that it leaks no additional information beyond the specification.

We start by defining the TRNG model, as well as the specification and implementation structure, before then providing the definition of IPRR.

3.1 TRNG Model

We model the physical TRNG based on real world TRNGs that output random bits synchronized with the clock, such as the TRNG-P200 [8] and the RPG100 [10].

To represent to nondeterministic behavior of the TRNG, we model the TRNG as being

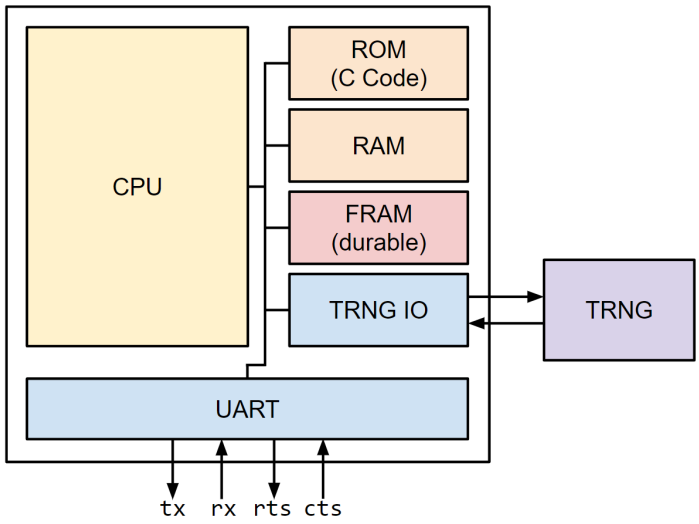

Figure 3.1: An example architecture model of a HSM implementation. This HSM uses the UART protocol to communicate with the host through the `tx`, `rx`, `rts`, and `cts` wires. It stores software written in C in ROM and uses ferroelectric RAM (FRAM) for persistent storage. It also contains a TRNG IO module to interact with the TRNG.

parameterized by an internal state of an infinite stream of bits that represents all future outputs of the TRNG. This representation simplifies the behavior of the TRNG in terms of concrete bits that we can then easily reason with and also allows us to define what it means for two TRNGs be equivalent. For a single stream of bits, the behavior of the TRNG is deterministic, but randomness is captured by allowing all choices of bit streams.

The interface of the TRNG consists of two wires: the output `trng_bit`, which outputs the next random bit, and the input `trng_next`, which requests a new `trng_bit` at the next cycle when set to 1. The `trng_next` wire is similar to the `enable` wire from Figure 2.2, but with some notable differences, discussed in Section 8.1. As in the real world examples, our TRNG model is able to output a new random bit every cycle if it is requested.

The description of the TRNG model is displayed in Figure 3.2.

```
module trng (
    input trng_next,
    output trng_bit);
trng_bit = trng_next ? Stream.next() : trng_bit
end module
```

Figure 3.2: The description of the TRNG model, which outputs a new random bit from the stream when `trng_next` is high.

3.2 Implementation

The implementation in IPRR reflects the real world circuit that we want to verify security for, with an example architectural diagram shown in Figure 3.1. Its interface consists of input and output wires that the host can interact with, along with the wires that it uses to interact with the TRNG. As an example, the interface of a password-hashing HSM is displayed in Figure 3.3. The password-hasher is designed to be used with the UART protocol to send and receive bits with the host machine through the `rx`, `cts`, `tx`, and `rts`. To sample random bits from the TRNG, it uses the `trng_next` and `trng_bit` wires.

```
module pwhash (
    input clk,
    input resetn,
    input rx,
    input cts,
    output tx,
    output rts.
    input trng_bit,
    output trng_next);
```

Figure 3.3: The interface of the physical implementation for the password-hashing HSM.

3.3 Specification

The functional specification describes the behavior of the functions provided by the HSMs. These specifications can be written in a high-level programming language, making them

easier to reason about than the circuit implementation.

The specification can represent random values by using the `get_rand_bit()` function, as shown in the password-hasher's specification in Figure 1.1. Internally, the specification's state contains an infinite stream of bits, like the TRNG state in the real world, and the `get_rand_bit()` function reads the next bit from the stream. We will refer to the specification's infinite stream of bits as the specification's TRNG.

In addition to the functional specification, the specification in IPRR also includes the driver. The driver is an object that describes how the host would invoke each spec-level operation by interacting with the HSM over its wire-level interface. For instance, for the `hash(password)` function, the driver code would involve setting the `rx` wire to the bits of `password` one by one and sending them over to the HSM, waiting for the circuit to run, and then reading the bits of the output from the `tx` wire. One point to note is that the driver cannot view or modify the values of the `trng_bit` or `trng_next` wires, as these wires communicate with the TRNG rather than the host.

3.4 IPRR Definition

Using the previously mentioned components, we are able to construct a definition for IPRR, which is a zero-knowledge style definition that captures the idea that a HSM with a TRNG correctly implements its specification and leaks no additional information. At a high level, IPRR says that the real world, based on the HSM implementation, and ideal world, based on the specification and secure by construction, are indistinguishable given the same TRNG stream. The definition is depicted in Figure 3.4.

The real world in IPRR reflects how the HSM interacts with the host and the TRNG in reality. The host can interact with the HSM by either sending arbitrary wire-level I/O through the physical interface or by sending spec-level operations through the functional interface. The wire-level I/O can directly interact with the HSM, but the spec-level opera-

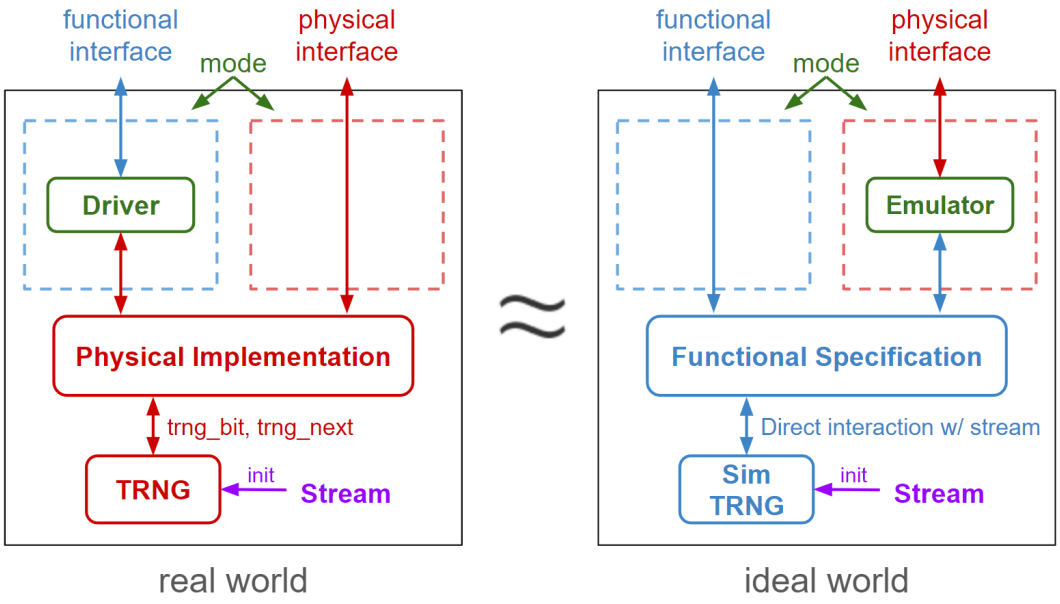

Figure 3.4: Information-Preserving Refinement with Randomness (IPRR), which asserts that the real and ideal worlds are indistinguishable given access to randomness.

tions must first be read by a driver and translated to wire-level operations. To interact with the TRNG, the implementation can query the TRNG through the `trng_next` wire and get random bits through the `trng_bit` wire.

The ideal world is based on the specification and is correct and secure by construction. Like in the real world, the host can interact with the ideal world through the functional and physical interfaces. However, in the ideal world, the host can directly interact with the specification through the functional interface, but wire-level I/O must first be translated to spec-level operations. This is done through the emulator which mimics wire-level behavior given only query access to the spec. Unlike the driver, the emulator is not part of the specification and instead is a proof artifact. The existence of any emulator such that the real and ideal worlds are indistinguishable implies IPRR.

More specifically, the emulator has five main operations, detailed in Figure 3.5. The first three are exposed to the host and corresponds the wire-level interactions of setting an input, getting an output, and running for one cycle. The others are for initializing emulator state

and shutting down the emulator, cleaning up any unresolved operations before switching back to functional view. One point to note is that the emulator does not have direct access to the specification's simulated TRNG and must query the specification through the API if it needs the values. This ensures that random state that is part of a secret does not get leaked. However, the emulator does have access to a `discard` function, which when called, discards the next bit in the specification's TRNG stream (security is maintained here because the function returns nothing so the emulator does not learn anything). This function is necessary in order to maintain TRNG stream equivalence between the real and ideal worlds. We will discuss the use case of `discard` further in Sections 4.4 and 6.1.4.

```
var emulator_state
# the format and typing of emulator state is specified by the
   developer

def with_input(i) # sets the input for the cycle, where the input is
   formatted in terms of the input wires to the implementation

def get_output() # gets the values of the output wires

def step() # simulates running the circuit for one cycle

def init() # initializes emulator state

def shutdown() # run when the circuit is reset, usually calls discard
    here
```

Figure 3.5: The functions that the emulator implements.

In IPRR, the host can switch between the functional and physical views at any time. Switching from functional to physical models situations where the host gets compromised, so rather than sending inputs based on the specification it can start to send arbitrary I/O values to the HSM in an attempt to get it to leak information. On the other hand, switching from physical to functional models recovery of the host machine. When switching from physical to functional, the driver is reinitialized, and when switching from functional to physical, the emulator is reinitialized. This ensures that our emulator cannot "remember"

state outside the specification and is thus secure.

With these factors in mind, we can give our precise definition of IPRR. IPRR states that there exists an emulator such that for all TRNG streams where the real and ideal worlds are initialized with the same TRNG stream, the real and ideal worlds are indistinguishable to the host.

For instance, in the functional view of the password-hasher, the host sends spec-level commands such as `config` and `hash`. In the ideal world, this would interact directly with the specification, but in the real world, we use a driver to translate these commands into inputs for the implementation. IPRR states that the output from each of these spec-level operations is the same for both the real and ideal worlds, capturing the idea that the implementation correctly follows the specification.

In the physical view, the host can modify the signals of `rx`, `cts`, etc, and read the signals of `tx` and `rts`. This can be done directly in the real world, but in the ideal world we need to translate these signals into spec-level queries through the emulator. Because the specification has no way of recovering the secret after it is set, the emulator also has no way of recovering the secret. Therefore, an implementation that satisfies IPRR has no way of leaking the secret, both directly through its outputs or indirectly through timing, or else its behavior would differ from the emulator.

It is important to note the order of quantifiers in IPRR: if we were to say "for all TRNG streams where the real and ideal worlds are initialized with the same TRNG stream, there exists an emulator such that the real and ideal worlds are indistinguishable to the host", then IPRR would not capture security. Under this definition, implementations that leak a secret that depends on randomness could pass because the emulator can depend on the TRNG stream. For instance, the emulator in the password-hasher would know the secret because the secret is simply composed of bits from the TRNG, and insecure implementations that leak the secret would still be considered "secure".

Chapter 4

Proving IPRR

This section describes the strategy we use to prove that a HSM satisfies IPRR. Given a circuit description of a HSM and the specification, we start by modeling the circuit, the specification, and the TRNG as state machines. We then connect the circuit and specification using a refinement relation, which enables the verification of functional equivalence and physical equivalence (the idea that the functional view and the physical view, respectively, are indistinguishable to the host). Together, functional and physical equivalence imply IPRR.

4.1 Modeling State

We model circuit state as the values of all the internal registers and wires. To move between states, we use a `step` function that describes how the circuit changes after executing for one cycle, which we obtain through parsing the implementation code. The circuit's TRNG state, as described in the previous chapter, is represented as a stream of bits.

Specification state is modeled as a combination of the state supplied by the developer (for instance the variable `secret` in the password hasher in Figure 1.1) which we refer to as the oracle state, the TRNG state, and the emulator state. The oracle and emulator state are supplied by the developer, but the TRNG state is represented the same way as the circuit's TRNG state: as a stream of bits.

4.2 Refinement Relation

To prove the properties of physical and functional equivalence, we use the concept of a refinement relation, denoted by R, to establish the connection between the state of the implementation and the state of the specification. We require that this relation holds in between spec-level operations: before or after querying the spec or when switching between the physical and functional views, but the relation is not required to hold at arbitrary steps within an operation. The relation is a proof artifact supplied by the developer and imposes constraints that allow us to reason between the real and ideal worlds. For instance, in the password-hasher, the refinement relation allows us to identify which locations in memory hold the secret. For example, by having R in the form `spec.secret = impl.fram[1...5]`, we specify that after calling `config`, the secret is saved in locations 1 through 5 (the logic is more complex in the actual password-hasher but the core idea still holds). Then, when the implementation returns the values of locations 1 through 5, we are able to verify that these values correspond to the specification's secret.

4.3 Functional Equivalence

Functional equivalence asserts that the behavior at the specification level can be derived from the circuit's wire-level interface by following the specified I/O protocol of the driver, given the same initial TRNG state; in other words, the circuit correctly implements the specification and outputs the same values when the circuit's TRNG stream and the spec's TRNG stream output the same bits. More formally, functional equivalence states that for all circuit and specification states related by R and for all bit streams t such that the circuit's TRNG and the spec's TRNG both have state t, after invoking an operation on the specification and the same driver function on the circuit, the output is the same and the final circuit/spec states continue to be related by R. Furthermore, the final TRNG states of the circuit's TRNG and

the spec's TRNG are the same (we have queried the same number of bits in both streams). We illustrate functional equivalence in Figure 4.1.

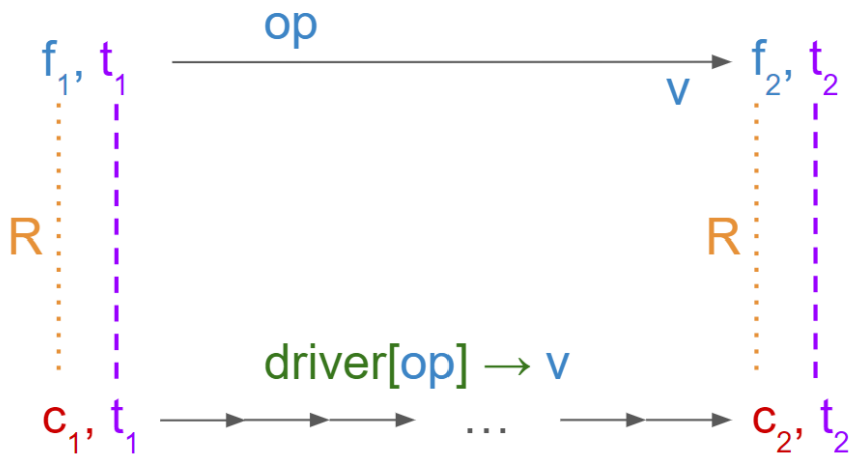

Figure 4.1: Functional equivalence for IPRR, which states that for all implementation states c_1 and spec states f_1 that are related by R, for all spec-level operations op, and for all TRNG streams t_1 such that both the implementation's TRNG and the spec's TRNG have initial state t_1: (a) the spec-level output v matches the driver output (b) the final states c_2 and f_2 are related by R (c) the final state of the circuit's TRNG t_2 matches the final state of the spec's TRNG

4.4 Physical Equivalence

Physical equivalence states that the circuit's wire-level behavior can be matched by running an emulator, which reflects the idea that the circuit is secure and leaks no information beyond the specification. The emulator is an artifact supplied by the developer that translates wire-level I/O into queries to the specification. We define physical equivalence as follows: starting from circuit/spec states related by R and any bit stream t such that the circuit's TRNG and the spec's TRNG both have state t, any wire-level 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

and the emulator shuts down), and the final TRNG states of the circuit's TRNG and the spec's TRNG match. We illustrate physical equivalence in Figure 4.2.

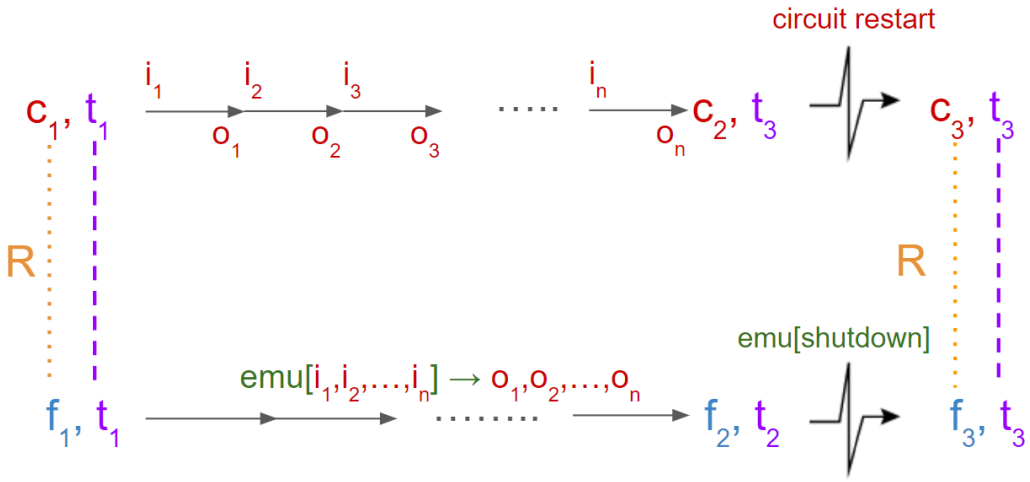

Figure 4.2: Physical equivalence for IPRR, which states that for all spec states f_1 and implementation states c_1 that are related by R, for all bit streams t_1 such that both the implementation's TRNG and the spec's TRNG have initial state t_1, and for all wire-level inputs $i_1, ..., i_n$: (a) the circuit outputs $o_1, ..., o_n$ match the emulator outputs (b) the final states f_3 and c_3 (f_2 after shutdown and c_2 after a reset respectively) are related by R (c) the final state of the circuit's TRNG t_3 matches the final state of the spec's TRNG after emulator shutdown

The transition from c_2 to c_3 and f_2 to f_3 models switching from physical view to functional view. This transition can be performed at any cycle during execution in the physical view. In the real world, the circuit is reset, while in the ideal world, the emulator runs its shutdown operation. The circuit's TRNG stays the same before and after the transition, as the circuit cannot query the TRNG when it is being reset. However, the emulator's `shutdown` operation is able to make changes to the TRNG; for example with the `discard` function that allows it to discard a bit from the specification's TRNG.

A use case for the `shutdown` function is with implementations that query the TRNG multiple times for a single operation (or that query the TRNG once but must perform other operations as well); then if the circuit is reset before all the queries are performed, the circuit's TRNG is left in an intermediate state. For the specification's TRNG to reflect this, we use can use the `shutdown` and `discard` operations to bring the spec's TRNG to an

equivalent state.

4.5 Analysis

Together, functional equivalence and physical equivalence, along with an initialization property that simply says that the initial circuit/specification states satisfy R, imply IPRR.

We can understand why this is the case using induction. In IPRR, we assert that the real and ideal worlds are indistinguishable for any stream of operations. We can break down this stream of operations into smaller segments by the type of view: physical or functional, and further break down execution in functional view into executions of individual operations. For each segment, we want to show that real/ideal world indistinguishably holds, but for most implementations this is not possible without further assumptions, thereby giving us the need for the refinement relation. We further need to assume that the TRNG state is equivalent at the start of the operation to relate the randomness in the real and ideal worlds.

This leads us to the following inductive hypothesis: if we have functional and physical equivalence along with the initialization property, then for all streams of operations, real/ideal world indistinguishably holds for all operations in the stream, and R and TRNG state equivalence hold at the end of the stream. We can prove the hypothesis through standard induction with the following two properties.

- The base case is true: R holds by the initialization property, and the TRNG state is initialized to be the same for both worlds by our model definition.

- The induction step holds: Assuming that at the end of the last segment of operations, R and TRNG state equivalence holds, then for all subsequent segments of operations, real/ideal world indistinguishability is maintained. Furthermore R holds and TRNG states are equivalent at the end of execution. This is captured by functional and physical equivalence.

Chapter 5

Karatroc

This chapter describes how we implement the aforementioned proof technique through our tool, Karatroc. We developed Karatroc as an extension to the Knox framework [3], using the Rosette library [12] for the programming language Racket. Karatroc is programmed such that it preserves functionality for HSMs without randomness; users can specify which mode they want using the `random` flag in the specification. Karatroc adds about 274 lines of code to the base Knox framework.

Karatroc proves IPRR using symbolic execution by representing circuit, specification, and TRNG state in terms of symbolic variables, then making solver-aided queries through the Rosette library to verify our desired properties.

5.1 TRNG State Representation

In our theoretical model, we view TRNG state as a stream of bits. To implement this idea in Karatroc, we represent TRNG state instead as a list of symbolic booleans in order to support solver-aided verification queries on these values. Since there are no additional assertions or assumptions on the terms in the TRNG state, the symbolics could take any combination of values, representing all possible outputs from the TRNG.

The length of the list is a constant inputted by the developer through the flag `max-trng-bits`,

representing the maximum number of random bits that a given operation can take. If an operation takes at most n random bits, then a finite list of $> n$ bits is functionally equivalent to an infinite stream of bits. Although there can still be an infinite amount of operations, because we break down the proof of IPRR into proofs of physical and functional equivalence, we are able to analyze operations one at a time, making it so that our finite list representation is sufficient in most cases.

However, limiting the length of the list to a constant does exclude us from verifying HSMs that use an unbounded number of random bits. Even so, many of these HSMs still can be verified by pivoting to a different interface, which we describe in Chapter 8. Furthermore, this representation greatly simplifies the implementation logic, and our framework contains proper error handling for cases where the developer attempts to use more than the specified number of random bits.

5.1.1 TRNG Update Logic

To implement the execution logic of the TRNG in the real world, we modify the Knox's circuit `step` function, which models running the circuit for one cycle. When `step` is called, in addition to updating the circuit state as usual, we check the `trng_next` wire and update the TRNG state using the following function in Figure 5.1.

```
update_trng(trng_next, trng_state) -> trng_bit:
    if trng_next:
        trng_state = trng_state[1:]
    return trng_state[0]
```

Figure 5.1: The `step` function, which describes how the TRNG state and outputs change at every cycle.

In the ideal world, the developer defines how queries to the specification's TRNG state are handled. Each operation in the specification has access to the TRNG state and can make modifications to it. In Karatroc, the developer must directly implement TRNG state reads and updates; for instance, in order to read a random bit like in the `get_rand_bit()` function

in Figure 1.1, the developer must read the first item from the TRNG state list, then discard that item from the list. Since the specification is trusted, we expect the developer to correctly implement TRNG state reads and updates, but we will discuss further improvements to the framework in Section 9.2.

5.1.2 Proving Functional Equivalence

To prove functional equivalence, we initialize a new circuit state and specification state such that the refinement relationship R holds. We also initialize a new TRNG state, and a copy is given to both the circuit and the specification. We then run both the circuit (following the instructions supplied by the driver) and the specification, and use Rosette's solver-aided queries to verify that the output is the same, R still holds, and the final TRNG states of both the circuit and the specification are the same.

5.1.3 Emulator Implementation

In IPRR, in addition to having access to spec-level operations, the emulator also has access the a `discard` function that allows it to discard bits from the specification's TRNG state. We implement this through Knox's `leak` functionality. Knox allows developers to specify when it is okay for the implementation to leak non-sensitive information by creating a `leak` function in the specification. The `leak` function can be invoked by the emulator, but unlike other specification functions, it does not need to be verified through functional equivalence as it is expected that a well behaved host would not execute this function.

We are able to implement the `discard` function through the `leak` framework by modeling `discard` as a leak that returns nothing but modifies the specification's TRNG state.

5.1.4 Proving Physical Equivalence

Similar to functional equivalence, we start by initializing a new circuit state and specification state such that the refinement relationship R holds and give both the same initial TRNG state. We also initialize an emulator, supplied by the developer.

To verify physical equivalence, we adapt Knox's technique of guided symbolic model checking [3], which is a method that involves exploring every possible state starting from the initial state. At every step of the exploration, we check that the outputs of the circuit and emulator match. Furthermore, we check that the R holds and the circuit's and spec's TRNG states are equivalent after resetting the circuit and running the emulator's `shutdown` operation.

To prevent exploration from continuing infinitely, Knox introduces the concept of subsumption. Intuitively, if a given symbolic state A with predicate p_A is subsumed by another symbolic state B with predicate p_B, then that means that the set of all possible concrete values that A under p_A can evaluate to is a subset of all possible concrete values that B under p_B can evaluate to. Knox provides an efficient algorithm for subsumption checking, so therefore, when we reach a state that we have already covered in our exploration, we check that it is subsumed by a previous state and finish that branch of exploration. Because we already check that the circuit's and spec's TRNG states are equivalent after reset on every step, there is no need for additional checks in relation to TRNG state on subsumption.

5.1.5 Hints

Karatroc additionally supports Knox's hints, which allow for human guidance in proofs. All hints are untrusted and are verified by the framework before application. Karatroc provides all of Knox's primitive hints, such as `CONCRETIZE` to replace a symbolic variable with a concrete one, `REPLACE` to rewrite and simplify a symbolic, and `OVERAPPROXIMATE` to replace a term with a fresh symbolic. In addition, Karatroc provides a `REPLACE-TRNG` hint as a

specialized version of `REPLACE` for easy simplification of the TRNG state.

Chapter 6

Case Studies

In this chapter we present two HSMs with TRNGs verified with Karatroc to illustrate the proof process, demonstrate the key techniques used, and analyze where in the proof the TRNG can add complexity.

The first case study is toy example of a random byte generator that takes eight random bits and concatenates them into a random byte. In this case study, we present a detailed breakdown of the components of the proof.

The second is a password-hasher HSM; this is a HSM that has been verified using the original Knox framework that we have modified to query random bits instead of requiring the host to supply a secret. The password-hasher is more complex and runs on a PicoRV32 RISC-V CPU, and verification of this HSM demonstrates the applicability of Karatroc to more complex hardware and procedures.

Code for both HSMs and their proofs are available at https://github.com/katherinezhao02/karatroc-hsm.

6.1 Random Byte Generator

6.1.1 Functional Specification

The specification of the random byte generator holds no state and consists of only a single operation, `get-random-byte` which returns a random byte. We depict a Python style version of the specification in Figure 6.1 for readability, and the real specification, written in Racket, is included in Appendix A. In Karatroc, the `discard` function that the emulator uses to discard bits from the TRNG is also contained within the specification file.

```
var TRNG

def get-random-byte():
    ret = 0
    for i in range(8):
        ret += TRNG[i] << (7-i)
    TRNG = TRNG[8:]
    return ret

def discard(n):
    TRNG = TRNG[n:]
```

Figure 6.1: The specification of the random byte generator, which takes eight random bits and makes a random byte

6.1.2 Implementation Details and Driver

The implementation of this HSM consists of a single Verilog module. The interface consists of the inputs `reset`, `en`, and `req` to reset the circuit, enable execution, and request a new random byte, and the outputs `random_word` that contains the random byte and `output_valid` that goes high when the random byte is available. The code for the implementation is included in Appendix B.

The driver is also relatively simple, consisting of setting `en` and `req` to be true, running the circuit for eight cycles, getting the output, and then running for one additional cycle to

reset the circuit.

```
(define (tick-n n)
  (if (zero? n)
      (void)
      (begin
        (tick)
        (tick-n (sub1 n)))))

(define (get-random-byte)
  (out* 'en #t 'req #t)
  (hint concretize-trng)
  (tick-n 8)
  (let ([r (output-random_word (in))])
    (tick)
    r)
  )
```

Figure 6.2: The driver of the random byte generator. `tick-n` is a helper function that runs the circuit for `n` cycles. `get-random-byte` contains the driver code. On the first line we set the inputs `en` and `req` to be true. We then invoke a hint, described in Section 6.1.3, and run the circuit for 8 cycles to sample 8 bits. Finally, we run for one additional cycle to reset the circuit and return the output, which we get by invoking the `(output-random_word (in))` function.

6.1.3 Proof process: Functional Equivalence

The proof process for functional correctness is relatively straightforward. The only significant proof component relating to the TRNG is the `concretize-trng` hint which replaces the `trng_next` output and all related variables with concrete values. This is necessary to make sure that the TRNG state has a concrete length, otherwise it would be represented as a union of two or more lists (for instance, if the `want_next` register was a symbolic, then the TRNG state would be in the form Union(`want_next` $\rightarrow$ List($t_2, t_3, ...$), `!want_next` $\rightarrow$ List($t_1, t_2, t_3...$))). The union structure is difficult for the framework to reason with and can cause it to produce assertions that lead future queries to fail.

6.1.4 Proof process: Writing an Emulator

The emulator for the random byte generator is depicted in Figure 6.3. It has state consisting of the field `circuit`, a copy of the circuit that holds dummy data, and the field `num-trng`, representing the number of TRNG bits to discard on shutdown. Reading and writing to the emulator simply involves reading and writing to the circuit copy. The `step` function involves some more logic. Since the emulator cannot directly access the TRNG, it must query the specification. Thus, we design the emulator such that when `valid` is high (indicating the circuit is about to return a random byte), it queries the specification to obtain the random byte and replaces the output. Additionally, the `step` function detects instances when the circuit is about to query the TRNG and increments `num-trng` accordingly. This ensures that if the circuit restarts before all eight bits are sampled, the emulator can still synchronize the TRNG state by calling `discard(num-trng)` on shutdown. If the circuit successfully samples all eight bits and returns the random byte, `num-trng` is reset to zero.

6.1.5 Proof process: Physical Equivalence

When proving physical equivalence, we encounter the necessity for more guidance in our proofs. One reason is due to the fact that there are more opportunities for the TRNG state to become a union when working with arbitrary wire-level I/O. For instance, at every cycle, the TRNG state depends on `en`, so we must do casework to concretize its value. This was not required in functional equivalence because the driver specified that `en` is high for the whole operation, but in physical equivalence the input could take any value.

Another area of additional complexity arises from the representation of symbolic variables and how subsumption operates in relation to them. When random bits are concatenated, the framework creates a long symbolic expression in a form such as `(concat (ite trng-bit$80f.. (bv #b1 1) (bv #b0 1)) (concat (ite trng-bit$f7f.. (bv #b1 1) (bv #b0 1)) ...)`, and making subsumption checks with these expressions is expensive.

```
(struct state (num-trng circuit))

(define (init)
  (set! (state 0 (circuit-new))))

(define (with-input i)
  (set! (state (state-num-trng (get)) (circuit-with-input
     (state-circuit (get)) i))))

(define (get-output)
  (circuit-get-output (state-circuit (get))))

(define (step)
  (let ([c (circuit-step (state-circuit (get)))]
        [t (state-num-trng (get))])
    (if (and
         (equal? (get-field c 'valid) (bv 1 1))
         (get-field c 'en)
         )
        (set! (state (- t 8) (update-field c 'cur_word
           (spec:get-random))))
        (set! (state t c))))

  (let ([c (state-circuit (get))]
        [t (state-num-trng (get))])
    (if (and
         (equal? (get-field c 'want_next) (bv 1 1))
         (get-field c 'en))
        (set! (state (+ t 1) c))
        (void)))
)

(define (shutdown)
  (spec:discard (state-num-trng (get)))
```

Figure 6.3: The emulator of the random byte generator. The `init` function initializes the emulator as a copy of the circuit with all registers zeroed out and `num-trng`, the number of TRNG bits to discard on shutdown, as 0. The `with-input` and `get-output` functions simply read and write to the circuit copy. The `step` function updates the circuit copy with the random byte from the specification when needed and updates `num-trng` as well. Finally, `shutdown` discards the corresponding number of TRNG bits from the spec TRNG state on circuit restart.

Combined with the fact that we must perform many instances of casework and each case creates a new branch that we need to subsume, this causes the proof of physical equivalence

to run for over a minute, even though the circuit itself is simple. We are able to reduce the runtime by replacing the long symbolic expressions with fresh symbolics, but this adds additional complexity to the proof.

6.2 Password-Hasher

The second HSM that we verify is a password-hasher HSM, which is a HSM that is used to compute peppered password hashes. A peppered hash is similar to a salted hash, but instead of having a different salt for every password and storing the salt along the hash, the pepper is kept secret on the HSM. This HSM is adapted from a similar password-hashing HSM that does not use a TRNG, which was verified with Knox. We change the HSM such that instead of requiring the host to supply a random secret, we are able to generate the secret internally in the HSM.

6.2.1 Functional Specification

The specification is displayed in Figure 6.4, which is an equivalent version of the specification shown in Figure 1.1 but adapted to the new framework and TRNG state representation. Like with the random byte generator, the specification in Figure 6.4 is a Python style adaptation for readability, and Karatroc's specification is provided in Appendix C.

Compared to the original password-hasher, we change the `config` function to sample random bits from the TRNG instead of requiring the host to input a secret. The specification for the original password-hasher is displayed in Figure 6.5 to serve as a comparison.

6.2.2 Implementation Details and Driver

The implementation of the password-hasher consists of C code with a PicoRV32 CPU. The wire-level interface is presented in Figure 3.3. To enable interaction between the PicoRV32 and the TRNG, we create a MMIO module that translates reads to the `x40007000` address to

```
var secret = 0
var TRNG
var SECRET_LENGTH = 160

def config():
    secret = TRNG[0:SECRET_LENGTH]
    TRNG = TRNG[SECRET_LENGTH:]

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

def discard(n):
    TRNG = TRNG[n:]
```

Figure 6.4: The specification of the password-hashing HSM.

```
var secret = 0
var SECRET_LENGTH = 160

def config(new_secret):
    secret = new_secret

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

Figure 6.5: The specification of the original password-hashing HSM that requires the host to input the secret.

a query of eight bits to the TRNG. Then, the C code for the `config` function simply involves reading from address `x40007000` 20 times (since our secret has length 160), as presented in Appendix D.

The driver involves minimal changes compared to the original driver code for the password-hasher without randomness, the only main difference being that there is no need for the host to send the secret to the HSM, since it can now generate a random secret by itself.

6.2.3 Proof Process

The proof for functional correctness also involves minimal modifications, with the only main change being the addition of the `concretize-trng` which functions the same way as in the

random byte generator. Same could be said for the emulator, where the only main additions are the `shutdown` function that calls `discard` the appropriate number of times, and an additional emulator state to store how many TRNG bits should be discarded.

The proof for physical equivalence involves more data processing, for similar reasons as the random byte generator. Because concatenated TRNG bits create long symbolic expressions, the final subsumption checks are very expensive, even more so here since the secret involves 160 random bits, and cause the subsumption check to time out. To resolve this, we must replace the long symbolic representation of the secret with fresh symbolics using the `REMEMBER`, `REPLACE`, and `CLEAR` hints.

Other than these hints however, the proof for physical equivalence actually involves less complexity in relation to the TRNG than that of the toy random byte generator HSM. The reason for this is mainly due to the fact that after the host sends the initial bits, the execution of the circuit in relation to the TRNG is deterministic. The circuit will always query the TRNG 20 times in batches of 8, while in the random byte generator HSM, the host has an ability to interrupt the TRNG queries at any time by turning off the `en` wire. The deterministic nature of the execution in the password-hasher prevents the TRNG state from becoming a union data structure that our framework has difficulty processing and eliminates the need to branch every cycle.

Chapter 7

Evaluation

We evaluate Karatroc with the following questions:

- Can IPRR and Karatroc be used to verify HSMs with TRNGs? How practical is using Karatroc to verify HSMs?
- What is the performance of Karatroc? How long does it take for machines to execute and verify our proofs in Karatroc?
- How long are the proofs written with Karatroc?
- How does performance and proof length scale with the size of the implementation and the amount of random bits used?

In this chapter, we start by answering the second and third questions through metrics from the password-hasher. We measure performance by timing how long it takes for our machine to run the physical and functional equivalence proofs, and we analyze proof length by looking at the ratio between the lines of code it takes for the implementation and the lines of code it takes for the proof. To serve as a comparison, we also include metrics for the password-hasher without a TRNG that was verified with Knox. Through our comparison, we are able observe if proofs with Karatroc are significantly more expensive than proofs with existing methods.

Then, we present an overarching analysis of all four questions in Section 7.3. We detail our overall conclusion of the performance and proof complexity of Karatroc as well as the main bottlenecks. We also analyze how runtime and proof complexity scale. Finally, we evaluate how practical and applicable Karatroc is to verifying HSMs with TRNGs.

7.1 Proof Length Metrics

We compare the size of the implementation and proof for the password-hasher with randomness to that of the original password-hasher. The results are displayed in Table 7.1. We break down the implementation into the hardware components (such as the PicoRV32 description) and the software components (the C code), and the proof into R, functional equivalence, physical equivalence, and the emulator.

HSM	HW	SW	**Total Impl**
PWHash w/ TRNG	3115	244	3359
PWHash w/o TRNG	3020	240	3260

HSM	R	Func eq	Phys eq	Emulator	**Total Proof**	**Proof : Impl**
PWHash w/ TRNG	110	143	385	77	715	**0.213**
PWHash w/o TRNG	106	142	369	72	689	**0.211**

Table 7.1: Lines of code for the implementation and proofs of HSMs, as well as a lines of proof to lines of implementation ratio. Implementation code is broken down into hardware and software components. Proof code is broken down into R, functional equivalence, physical equivalence, and emulator.

We observe that there is not much increase in proof size, with only a 1% difference in the proof to implementation ratio between the two HSMs. Indeed, R, functional equivalence, and the emulator only increase in size for a maximum of 5 lines each. The proof for physical equivalence increases by 16 lines to support symbolic variable simplification, but this is insignificant to the overall size of the proof. Therefore, we can conclude that verifying HSMs with TRNGs using Karatroc does not necessarily impose a significant additional burden on the developer than verifying one without a TRNG using previously existing tools.

7.2 Performance Metrics

To analyze the performance of the password-hasher with randomness, we compare the runtimes of the functional equivalence and physical equivalence proofs to those of the password-hasher without randomness. Results are shown in Table 7.2.

HSM	Func Eq	Phys Eq	Total Time
PWHash w/ TRNG	41min	10min	51min
PWHash w/o TRNG	61min	4min	65min

Table 7.2: Runtime for the proofs of each HSM in minutes, evaluated on a 2022-era AMD Ryzen 9 5900HX.

The password-hasher with the TRNG actually outperforms the password-hasher without a TRNG in functional equivalence by a significant amount, taking only 41 minutes to run as opposed to 61 minutes. The main reason for the decrease in runtime is the change in the interface for the `config` function; in the original password-hasher, the host needs to send the secret to the HSM using the UART protocol, and verifying this portion of the execution is expensive. This is because UART is an asynchronous protocol, and the host is allowed to wait an arbitrary number of cycles before sending and receiving bytes to the HSM. Therefore, when we symbolically execute this protocol, we must branch for every possible state of the circuit that could exist while the host waits. This turns out to be very expensive, especially since we must send bytes 20 times to make up the 160 bit secret. In contrast, because the password-hasher with the TRNG can generate the secret internally rather than needing the host to supply it, we are able to simplify the proof.

Verifying physical equivalence is more expensive with the TRNG. One reason for the increase in cost is the fact that we need to check TRNG equivalence at every step, and since physical equivalence runs the circuit for thousands of steps, this causes a noticeable increase in runtime. A second reason is that the TRNG bits cause some values to turn into long symbolic expressions, slowing down execution.

Even so, the overall runtime for the password-hasher with the TRNG is lower than the one without, allowing us to conclude that the time cost in verifying HSMs with TRNGs using Karatroc is not necessarily higher than that in verifying a HSM without a TRNG.

7.3 Overall evaluation

Through the verification of the HSM, we are able to analyze performance and complexity of proofs written with Karatroc and answer our original evaluation questions.

On practicality, performance, and proof length, we see that proofs written in Karatroc are not significantly more complex or runtime expensive than proofs in the baseline Knox framework. This shows that we are able to verify HSMs with TRNGs without incurring substantially more cost than what we would spend verifying HSMs without TRNGs. In fact, adding a TRNG may simplify some HSMs and cause proofs to be easier by enabling the design of verifiable HSMs that can directly sample random bits from a TRNG instead of requiring additional communication from the host.

We are also able to analyze the proofs of the two HSMs and identify the main sources of complexity associated with the TRNG. These sources include:

- The need for casework when faced with nondeterministic queries to the TRNG to prevent the TRNG state from turning into a union data structure.
- Expensive subsumption checks when the TRNG bit representation causes some values to turn into long symbolics.
- The additional check to make sure that TRNG states for the circuit and specification are the same after circuit reset and emulator shutdown. We must perform this check at every step in the physical equivalence proof.

When it comes to scalability, the actual impact on runtime and proof complexity depends largely on how the implementation utilizes the random bits and how hints in proofs are

used to simplify symbolic variable representations. However, we do observe that even when we sample 160 random bits with the password-hasher, the proof complexity and runtime remain generally unchanged compared to the proof of the password-hasher without a TRNG, suggesting that the impact of the TRNG on the proof does not tend to scale in a signficant way.

Finally, on ability to verify HSMs with TRNGs, we demonstrate that Karatroc is able to verify small HSMs with TRNGs. Although many existing HSMs on the market are much more complex and serve more than just one function (for instance, the Thales Luna Network HSM [5] is able to encrypt data, store keys, perform signatures, and more), Karatroc's password-hasher contains implementation features found in real-world HSM hardware and software: for example, the microprocessor, I/O peripheral, persistent memory, and cryptography.

Chapter 8

Discussion

This chapter discusses some design decisions made in IPRR and Karatroc, and their implications and limitations.

8.1 TRNG Model

We model the TRNG in Karatroc after real world TRNGs that output one bit every cycle. One notable difference between our model and real-world instances is the `trng_next` wire, which is similar to the `enable` wire in existing TRNGs but with an important difference. The output is undefined when `enable` is false, but the circuit holds on to the last output when `trng_next` is false. Our representation allows us to simplify the behavior of TRNGs for easier reasoning, and while real-world TRNGs may behave differently, it is straightforward to adapt a TRNG interface with `enable` to our model. We can do this by setting `enable` to be true, then adding a simple intermediate circuit (such as a D Flip Flop) that stores the random bit until the HSM requests it.

Another interesting point to note is that including the `trng_next` input makes verifying the HSM more feasible. If we abstracted away the wire so that the TRNG model simply outputted one bit every cycle, the TRNG state would have be the length of the whole execution, which would be difficult to represent for operations that could take an unbounded

number of cycles. Furthermore, the specification would need to know exactly which cycle that the HSM uses the random bits so that it can describe its change in state or output in terms of those bits. We believe that this places too much burden on the developer, especially with complex implementations that involve a CPU or operations where the host can wait for an unspecified number of cycles through the `yield` function.

8.2 TRNG State Representation

To implement the TRNG in Karatroc, we modeled the stream of random bits as a finite length list of symbolic bits. We made this decision to simplify our implementation logic. However, this representation excludes us from verifying HSMs that use an unbounded number of random bits per operation, notably those that use rejection sampling and read random bits until it gets a value that fits its requirements.

Even so, we can still verify these HSMs through Karatroc using an alternate structuring of the specification. The specification could expose a "retry" output, where the HSM would output "retry" after it samples a certain number of random bits without getting its desired value and prompt the user to run the operation again. For this reason, we believe that our representation of TRNG state is reasonable.

8.3 TRNG Interface Selection

In Section 2.2, we introduced the two main interfaces of existing TRNGs. For our definition of IPRR, we modeled our TRNG after the interface in which the TRNG outputs a bit every cycle. This interface's predictable behavior allows us to create a concise and intuitive security definition as well as a proof technique that, as analyzed in the evaluation section, poses no significant additional cost to the developer. However, this interface may not be suitable for every HSM. For instance, if a HSM needs to create a random string that has a long length, it must concatenate many TRNG bits, as performed in the password-hasher example. The

HSM would need to wait for the TRNG to generate bits one by one, which may not be suitable for HSMs where performance is important. Furthermore, the concatenation process can lead to long symbolics that depend on each TRNG bit, increasing proof complexity.

In contrast, the second TRNG interface, where a longer random word is outputted after some delay, may be a better fit for some HSMs. With this interface, we can directly sample longer random words rather than needing to wait and concatenate them bit by bit. In our Future Work section, we propose a new IPRR definition for this interface (Valid/Ready IPRR).

Chapter 9

Future Work

9.1 Valid/Ready IPRR

Future work includes presenting a definition that captures correctness and security for HSMs that use the valid/ready style interface of TRNGs. In this section we propose a definition for Valid/Ready IPRR (V-IPRR), as well as show preliminary work in developing a proof strategy for V-IPRR and implementing a tool to prove that a HSM satisfies V-IPRR. The main challenge with working with this model of TRNG is that the TRNG can leak information through its timing behavior, and therefore in our definition we capture the idea that no secret information is exposed to the TRNG.

9.1.1 Valid/Ready IPRR Definition

TRNG Model

We model the physical TRNG based on real world TRNGs that output random words after some delay, such as the neoTRNG [11], the microRNG [9], and the second interface of the RPG100 [10].

Our TRNG model uses a standard valid/ready interface, where the random byte is transferred from the TRNG to the HSM when both the HSM says "ready" and the TRNG says

"valid". The interface of the TRNG consists of three wires: the input `trng_req` for the HSM to indicate that it is ready to receive the random word, the output `trng_valid` that goes high when the TRNG has a valid random word, and the output `trng_word` that contains the actual random word.

It is reasonable to assume that the TRNG's random words are independent of all other variables, so we represent the internal state of the TRNG model as a stream of words. In contrast, the delay that the TRNG takes before outputting the next word may be dependent on other factors. For instance, in the RPG100 [10], the TRNG internally stores up to 32 random words. This means that if the HSM requests many random words in succession and uses up the stored random words, the delay may be longer than if we have not requested any random words in a while. Therefore, to model the delays, we represent it as being generated by an uninterpreted function that takes as input the past history of requests.

More specifically, the TRNG update logic follows the code outlined in Figure 9.1.

```
history = []
stream = new Stream()
delay = randInt()
delay_gen = new UninterpretedFunction()
cycle_count = 0

update_trng(trng_req) -> (trng_valid, trng_word):
    valid = False
    word = 0
    if delay == 0:
        valid = True
        if trng_req:
            history.append(cycle_count)
            word = stream.next()
            delay = delay_gen(history)
    else:
        delay -= 1
    cycle_count += 1
    return valid, word
```

Figure 9.1: The update logic of the valid/ready TRNG model.

Implementation

The HSM implementation follows the structure in the original IPRR, but now with the `trng_req`, `trng_valid`, and `trng_word` wires to communicate with the TRNG.

Specification and Timing Simulator

The TRNG state for the ideal world matches that of the real world, namely `history`, `stream`, `delay`, `delay_gen`, and `cycle_count`. However, the specification only has access to the `stream`, since the timing behaviors of the circuit and TRNG are implementation specific details. To interact with the other TRNG state variables, we introduce a new object, the timing simulator. The timing simulator is an object that has access to the `history`, `delay`, `delay_gen`, and `cycle_count` states and its own internal state. It exposes only one operation, `simulate`, that is automatically called after every spec-level operation. Like the emulator, the timing simulator is a proof artifact supplied by the developer.

However, this alone is not enough for the timing simulator to feasibly simulate the timings of each operation. Drivers may include the `yield` function, which represents the host waiting for arbitrary number of cycles. Without knowing how long the host waits, the ideal world cannot simulate the timings of the real world. Therefore, in addition to the previous states, the timing simulator also has access to the `yield` timings of the host in the form of a stream, where the ith request to the stream outputs the amount of time the host waits in the ith yield call.

Emulator

In addition to having access to the `discard` function, the emulator also has access to a `get_timing` function, which returns the `history`, `delay`, `delay_gen`, and `cycle_count` states. Security is still maintained because these states are only modified by the timing simulator, which knows nothing except for the number of operations invoked and the `yield` timings of the host. We assume that the `yield` timings of the host do not reveal any

information, which we believe is reasonable since we assume that hosts are well-behaved under the functional interface. The number of operations invoked is also leaked, but we believe that this is acceptable for most HSMs, and we will discuss this more in Section 9.1.4. Therefore, the emulator can learn no additional information, except for the number of previous operations, from the exposed states, keeping the ideal world secure.

Definition

With these components, we can present our definition of Valid/Ready IPRR, which captures the idea that the HSM correctly implements the specification and leaks no additional information beyond the number of previous operations invoked. Valid/Ready IPRR follows the same real/ideal world structure as the original IPRR and contains the same functional and physical views. Similarly, we can also switch between views; switching from functional to physical reinitiates the emulator and the timing simulator, and switching from physical to functional reinitiates the driver and the timing simulator. Furthermore, switching from physical to functional resets the `history` and `delay` states of both the TRNG state in the real and ideal worlds to an empty list and a new random integer, as we can assume that when the circuit gets reset the TRNG does as well.

Then, Valid/Ready IPRR states that there exists an emulator and a timing simulator, such that for all TRNG states where the real and ideal worlds are initialized to the same TRNG state, and for all yield timings, the real and ideal worlds are indistinguishable to the host, as depicted in Figure 9.2.

9.1.2 Proof Strategy

The proof strategy for V-IPRR follows the same general structure as the original IPRR, but we make changes to functional and physical equivalence to account for the nondeterministic delay before a TRNG word is outputted. We also split functional equivalence into two components: functional output equivalence, which mirrors the original functional equivalence

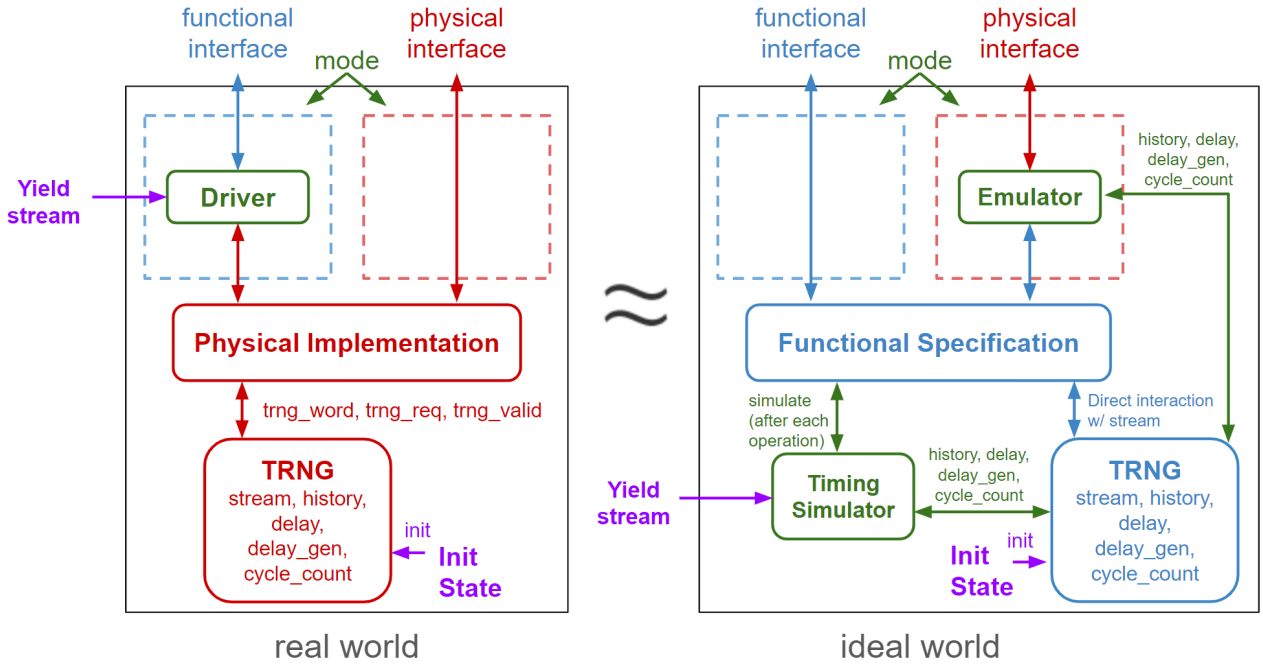

Figure 9.2: Valid/Ready Information-Preserving Refinement with Randomness (V-IPRR), which asserts that the real and ideal worlds are indistinguishable given access to randomness in a setting where the TRNG has a valid/ready interface.

in IPRR, and functional timing equivalence, which ensures that spec-level operations in the real world are able to be correctly simulated by the timing simulator.

V-IPRR Functional Timing Equivalence

Functional timing equivalence states that for all circuit states c_1, c_2 and specification states f_1, f_2 such that $R(c_1, f_1)$ and $R(c_2, f_2)$ holds, for all TRNG states t such that c_1, c_2, f_1, f_2 all have TRNG state t, and for all yield timings streams y, after invoking any spec-level operations op_1 on c_1 and op_2 on c_2 such that the ith yield of each operation waits for time matching the ith output of y, the final TRNG states of c_1 and c_2 are the same. This ensures that the interaction between the implementation and the TRNG does not depend on any secret state, therefore no information is leaked to the TRNG and thus the timing simulator can simulate the behavior. Functional timing equivalence is depicted in Figure 9.3.

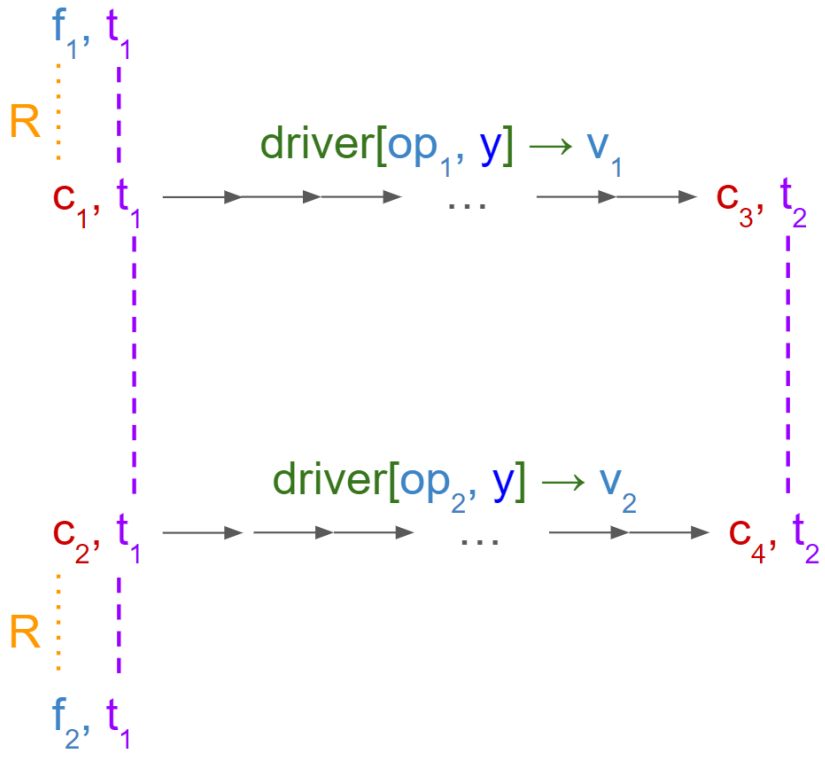

Figure 9.3: Functional timing equivalence for V-IPRR, which states that for all implementation states c_1, c_2 and spec states f_1, f_2 where $R(c_1, f_1)$ and $R(c_2, f_2)$ hold, for all spec-level operations op_1, op_2, for all yield timing streams y, and for all TRNG streams t_1 such that both the implementations and specifications have initial TRNG state t_1: the final state of the TRNG t_2 matches for both implementations.

V-IPRR Functional Output Equivalence

Functional output equivalence states that for all circuit and specification states related by R, for all TRNG states t such that the circuit's TRNG and the spec's TRNG both have state t, and for all yield timings stream y, after invoking an operation on the specification and the same driver function on the circuit, the output is the same and the final circuit/spec states continue to be related by R. Furthermore, the final TRNG states of the circuit's TRNG and the spec's TRNG are the same. Functional output equivalence is depicted in Figure 9.4.

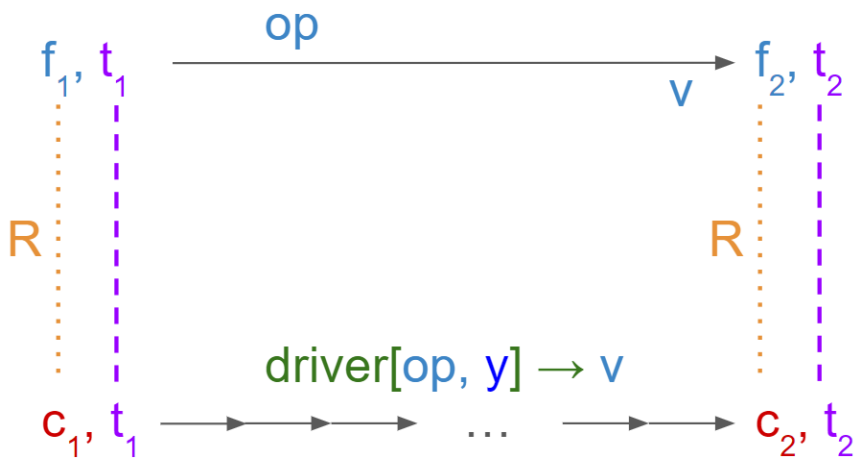

Figure 9.4: Functional output equivalence for V-IPRR, which states that for all implementation states c_1 and spec states f_1 that are related by R, for all spec-level operations op, all yield timings y, and all TRNG streams t_1 such that both the implementation's TRNG and the spec's TRNG have initial state t_1: (a) the spec-level output v matches the driver output (b) the final states c_2 and f_2 are related by R (c) the final state of the TRNG t_2 matches the final state of the TRNG

V-IPRR Physical Equivalence

Physical equivalence for V-IPRR states that starting from circuit/spec states related by R and any TRNG state t such that the circuit's TRNG and the spec's TRNG both have state t, any wire-level 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 and the emulator shuts down) and the final TRNG `stream` and `cycle_count` states match. Note that we do not require all components of the final TRNG states to match. This is because after the circuit is reset we also assume that the TRNG history and delays are reset as well. Physical equivalence for V-IPRR is depicted in Figure 9.5.

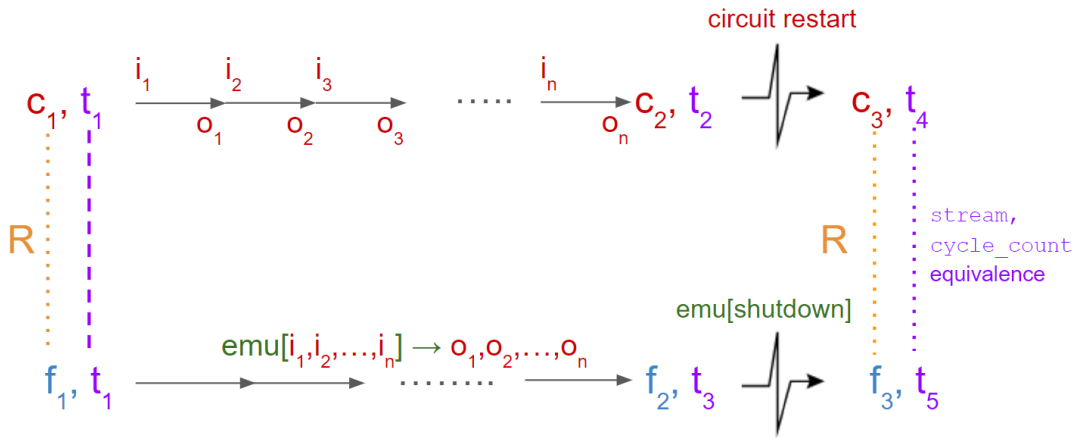

Figure 9.5: Physical equivalence for V-IPRR, which states that for all spec states f_1 and implementation states c_1 that are related by R, for all bit streams t_1 such that both the implementation's TRNG and the spec's TRNG have initial state t_1, and for all wire-level inputs $i_1, ..., i_n$: (a) the circuit outputs $o_1, ..., o_n$ match the emulator outputs (b) the final states f_3 and c_3 (f_2 after shutdown and c_2 after a reset respectively) are related by R (c) the final state of the circuit's TRNG t_4 and the final state of the spec's TRNG t_5 have the same `stream` and `cycle_count` states

9.1.3 Preliminary Work on Implementation

One major difference between V-IPRR and IPRR is that in V-IPRR, there is a nondeterministic delay before the next TRNG word is ready, while in IPRR the random bit is always outputted in the next cycle. For functional output and timing equivalence, we must make sure that the property holds for every possible delay. We have implemented this portion of the proof in order to test its functionality and performance.

We implement the nondeterministic delay in a similar way that Knox handles the `yield` call, by finding a fixed point [3]. A fixed point is defined as a set of symbolic terms such that the set is closed under the step function, and represents all possible circuit states after waiting for any number of cycles. We introduce the `wait-trng` hint, which, when called, creates a fixed point by setting the `delay` state to 1 at the start of every cycle and iteratively stepping the circuit, stopping when the next circuit state is already covered by a previous one. This models running the HSM for an arbitrary number of cycles while the TRNG is

not ready yet. When the fixed point is completed, we branch on every element in the fixed point and set `delay` to 0.

We also conduct preliminary tests on the `wait-trng` to analyze its performance. Since `wait-trng` creates branches every time we query for a new random word, this could lead to an exponential number of branches if we create two or more branches in every fixed point. However, through testing on the password-hasher HSM, we discover that with the proper use of hints, the `wait-trng` usually just creates a fixed point of size 1, since the circuit blocks and maintains the same state while it waits for the TRNG to respond. Therefore, we can conclude that for many implementations, resolving the nondeterminism of delay timings does not incur a significant cost on the proof. More analysis needs to be done on other components of the implementation, however.

9.1.4 Preliminary Evaluation

Although more work needs to be done on the proof strategy and the implementation, and likely changes will be made, we can conduct some preliminary analysis on the V-IPRR definition, assumptions we have made, and feasibility of development.

V-IPRR and IPRR provide different security guarantees, as V-IPRR allows leakage of the number of previous operations performed. For many HSMs, this leakage is acceptable; for instance in the password-hasher we care about maintaining security of the secret but not how many calls were made to the HSM. A natural question that follows is how we can modify V-IPRR to prevent leakage of the number of previous operations. This would likely involve changing the TRNG model or making additional assumptions, as under the current model a TRNG that has been queried has different state than the initial TRNG, and therefore all subsequent queries to the TRNG will leak information about previous operations.

A key assumption we make for V-IPRR is that all history is remembered by the TRNG. The rationale behind this assumption is to capture all possible internal behaviors of a TRNG. For instance, a TRNG that can store n random words would be able to "remember" the n

past queries (for example if we wait for a long time then perform $n + 1$ queries in rapid succession, we will get n fast responses and 1 slow response). However, the assumption may be too strict, as it would be reasonable to think that after a certain amount of time, the TRNG forgets past state. Our model currently has no way of representing forgetting history.

Another point of evaluation is on the feasibility of developing HSMs that satisfy V-IPRR. The HSMs would need to ensure that TRNG query timings, as well as overall runtime of each operation, is the same for all operations, so that no information is leaked to the TRNG. This may be difficult to do for HSMs that use CPUs. Furthermore, maintaining the property may decrease performance. For instance, the password-hasher would need to sample the TRNG in the `hash` function, even though there is no need for additional randomness.

9.2 Improvements to Karatroc

An area for future work is in improving the practicality of use and performance of Karatroc. One component in Karatroc proofs that leads to additional proof complexity and greater runtime is the need to branch many times to prevent the TRNG state from becoming a union data structure. This is because the current framework has trouble processing unions. When the representation becomes too complex and it is not immediately clear when something can go wrong, the framework will often emit assertions to be checked later (such as making sure we do not go out of bounds on a list), and many components of the base Knox framework do not support unions. However, if we could give Karatroc the ability to automatically simplify TRNG state when it becomes a union or to add logic for unions in currently unsupported areas, this could remove the need to create a lot of branches in the proof in many instances.

Another area for improvement in Karatroc is greater abstraction of the specification. The developer is currently required to manually specify how the TRNG state changes, and it would simplify the workflow if Karatroc provided abstractions such as a `get_rand_bit` function that automatically returns the next random bit and modifies the TRNG state.

Furthermore, the `discard` function must currently be contained within the specification file, but since every emulator should have access to `discard`, it would make more sense to incorporate it into the framework instead.

Chapter 10

Related Works

10.1 Information-Preserving Refinement

Our definition of IPRR extends the idea of *Information-Preserving Refinement* (IPR) [3]. IPR is a zero-knowledge style definition that captures the notion that an implementation correctly implements its specification and leaks no additional information. IPR asserts the indistinguishability of two worlds: a real world that is based on the HSM implementation, and an ideal world that is based on the specification and secure by construction.

10.2 Knox

Knox [3] is a framework that uses symbolic execution to verify that a HSM satisfies IPR. Knox does not support nondeterministic specifications, and thus cannot verify HSMs that use TRNGs. Karatroc extends Knox, enabling the verification of HSMs that use randomness. Key features that Karatroc provide include the representation of the TRNG model, additional verification checks for TRNG state when proving functional and physical equivalence, and new functions such as `shutdown` and `discard`.

10.3 Verification of Security with Randomness

This section describes previously used techniques to verify security in the context where nondeterminism and randomness is present.

10.3.1 Stepwise Refinement

Stepwise refinement is a method that involves starting from an abstract program and refining it step by step towards a concrete program. Prior work in preserving security in stepwise refinement has coined the idea of a "refinement paradox", which is the notion that a nondeterministic secure program can be refined into a deterministic but insecure program, due to the fact that many security properties are not preserved under refinement. For instance, a program that allows nondeterminism in a variable v could be refined by a program that sets v to be equal to a secret value. This problem is very similar to issues we considered when developing the TRNG model in Karatroc, where we needed to design our model such that such insecure deterministic programs are excluded in the definition of IPRR.

To handle the refinement paradox, previous works have developed a framework where secrecy is preserved under refinement [13] and created an operational model that alters refinement for sequential programs such that ignorance of secret variables is maintained [14].

10.3.2 Restricting Nondeterminism

Another way to handle nondeterminism when verifying security is to restrict the nondeterminism allowed in specifications. For instance, in verifying seL4, to prevent nondeterminism from revealing secrets, Klein et al. [15] remove user-visible nondeterminism by replacing nondeterministic operations in the specification with placeholders derived from deterministic implementations. Constanzo et al. [16] use a similar strategy, disallowing specifications from exhibiting visible nondeterminism. However, this is not a valid strategy in the setting of IPRR, as we want to permit and verify nondeterministic specifications.

10.3.3 Factoring Out Nondeterminism

The work of Ileri et al. [17] describes modeling randomness in a similar setting. The paper presents DiskSec, an approach for verifying confidentiality for file systems, which requires that the adversary cannot observe differences correlated with confidential data. Because many specifications of file systems can be nondeterministic, DiskSec also faces the challenge of verifying security without leaking information through nondeterminism (for instance through nondeterministic ordering of files). DiskSec handles this by factoring out an oracle that supplies all nondeterminism. Then, DiskSec requires that the required equivalence holds for all choices of the oracle. DiskSec's approach regarding randomness is similar to our strategy, by factoring out nondeterminism and proving that our security and correctness properties hold for all values of the random variables.

Ironclad [18] is another system that provides verified security by factoring out randomness. Ironclad provides access to secure randomness via a Trusted Platform Module (TPM), and applications that correctly use randomness from the TPM can be verified as secure. The TPM has many similarities to Karatroc's TRNG, both being trusted modules that provide access to a stream of secure random numbers.

10.3.4 Probabilistic and Statistical Verification

Other works related to verifying programs with randomness include the work of Susag et al. [19] which proves probabilistic properties of a program. They do this by introducing a class of probabilistic symbolic variables and computing symbolic branch probabilities. In addition, the work of Eldib et al. [20] uses statistical analysis to verify the security of masking countermeasures against power analysis based side channel attacks. In their paper, they introduce the concept of perfect masking, which is the idea that intermediate results are statistically independent of the sensitive data. Although these papers take approaches that are rather different from that of Karatroc, since we does not currently support probabilistic

specifications nor statistical analysis of distributions, they could be interesting directions to investigate as future work.

Chapter 11

Conclusion

This thesis presents IPRR, a zero-knowledge style definition that captures the idea that a HSM that uses a TRNG correctly implements its specification and leaks no additional information. This thesis also develops a proof strategy for verifying that a HSM satisfies IPRR, and Karatroc, a tool that implements the proof strategy. Through evaluation of Karatroc on sample HSMs, we are able to conclude that proofs written with Karatroc perform similarly to and have similar complexity as those written with existing verification tools. We furthermore present a definition of V-IPRR, which adapts IPRR to TRNGs with a valid/ready interface, and nondeterministic and history dependent delays.

Appendix A

Random Byte Generator Specification

```
#lang knox/spec

#:init s0
#:symbolic-constructor new-symbolic-state
#:methods
(get-random)
#:leak discard
#:random #t
#:max-trng-bits 10

(provide discard)

;; stateless
(define (new-symbolic-state)
  (void))

(define s0 (void))
```

```
(define (make-word t n)
  (if (equal? n 1)
    (bool->bitvector (car t) 1)
    (concat (bool->bitvector (car t) 1) (make-word (cdr t) (- n
       1)))))

(define (update-trng t n)
  (if (equal? n 0)
    t
    (update-trng (cdr t) (- n 1))))

(define ((get-random) s)
  (define t (rstate-trng s))
  (result (make-word t 8) (rstate (rstate-spec s) (update-trng t
     8))))

(define ((discard n) s)
  (define t (rstate-trng s))
  (result #f (rstate (rstate-spec s) (update-trng t n))))
```

Appendix B

Random Byte Generator Implementation Code

```
module rng#(
    parameter WIDTH = 8
)(
        input clk,
        input reset,
        input en,

        input trng_bit,
        output trng_next,

        input req,
        output [WIDTH-1:0] random_word,
        output output_valid,
);

reg [WIDTH-1:0] cur_word;
```

```
reg [5:0] cur_bit_ind;
reg valid;
reg want_next;
reg reset_ind;
always @(posedge clk) begin
    if (reset) begin
        cur_word <= 0;
        cur_bit_ind <= 0;
        valid <= 0;
        want_next <= 0;
        reset_ind <= 0;
    end
    else if (en) begin
        if (cur_bit_ind <= WIDTH-1) begin
                cur_word <= (cur_word<<1) + trng_bit;
                cur_bit_ind <= cur_bit_ind + 1;
        end
        if (reset_ind) begin
                cur_bit_ind <= 0;
                valid <= 0;
                want_next <= 0;
                cur_word <= 0;
                reset_ind <= 0;
            end else if (req && cur_bit_ind >= WIDTH-1) begin
                valid <= 1;
                reset_ind <= 1;
                want_next <= 1;
            end else if (cur_bit_ind >= WIDTH-1) begin
                valid <= 0;
```

```
                want_next <= 0;
            end else begin
                want_next <= 1;
                valid <= 0;
            end
    end
end
assign trng_next = en ? want_next:0;
assign output_valid = en ? valid:0;
assign random_word = en ? (valid ? cur_word:0) : 0;

endmodule
```

Appendix C

Password-Hasher Specification

```
#lang knox/spec

#:init s0
#:symbolic-constructor new-symbolic-state
#:methods
(set-secret)
(get-hash [msg (bitvector MESSAGE-SIZE)])
#:leak discard
#:random #t
#:max-trng-bits 161

(require rosutil
         (only-in "spec-sha256.rkt" sha256))

(provide
 SECRET-SIZE
 SECRET-SIZE-BYTES
 MESSAGE-SIZE
```

```
 MESSAGE-SIZE-BYTES
 new-symbolic-state
 set-secret
 get-hash
 s0
 discard)

(define SECRET-SIZE-BYTES 20)
(define SECRET-SIZE (* 8 SECRET-SIZE-BYTES))
(define MESSAGE-SIZE-BYTES 32)
(define MESSAGE-SIZE (* 8 MESSAGE-SIZE-BYTES))

(define (new-symbolic-state)
  (fresh-symbolic 'secret (bitvector SECRET-SIZE)))

(define (make-word t n)
  (if (equal? n 1)
    (bool->bitvector (car t) 1)
    (concat (bool->bitvector (car t) 1) (make-word (cdr t) (- n
       1)))))

(define (update-trng t n)
  (if (equal? n 0)
    t
    (update-trng (cdr t) (- n 1))))

(define ((set-secret) s)
  (define t (rstate-trng s))
  (define secr (make-word t SECRET-SIZE))
```

```
  (define new-t (update-trng t SECRET-SIZE))
  (result #t (rstate secr new-t)))

(define ((get-hash msg) s)
  (define secr (rstate-spec s))
  (result (sha256 (concat secr msg)) s))

(define ((discard n) s)
  (define t (rstate-trng s))
  (result #f (rstate (rstate-spec s) (update-trng t n))))

(define s0 (bv 0 SECRET-SIZE))
```

Appendix D

Password-Hasher C Code

```
#include "drivers.h"
#include <stdbool.h>

#define BAUD_RATE 6
#define SECRET_SIZE 20
#define MESSAGE_SIZE 32

struct state {
    uint32_t active;
    uint8_t secret0[SECRET_SIZE];
    uint8_t secret1[SECRET_SIZE];
} *state = (struct state *) FRAM_BASE;

#define CMD_SET_SECRET (0x01)
#define CMD_GET_HASH (0x02)

void do_set_secret();
void do_get_hash();

void main() {
    uart_init(UART1, BAUD_RATE);
```

```
    uint8_t cmd = uart_read(UART1);
    switch (cmd) {
        case CMD_SET_SECRET:
            do_set_secret();
            break;
        case CMD_GET_HASH:
            do_get_hash();
            break;
        default:
            break;
    }
    poweroff(); // done, don't interact with world any more until next
       reset
}

void memcpy(uint8_t *dest, const uint8_t *src, size_t sz) {
    for (int i = 0; i < sz; i++) {
        dest[i] = src[i];
    }
}

// written this way, it turns into branch-free code (= constant time)
uint8_t *get_active() {
    uint8_t *base = state->secret0;
    uint32_t zlt = 0 < state->active;
    zlt = zlt * SECRET_SIZE;
    base = base + zlt;
    return base;
}

// written this way, it turns into branch-free code (= constant time)
uint8_t *get_inactive() {
    uint8_t *base = state->secret0;
```

```
    uint32_t zlt = 0 < state->active;
    zlt = 1 - zlt;
    zlt = zlt * SECRET_SIZE;
    base = base + zlt;
    return base;
}

void do_set_secret() {
    uint8_t secret[SECRET_SIZE];
    for (int i=0; i<SECRET_SIZE; i++) {
        secret[i]=trng_read();
    }
    // write into inactive region, then make active
    uint8_t *dest = get_inactive();
    memcpy(dest, secret, SECRET_SIZE);
    state->active = !state->active;
    uart_write(UART1, 0x01);
}

uint8_t digest[SHA256_DIGEST_SIZE];

void do_get_hash() {
    uint8_t buf[SECRET_SIZE + MESSAGE_SIZE];
    for (int i = 0; i < MESSAGE_SIZE; i++) {
        buf[SECRET_SIZE + i] = uart_read(UART1);
    }
    uint8_t *secret = get_active();
    memcpy(buf, secret, SECRET_SIZE);
    sha256_digest(SHA256, buf, SECRET_SIZE + MESSAGE_SIZE, digest);
    for (int i = 0; i < sizeof(digest); i++) {
        uart_write(UART1, digest[i]);
    }
}
```

References

[1] Google, *Google Meet hardware secure module*, 2024. URL: https://support.google.com/a/answer/10847090?hl=en.

[2] Apple, *ICloud Security Overview*, 2024. URL: https://support.apple.com/guide/security/icloud-security-overview-secacde2d0da/web.

[3] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying hardware security modules with Information-Preserving refinement," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 503–519, ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/athalye.

[4] D. Lubicz and N. Bochard, "Towards an oscillator based trng with a certified entropy rate," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 1191–1200, 2014.

[5] Thales, *Thales Luna Network HSM*, May 2021. URL: https://cpl.thalesgroup.com/sites/default/files/content/product_briefs/field_document/2021-07/thales_luna_network_7_hsm_pb.pdf.

[6] Yubico, *Secure, low cost hardware protection for cryptographic keys*, 2024. URL: https://resources.yubico.com/53ZDUYE6/at/937nrcp925s6jhnxktpzhnfh/YubiHSM_2_Technical_Data_Sheet.pdf?format=pdf.

[7] B. Ray and A. Milenković, "True random number generation using read noise of flash memory cells," *IEEE transactions on electron devices*, vol. 65, no. 3, pp. 963–969, 2018.

[8] Berten Secure Communications, *True Random Number Generators (TRNG)*, Mar. 2024. URL: https://www.bertendsp.com/products/trng-p200/#1478026309160-d1160395-223f.

[9] TectroLabs, *MicroRNG Datasheet*, Mar. 2024. URL: https://tectrolabs.com/assets/documents/microrng-datasheet.pdf.

[10] FDK Corporation, *True Random Number Generation IC RPG100 / RPG100F*, May 2011. URL: https://www.fdk.com/cyber-e/pdf/HM-RAE001.pdf.

[11] S. Nolting, *The neoTRNG V3 - A Tiny and Platform-Independent True Random Number Generator*. Version 3.0.0, Nov. 2023. DOI: 10.5281/zenodo.10071201. URL: https://github.com/stnolting/neoTRNG.

[12] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 530–541, 2014.

[13] J. Jürjens, "Secrecy-preserving refinement," in *International Symposium of Formal Methods Europe*, Springer, 2001, pp. 135–152.

[14] C. Morgan, "The shadow knows: Refinement and security in sequential programs," *Science of Computer Programming*, vol. 74, no. 8, pp. 629–653, 2009, Special Issue on Mathematics of Program Construction (MPC 2006), ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0167642309000264.

[15] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, pp. 1–70, 2014.

[16] D. Costanzo, Z. Shao, and R. Gu, "End-to-end verification of information-flow security for c and assembly programs," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 648–664, 2016.

[17] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich, "Proving confidentiality in a file system using {disksec}," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 323–338.

[18] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 165–181.

[19] Z. Susag, S. Lahiri, J. Hsu, and S. Roy, "Symbolic execution for randomized programs," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1583–1612, 2022.

[20] H. Eldib, C. Wang, and P. Schaumont, "Formal verification of software countermeasures against side-channel attacks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–24, 2014.