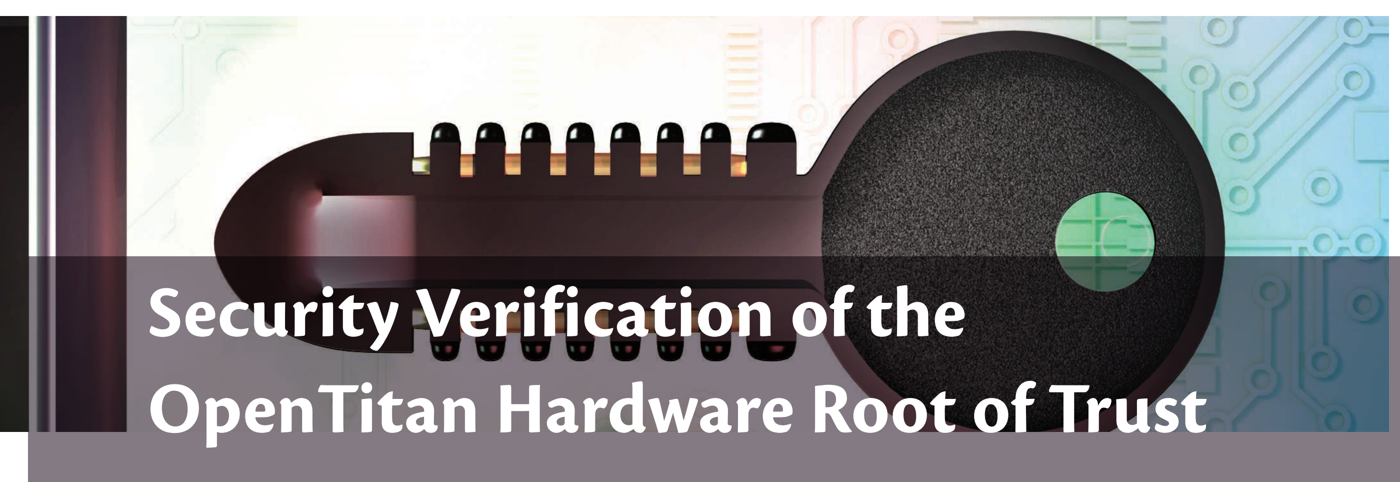


Security Verification of the OpenTitan Hardware Root of Trust

Andres Meza and **Francesco Restuccia** | University of California San Diego
Jason Oberg | Cycuity
Dominic Rizzo | OpenTitan
Ryan Kastner | University of California San Diego

We describe the security verification of OpenTitan. We illustrate how information flow tracking turns human knowledge of assets and security requirements into formal security properties verified using Cycuity's Radix. The verification uncovered weaknesses and helped produce hardware fixes to eliminate vulnerabilities.

OpenTitan is a commercial-grade, open-source hardware root of trust (RoT). RoTs perform security-critical functionalities such as secure boot, the configuration of operation modes (e.g., debug versus normal), and management of sensitive data (e.g., cryptographic keys). OpenTitan is targeted for use by enterprises, platform providers, and chip manufacturers as a platform integrity module, universal second-factor security key, and trusted platform module. The OpenTitan includes a security-enhanced RV32IMCB RISC-V Ibex core, various security peripherals (e.g., Advanced Encryption Standard, Keccak Message Authentication Code, Hash-based Message Authentication Code), multiple memories [e.g., ROM, FLASH, static random-access memory (SRAM), one-time programmable (OTP)] with dedicated controllers for access control and scrambling purposes, and different input–output peripherals.

OpenTitan has well-documented security requirements and verification procedures. The OpenTitan threat model describes the security assets, potential attacker profiles, attack surfaces, and methods. The threat model is used to derive relevant security requirements. OpenTitan defines a security model specification that includes device provisioning and run-time operations, secure hardware design guidelines, and functional guarantees. OpenTitan includes testing plans, testbench architecture, a security countermeasure verification framework, design guides, and integrates with formal and simulation-based verification tools.

This article steps through the security verification process for the OpenTitan OTP controller. The OTP holds secret data used for secure boot, lifecycle provisioning, and attestation. Thus, it plays a key role in the overall security of OpenTitan. We describe an important security asset, derive requirements for that asset and the OTP operation, and write formal properties that specify the requirements. The goal is to give the

Digital Object Identifier 10.1109/MSEC.2023.3251954
Date of current version: 20 April 2023

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

reader an understanding of the state of the art in silicon security verification.

Our security verification process uses information flow tracking (IFT) to perform hardware security verification. Using IFT, we find a potential hardware weakness, localize the weakness, propose a hardware patch, and verify the patch is secure. IFT enables verification engineers to reason about noninterference[1] expressed as a hyperproperty.[2] Hyperproperties provide a more concise and expressive representation of confidentiality, integrity, and availability, which is crucial for hardware security verification.[3] Commercial hardware IFT techniques have emerged as a critical tool for hardware security.

We use Cycuity's Radix to formalize and verify the security requirements. Radix is an IFT-enhanced simulation tool that allows designers to express and verify security properties easily. The verification engineer must identify critical design assets and formalize the security requirements for those design assets using security properties. Those properties are provided to IFT security verification tools, which uncover security property violations. We describe how to assess the severity of these weaknesses and determine how to repair the security weakness to eliminate unnecessary confidential information leakage. This hardware patch was integrated into the OpenTitan design.

The contributions of the article include the following:

- describing the state of the art in hardware security verification using open-source OpenTitan hardware root of trust
- demonstrating the value and effectiveness of hardware IFT as a verification approach to formalize the security property, identify a potential weakness, debug the root cause, and repair the flaw
- uncovering a weakness in the OpenTitan OTP memory controller, providing a patch to fix the OpenTitan, and submitting a common weakness enumeration (CWE) around the weakness.

OpenTitan OTP Memory Controller

The OTP memory controller is a peripheral on the chip interconnect bus which manages the OpenTitan's OTP memory. The OTP data are nonvolatile and irreversible, and includes information critical to secure system operation like device calibration settings, hardware configuration data, test and unlock tokens, and root keys. The controller provides access control to the OTP memory. Secret data are not readable by software once it is locked and is scrambled in storage. Data can be set to be read and write lockable and undergo integrity and storage consistency checks. The OTP controller is crucial for the correct and secure operation of OpenTitan.

Figure 1 describes the OTP controller architecture. It contains eight OTP partitions (`P0`-`P7`) that hold data from the OTP memory. Partitions P0, P1, and P2 are unbuffered, i.e., they do not store data; data are retrieved from the OTP memory on every access request. Partitions P3–P7 are buffered. The data in the buffered partitions are retrieved upon boot from the OTP memory and stored locally in the OTP controller after that.

Each of these partitions contains unique data with different access control requirements. The data in partitions P4, P5, P6 are SECRET. SECRET data are stored encrypted (scrambled) in the OTP memory. Data can be read and write locked from software access statically or at run-time. Locked data are stored with a digest for integrity checks. Buffered data are stored with error correction control protection also used for integrity checks.

The OTP scrambling datapath performs lightweight scrambling operations as requested by the partitions and its different interfaces. The OTP scrambling datapath uses the lightweight 64-bit PRESENT block cipher. Secret data stored in the OTP memory are scrambled to protect against physical attacks. A global netlist constant is used as the key, which is set at hardware design time (premanufacturing). The scrambling datapath also computes lightweight ephemeral key derivation function for RAM and FLASH scrambling mechanisms.

The interfaces manage the interactions between OTP memory, the buffers, and other OpenTitan hardware modules. The register interface enables software to interact with the underlying OTP block. The direct-access interface facilitates accessing and programming the contents of the OTP memory. The life cycle interface (LCI) allows the OpenTitan's life cycle controller to update the life cycle state stored in `P7` once per power cycle. The key derivation interface (KDI) enables OpenTitan's key manager to interact with the scrambling datapath and partition P5 that holds the scrambling root keys used to derive static and ephemeral scrambling keys for the OpenTitan FLASH and SRAM memories.

Security Verification

Hardware security verification can be broken down into six steps.[4]

1. Create the threat model.
2. Identify the assets.
3. Determine security weaknesses for the assets.
4. Define the security requirements based on Steps 1, 2, and 3.
5. Specify security properties.
6. Verify the security properties.

The first five steps of this process are still largely manual. They rely heavily on verification engineers to describe the threat model, explicitly define the assets, develop the weakness and requirements, and finally specify the properties. Once the properties are specified, automated verification tools can take the properties and assess whether the hardware design adheres to them. Verification engineers often iterate these steps to refine the properties to more precisely define the security properties and provide adequate security coverage.

Now we use this six-step process to verify the confidentiality of the key to encrypt secret data stored in the OTP memory. We discussed how hardware IFT verification is crucial to assess potential security vulnerabilities. Our analysis uncovered a weakness in the OTP memory controller. We describe how to assess the severity of the weaknesses and determine an appropriate hardware patch. This patch was integrated into the OpenTitan design.

Step 1: Create the Threat Model

The OTP controller specification states that its primary purpose is to provide "high-level logical security protection, such as integrity checks and

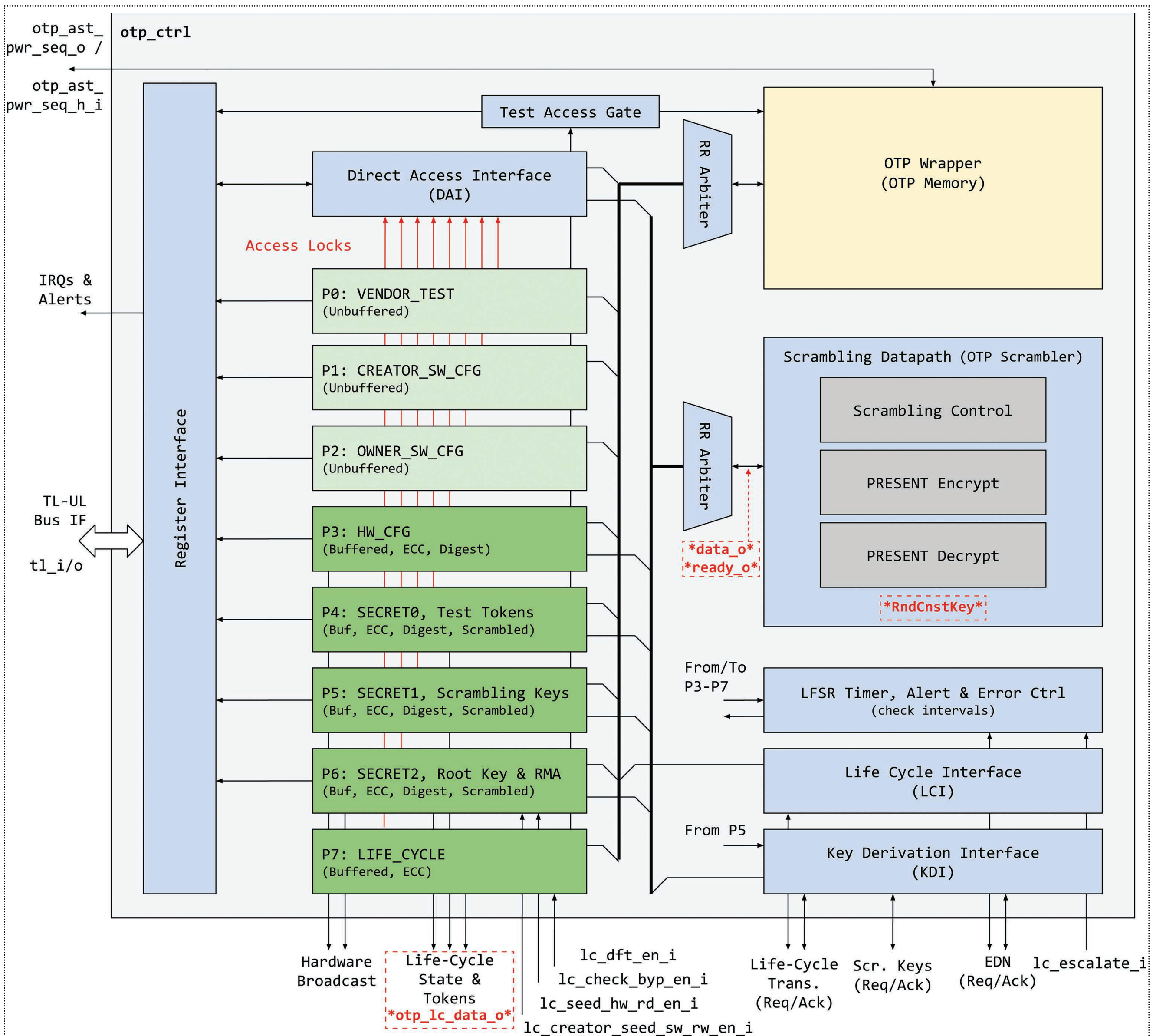


Figure 1. Block diagram of the OpenTitan's OTP Memory Controller. The OTP controller provides access control to different interfaces. Data can be marked as secret (stored encrypted), locked, and integrity checked. The scrambling datapath provides lightweight encryption operations to assist in different aspects of the access control. LFSR: linear feedback shift register.

scrambling of sensitive content." Secret data are stored encrypted in the OTP memory to help secure it against physical attacks, e.g., to protect the data from readout, fault injection, and probing. OpenTitan encrypts secret data using a netlist constant key that is fixed in the hardware at design time. Our security verification focuses on the confidentiality of this scrambling key.

Step 2: Identify Assets

`RndCnstKey` is the security asset under verification. `RndCnstKey` is a global netlist constant in the OTP scrambler used as a key to protect secret data stored in the OTP memory. Secret data includes the test tokens, scrambling keys, and root key and return material authorization tokens stored in partitions P4, P5, and P6 (see Figure 1). Scrambled data moves to many locations within the OTP controller including the partitions, scrambling datapath, LCI, KDI, and register interface. Thus, there are many ways that `RndCnstKey` could inadvertently leak outside of the OTP controller.

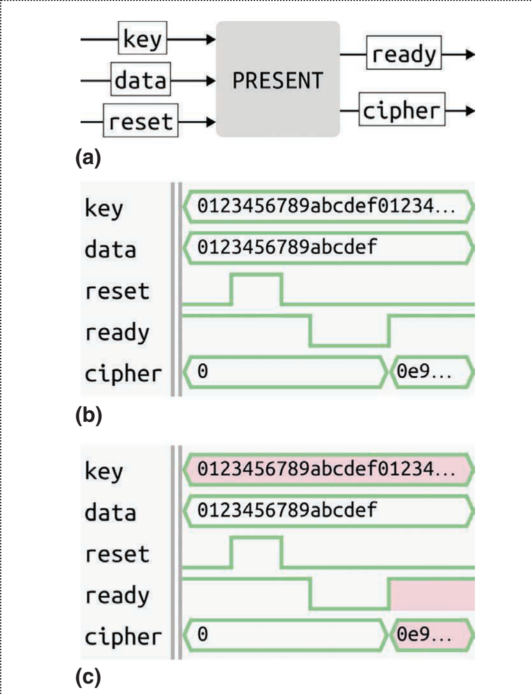


Figure 2. (a) A simplified block diagram of the PRESENT block cipher. (b) Waveform showing a single *functional* execution trace of the PRESENT block cipher module. (c) Waveform showing a single *IFT-enhanced* execution trace of the PRESENT block cipher module with red shading to indicate that a particular signal contains information (i.e., at least one of its security labels is `HIGH`) which originated from `key`.

> **Security Asset:** `RndCnstKey`

There are many other assets in the OpenTitan OTP controller. While we focus on security verification of `RndCnstKey`, the general methodology applies to other assets in the OpenTitan design.

Step 3: Determine Security Weaknesses

It is critical for `RndCnstKey` to remain confidential; access to `RndCnstKey` would enable an adversary to decrypt sensitive data from the OTP memory. Any information related to the `RndCnstKey` should remain within the OTP controller; no knowledge about `RndCnstKey` should leak outside the controller. Based on this, we can specify the security objective and security boundary for `RndCnstKey`.

> **Security Objective:** Confidentiality

> **Security Boundary:** All outputs of the OTP controller

Step 4: Define Security Requirements

Using the asset, objective, and boundary, we can specify the following plain-language security requirement:

> **Security Requirement:** Any information related to `RndCnstKey` should not be visible on the outputs of the OTP controller.

The "should not be visible" portion of the requirement comes about from the security objective of confidentiality. Similarly, the "on the outputs of the OTP controller" comes from the security boundary. We want to verify that `RndCnstKey` data stays within the OTP controller.

Step 5: Specify Security Properties

Now, we convert our plain-language security requirement into a formally specified security property. A property should precisely state the security requirements in a manner that can be analyzed by verification tools.

Verification involves writing properties about behaviors and using tools to assess if the hardware upholds those properties. Verification uses assertion languages to write statements about the behaviors. It allows the specification of temporal behaviors including event sequences, latencies, and pipelines. System Verilog Assertions and Property Specification Language are common languages for hardware verification.

Consider the PRESENT block cipher used in the OTP Scrambler. Figure 2 shows a simplified version

of PRESENT, which takes as input a `key` and `data` to scramble and outputs the scrambled data `cipher`. The `ready` signal indicates that the cipher data are valid. `reset` reinitializes the datapath. Figure 2(b) provides an example execution or `trace`.

Functional verification properties are typically expressed as *trace properties*, which describe behaviors over a single execution trace. Consider the trace property defined for the PRESENT block cipher module:

```
(reset != 1) || (ready! = 1)
```

The trace property specifies that if `reset` is 1 then `ready` is 0. The property can also be specified using the implication operator: |–>

```
(reset == 1) |–> (ready != 1)
```

A counterexample for a trace property can be described by a single trace of execution. Figure 2(b) provides a counterexample trace where `reset == 1` at the same time that `ready == 1`.

Trace properties are valuable for security verification, but they have limited expressiveness for many security-related properties. For example, confidentiality properties state that no information about some data (e.g., `RndCnstKey`) can ever be leaked or inferred at another location. Or, more generally, information should not flow from a source to a destination. The dual of this is that the source data should never be able to affect the sink data, which expresses properties related to data integrity. Confidentiality and integrity properties cannot be easily specified using a trace property. They require a more expressive property language.

Hardware IFT is a security verification technique that monitors how information from some source propagates throughout the hardware.[3] Hardware IFT adds security labels that indicate where information propagates and tracks how their information moves as the hardware executes. Hardware IFT enables designers to analyze the security of their design more efficiently. Verification engineers can learn where and how asset information travels throughout the hardware.

The key aspect of IFT properties is specifying the notion of information flows (or lack thereof). We adopt the notation using the no-flow operator =/=>, as used by Cycuity's Radix software, to indicate noninterference between the source signal and the destination signal. For example, we may want to assert that information from the `key` can never be inferred by observing the `ready` output signal.

```
//IFT Property
key =/=> ready
```

In other words, this IFT property states that no information from signal `key` should ever be deducible by viewing the signal `ready`. Any change in `key` should never affect the `ready` signal; they should operate independently. If there was a flow, then the attacker could learn information about the key by observing the ready signal, which would indicate a timing side channel in the PRESENT scrambler.

IFT properties are a type of hyperproperty that expresses noninterference behaviors specified over a set of execution traces.[2] Hyperproperties are fundamentally more expressive than trace properties. A counterexample for an IFT property requires more than one trace to describe an interfering behavior. A counterexample for `key =/=> ready` hyperproperty requires at least two traces with differing values of `key` which show an effect on the `ready` value.

The Cycuity Radix tools use IFT analysis for security verification. Radix takes as input IFT properties (also called *Radix rules*) that articulate behaviors related to the security requirements. These properties generally take the form of:

```
//Radix Security Rule/IFT Property
{src_signal_set} =/=> {dest_signal_set}
```

The property fails if any information from the `src_signal_set` flows to the `dest_signal_set`. Various additional qualifiers exist, e.g., qualifiers to specify when the source data should be tracked and conditions under which data flows to the destination are allowable. Some of these qualifiers will become clearer when demonstrated later on in this article.

Radix translates the IFT property into an information flow security monitor using the design register transfer level code. Radix then tracks information flows over time allowing the verification engineer to uncover potential weakness, localize sources of the vulnerability, and develop hardware patches that eliminate the weakness.

IFT-enhanced traces are more powerful than functional traces because they provide additional security properties to support noninterference using `HIGH` and `LOW` labels.[5] If a security label of a signal in an IFT-enhanced trace is `HIGH`, then it contains information from an asset defined in the `src_signal_set`

(that was initially marked as HIGH), i.e., any signal whose label is HIGH contains information about the source assets. Thus, one IFT-enhanced trace is sufficient in determining noninterference—this exemplifies the power and value of hardware IFT.

Figure 2(c) depicts a single IFT-enhanced trace that provides a counterexample to the `key =/=> ready` property. The `key` always has a HIGH label (depicted with red shading) as this is the information that property expresses to track. Later in the trace the `ready` and `cipher` signals are marked as HIGH, which indicates information about the `key` was transferred into those signals at that time, thus providing a counterexample to the trace.

Developing IFT properties is straightforward given a security requirement, objective, and boundary. The following IFT property shows the IFT property for the security requirement related to `RndCnstKey`:

```
assert iflow(
u_otp_ctrl_scrmbl.rnd_cnst_key_anchor
=/=>
$all_outputs
);
```

Since the security objective for `RndCnstKey`'s requirement is confidentiality, we make `RndCnstKey` the source signal for the IFT property by placing its corresponding design signal (`rnd_cnst_key_anchor`) on the left-hand side of the no-flow operator (`=/=>`). Similarly, we make the destination signals of the IFT property all outputs of the OTP controller (specified using Cycuity's `$all_outputs` shorthand) based on the requirement's associated security boundary. This property will fail if any information from `RndCnstKey` flows to any of the outputs of the OTP controller. It should be noted that this information is tracked through logical and sequential transformations and is independent of the value of `RndCnstKey`.

Step 6: Verify Security Properties

Now that we have formally specified a security property, we can verify whether this property holds for the OTP controller. This verification is performed via the functional simulation of OpenTitan alongside Radix. Radix automatically generates the security monitor, which precisely tracks information flows in the OpenTitan design. The security property provides an initial labeling of the `RndCnstKey` asset, i.e., setting its security label equal to HIGH. During simulation, Radix reports if/when the associated security property is violated by checking if the OTP output labels are set as HIGH.

Radix translates the security properties into security monitors. The security monitor can then be executed in simulation or emulation alongside the original design RTL. This process is shown in Figure 3. Radix can be run with any functional testbench. However,

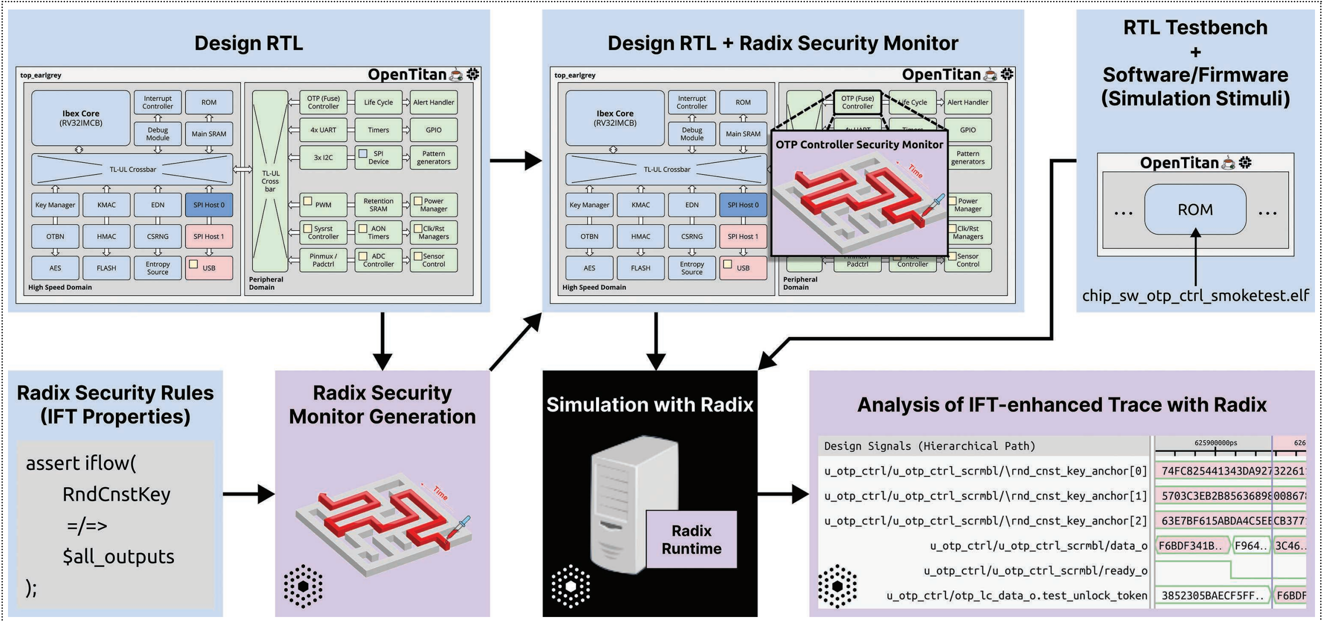


Figure 3. An overview of the Radix workflow. Radix security rules are combined with the design RTL to create a set of information flow security monitors. These security monitors are then inserted into the existing semiconductor simulation (shown) or hardware emulation (not shown) environments for execution.

certain testbenches will be more helpful in verifying a particular property than others due to how they stimulate the target design. We used the simulations specified in the chip_sw_otp_ctrl_smoketest—an OpenTitan's testbench explicitly designed for testing the OTP controller.

Figure 4 shows an IFT-enhanced trace where rnd_cnst_key_anchor is a source asset, i.e., those security labels are initialized as HIGH. We aim to understand where the RndCnstKey information flows. Any register with a HIGH label is shaded red. Indicated by the red shading on otp_lc_data_o.test_unlock_token, information from RndCnstKey leaks outside of the OTP controller via otp_lc_data_o.test_unlock_token which means that the specified security property does not hold for OpenTitan's OTP controller. Figure 5 shows the hierarchical path through which this information leakage occurs.

We found a violation of the security property. Now we must determine the extent of these weaknesses. IFT tools can help guide this debugging process.

Analyzing a Falsified Security Property

There are generally two approaches to consider when a property has been violated. The first approach assumes that the security requirement and properties are correct and attempts to find an error in the design's implementation by reviewing its source RTL code. The second approach attempts to determine if the security requirement was incorrectly specified.

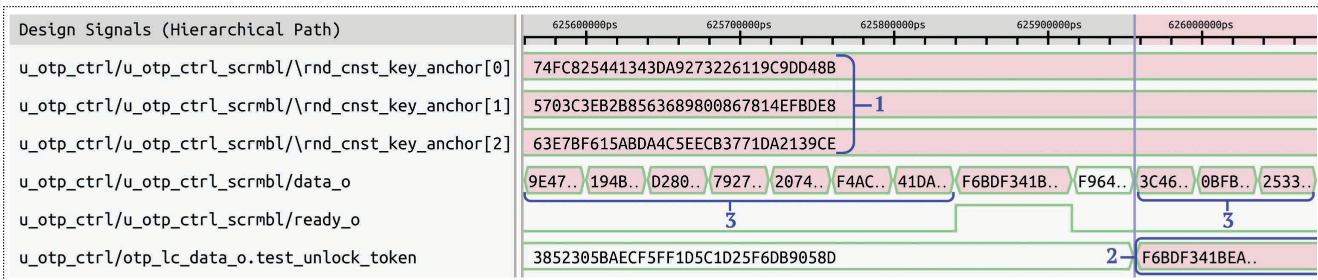


Figure 4. The first three waveforms (indicated in 1) correspond to RndCnstKey. Red indicates that the register's security label contains information from RndCnstKey. Information related to RndCnstKey leaks to the output of the OTP controller (test_unlock_token) as indicated in 2. The number 3 shows unintended leakage of RndCnstKey via the OTP scrambler's output data_o.

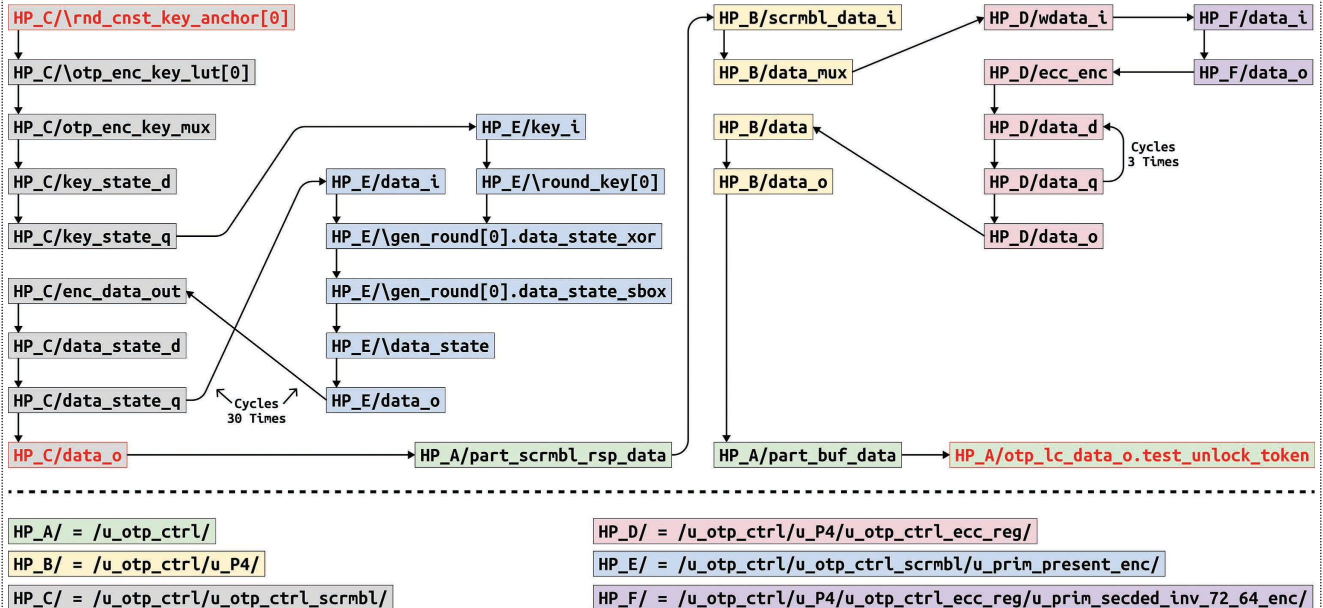


Figure 5. The hierarchical path through which information related to RndCnstKey leaks to the output of the OTP controller (test_unlock_token). Each node (i.e., colored rectangle) in the figure represents a design component (e.g., register, wire, and so on) in the OTP controller or one of its submodules. Each edge (i.e., arrow) indicates the flow of information from one design component to another.

In this case, the property is too restrictive. The property fails to account for the fact that encrypted data will hold information about the key, but it is mathematically secure. That is, due to the nature of the one-way encryption function, no information about the key is inferable from the output, even though the output depends on the marked asset (the key). It is important to note that this property is only true when the module implementing the PRESENT cipher outputs fully encrypted data; outputting intermediate cryptographic state/results during the encryption process invalidates the assumption that the key is protected by the one-way encryption function. A refined version of the RndCnstKey property, which performs an explicit downgrade of information from the output of the PRESENT cipher, is as follows:

```
assert iflow(
u_otp_ctrl_scrmbl.rnd_cnst_key_anchor
=/=>
$all_outputs
ignoring
u_otp_ctrl_scrmbl.data_o
when (u_otp_ctrl_scrmbl.ready_o == 1)
);
```

The base property remains the same—information from RndCnstKey should not flow to any of the outputs of the OTP controller. However, we now have an additional clause that explicitly ignores any information flows through the OTP scrambler's output (i.e., data_o) when the encryption operation is complete (i.e., when ready_o == 1. This is known as a label downgrade or declassification.[6] The information contained in data_o when ready_o is 1 is fully encrypted data. It is ok for this fully encrypted information to propagate outside of the OTP controller. Since RndCnstKey encrypts data within the OTP controller and only fully encrypted data are sent outside of the controller, the declassification of these flows using the ignoring clause is allowable. We pass this updated security property to Radix-S and perform security verification.

Figure 6 shows the simulation with this refined security property rnd_cnst_key_anchor. The difference between this result and the previous one is that the information leakage from RndCnstKey to the output of the OTP controller (otp_lc_data_o.test_unlock_token) no longer causes a property failure because it has been marked as an allowable flow. If this result still had the same property failure, that would indicate that the flow we saw in the initial result did not travel through data_o when ready_o was 1. However, since the addition of the ignoring clause removed the property failure, we know that the flow we previously saw was indeed due to the movement of the fully encrypted data.

The refined security property validates that OpenTitan does indeed prevent RndCnstKey from reaching the output of the OTP controller. However, both simulation results contain an unexpected weakness that could potentially jeopardize the confidentiality of RndCnstKey. There is an information flow during the intermediate results of the encryption operation.

Fixing Intermediate Leakage of RndCnstKey

Figures 4 and 6 show that the output of the OTP scrambler (data_o) exposes intermediate encryption results (when ready_o == 0) which contains information related to RndCnstKey. Additionally, the OTP scrambler exposes the input to the scrambler via data_o (see Figures 4 and 6). Although the modules which sample data_o only do so when ready_o is 1, a fault, bug, or attack on the sampling logic of any of the connected modules could lead to this intermediate state being sampled which poses a threat to the confidentiality of RndCnstKey and secrets encrypted in the OTP memory with this key.

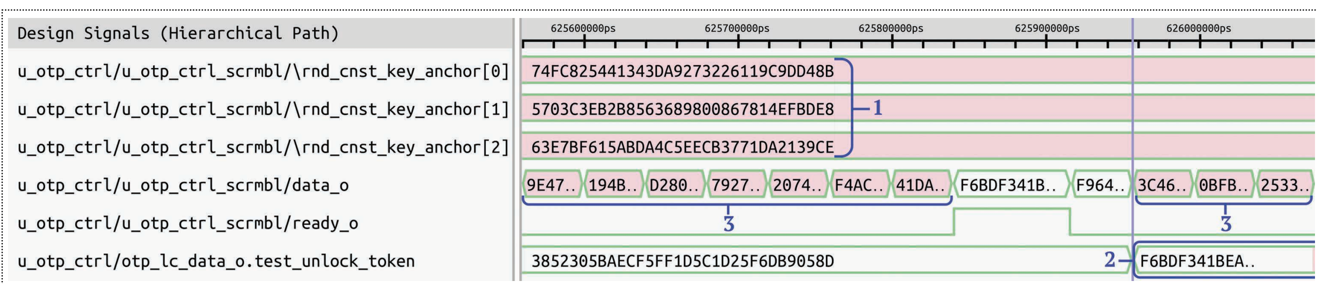


Figure 6. The first three waveforms (indicated in 1) correspond to RndCnstKey. Red indicates that the register's security label contains information from RndCnstKey. Unlike the trace shown in Figure 4, information related to RndCnstKey *does not* leak to the output of the OTP controller (test_unlock_token) as indicated by the lack of red in 2. The number 3 shows unintended leakage of RndCnstKey via the OTP scrambler's output data_o.

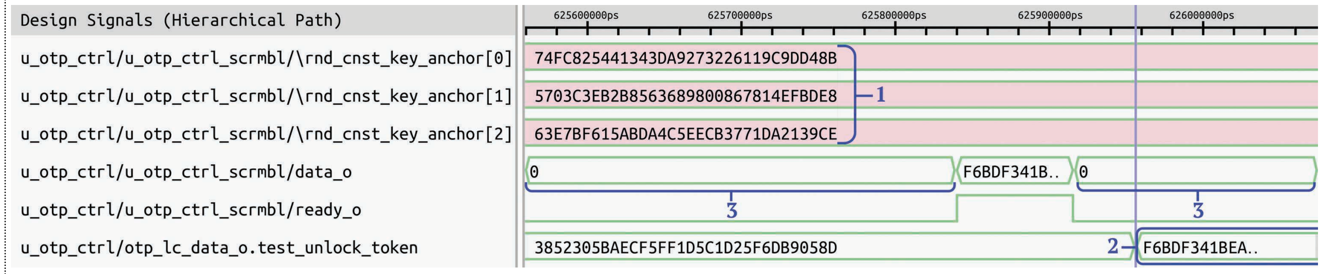


Figure 7. The first three waveforms correspond to RndCnstKey. Red indicates that the register's security label contains information from RndCnstKey. The number 3 shows that the unintended leakage of RndCnstKey has been fixed by our proposed solution. Instead of driving intermediate cryptographic state to the OTP scrambler's output (data_o), our solution drives a safe default value which carries no information from RndCnstKey.

We address this leakage with a simple fix.

```
//Old Code (Original Design)
assign data_o = data_state_q;

//New Code (Our Solution)
assign data_o = (valid_q)?
                data_state_q: 0;
```

At the start of an encryption operation, data_state_q is assigned the value of the input to the scrambler (data_i). Following this, data_state_q is assigned the result of each successive round of encryption (there are 32 in total). data_state_q will only contain the fully encrypted version of data_i after all rounds of encryption have completed. Before this, data_state_q will contain intermediate cryptographic state that could be used to learn the value of RndCnstKey and other secret assets. The old code continuously drives data_state_q to the output of the OTP scrambler (data_o), which leads to the intermediate state leakage outside of the OTP scrambler. To prevent this intermediate leakage, our proposed solution only drives data_state_q to data_o when data_state_q contains fully encrypted data; otherwise, it drives a safe default value (e.g., 0) to data_o. Figure 7 shows how this solution impacts information flows from RndCnstKey.

We disclosed this potential weakness and our proposed solution to the OpenTitan team. OpenTitan issued a patch to mitigate this leakage. The potential weakness is a low risk according to their threat model. However, the mitigation is simple, with minimal overhead. In addition to this disclosure to the OpenTitan team, we also submitted a new CWE to Mitre's CWE database[7] to cover the improper protection and leakage of intermediate cryptographic state; this weakness was not previously covered by existing CWEs and is expected to appear in future CWE releases.

We demonstrate the value of simulation-based hardware IFT analysis for hardware security verification. IFT quickly moves knowledge about design assets to define security requirements, security objectives, and security boundaries. IFT enables concise specification of security properties related to confidentiality, integrity, and availability. IFT hardware verification tools like Cycuity Radix can help verify, refine, and extend security properties. We identified a weakness in the OpenTitan OTP memory controller, localized the source of the error, and developed a hardware patch that was accepted as a pull request into the OpenTitan repository. We also submitted the findings as a hardware weakness to the CWE database. ■

References

1. J. McLean, "Proving noninterference and functional correctness using traces," *J. Comput. Secur.*, vol. 1, no. 1, pp. 37–57, Jan. 1992, doi: 10.3233/JCS-1992-1103.
2. M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Jan. 2010, doi: 10.3233/JCS-2009-0393.
3. W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *ACM Comput. Surv.*, vol. 54, no. 4, pp. 1–39, May 2021, doi: 10.1145/3447867.
4. R. Kastner, F. Restuccia, A. Meza, S. Ray, J. Fung, and C. Sturton, "Automating hardware security property generation," in *Proc. Des. Automat. Conf.*, 2022, pp. 1–6.
5. J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. Secur. Privacy*, 1982, p. 11, doi: 10.1109/SP.1982.10014.
6. S. Chong and A. C. Myers, "Security policies for downgrading," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, 2004, pp. 198–209, doi: 10.1145/1030083.1030110.
7. MITRE. [Online]. Available: https://cwe.mitre.org/data/index.html

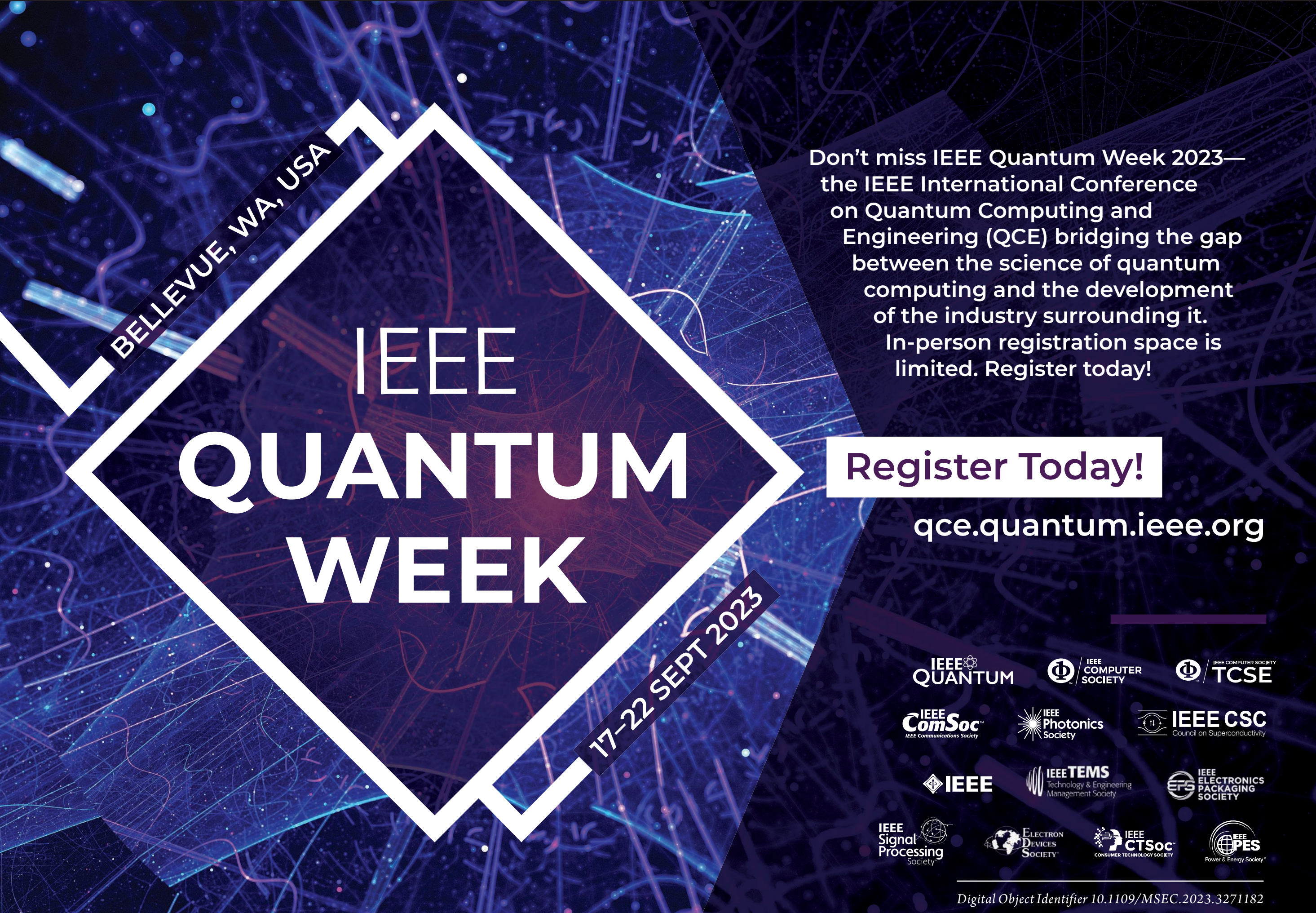
Andres Meza is a researcher at the University of California San Diego (UCSD), La Jolla, CA 92093 USA. His research interests include hardware security, optimization of machine learning models for hardware deployment, and computer vision. Meza received a B.S. in both computer science and cognitive science with a machine learning and neural computation specialization from UCSD. He is a Member of IEEE. Contact him at anmeza@ucsd.edu.

Francesco Restuccia is a postdoctoral researcher at the University of California San Diego, La Jolla, CA 92093 USA. His research interests include predictability, safety, security for hardware acceleration on heterogeneous platforms, cyber-physical systems, and time predictable hardware acceleration of deep neural network models on system-on-chip platforms. Restuccia received a Ph.D. in computer engineering (cum laude) from the Scuola Superiore Sant'Anna Pisa, Italy, in 2021. Contact him at frestuccia@ucsd.edu.

Jason Oberg is a cofounder and chief technology officer (CTO) of Cycuity, San Jose, CA 95113 USA. His research interests include hardware security, security verification, and vulnerability analysis. Oberg received a Ph.D. in computer science from the University of California San Diego. Contact him at jason@cycuity.com.

Dominic Rizzo is the founder and project director of the OpenTitan project, Cambridge CB2 1GE, United Kingdom. His research interests include hardening silicon implementations against physical attacks and side channels, trustworthy authenticators, and formal methods to provide implementation security and correctness guarantees. Rizzo received a B.S. in aerospace engineering from the Massachusetts Institute of Technology and an M.S. in computer science from the California Institute of Technology. Contact him at domrizzo@opentitan.org.

Ryan Kastner is a professor of Computer Science and Engineering at the University of California San Diego, La Jolla, CA 92093 USA. His research interests include hardware acceleration, hardware security, and remote sensing. Kastner received a Ph.D. in computer science from the University of California Los Angeles. He is a Fellow of IEEE. Contact him at kastner@ucsd.edu.



Digital Object Identifier 10.1109/MSEC.2023.3271182