Making Security Verification "SECURE"

Subin Thykkoottathil
Analog Devices India, Bengaluru
Subin.Thykkootathil@analog.com

Nagesh Ranganath
Analog Devices India, Bengaluru
Nagesh.Ranganath@analog.com

Abstract

Today's SoCs have to deal with information that requires high assurance of protection. Hardware security is critical as it forms the foundation for secure applications. A security hole in hardware can compromise the whole system. There is an increasing demand in the industry to verify the designs from the prism of security. This paper discusses a typical SoC with specific security requirements, its security verification challenges and how simulation and formal based verification methods complement each other in achieving the verification objective.

I. Introduction

Microcontrollers find use in various applications in the field of IoT, Automation, autonomous driving among many others. Security architects employ different methods to protect the device against security attacks. While developing the application, software developers thrive to make their code free of software holes. But root of trust is typically implemented in hardware. It may be a piece of code or data. Hardware designers have to protect root of trust against illegal accesses.

II. Security Verification

Consider a simplified diagram of microcontroller-based SoC, as shown in the Figure 1. It shows masters and slaves being connected through a bus interconnect. Peripherals may be either secure or nonsecure. Memory may be divided into sections; each section can either be secure or nonsecure. Masters may be secure, nonsecure or security-aware. Configuring security states of masters and slaves is generally a one-time process and it is done during the boot sequence. Exception to this is a security-aware master which allows runtime switching of its security state from secure to nonsecure and vice-versa. Secure_mode is the signal that reflects the current security state of the master. Secure masters accessing secure slaves is a valid transaction. But a nonsecure master accessing a secure slave can be a security leak.

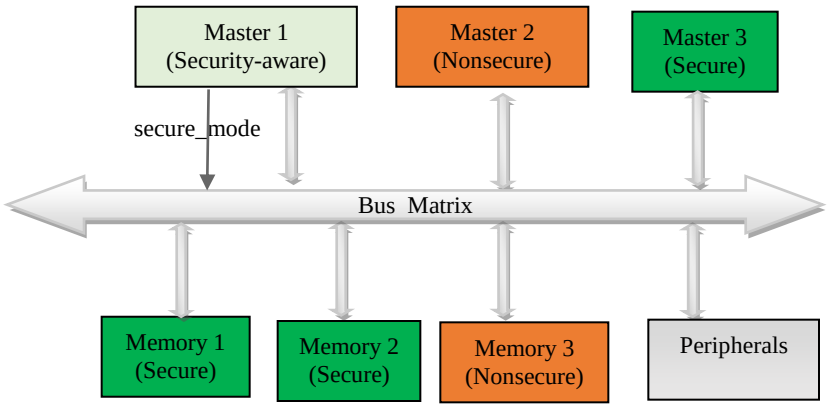

Figure 1: Microcontroller-based SoC with some security requirements

Depending on the application, memories and peripherals can be configured as secure or non secure. Master 1, in its secure mode, is allowed to do this configuration; typically in the beginning of the application. Secure memory could be code memory, data memory or even an internal register. Root of trust is expected to be placed in this secure memory. A secure peripheral could be either a communication peripheral transmitting secure information out of the chip or it could be an accelerator accelerating critical algorithms. In figure 1, master 2 is non_secure and master 3 is configured as secure. Similarly, memories 1 and 2 are secure and memory 3 is configured as non_secure. Given this scenario, the following are some of the security requirements:

1. Master 1 can access the secure memory 1 only if one of its interface signals "secure_mode" goes high.
2. Nonsecure master 2 cannot access the secure memory 1.
3. Once a peripheral is configured as secure, its configuration registers should not be overwritten.
4. When Master1 (in secure state) is accessing secure memory, there shouldn't be any data leakage to nonsecure master 2

Violation of any of the above requirements may invite adversaries to exploit them and lead to threats like IP theft, Identity theft, Config Tampering etc. Verification of hardware features designed to provide security is a different task in hand. Verification engineer has to think like a hacker and come-up with as many scenarios as possible, exercise them and prove that there are no security holes. Figure 1 is a simplified version of the actual SoC. In reality, SoCs are extremely complex with many peripherals and many such memory regions, each capable of being configured as either secure or nonsecure. Also, the number of masters in the system would be more than 10, including debug masters. Adding to the complexity is the fact that some masters are "security-aware". This means that not only is security verification exponential to each configurable option but also time consuming and laborious.

III. Verification Approaches

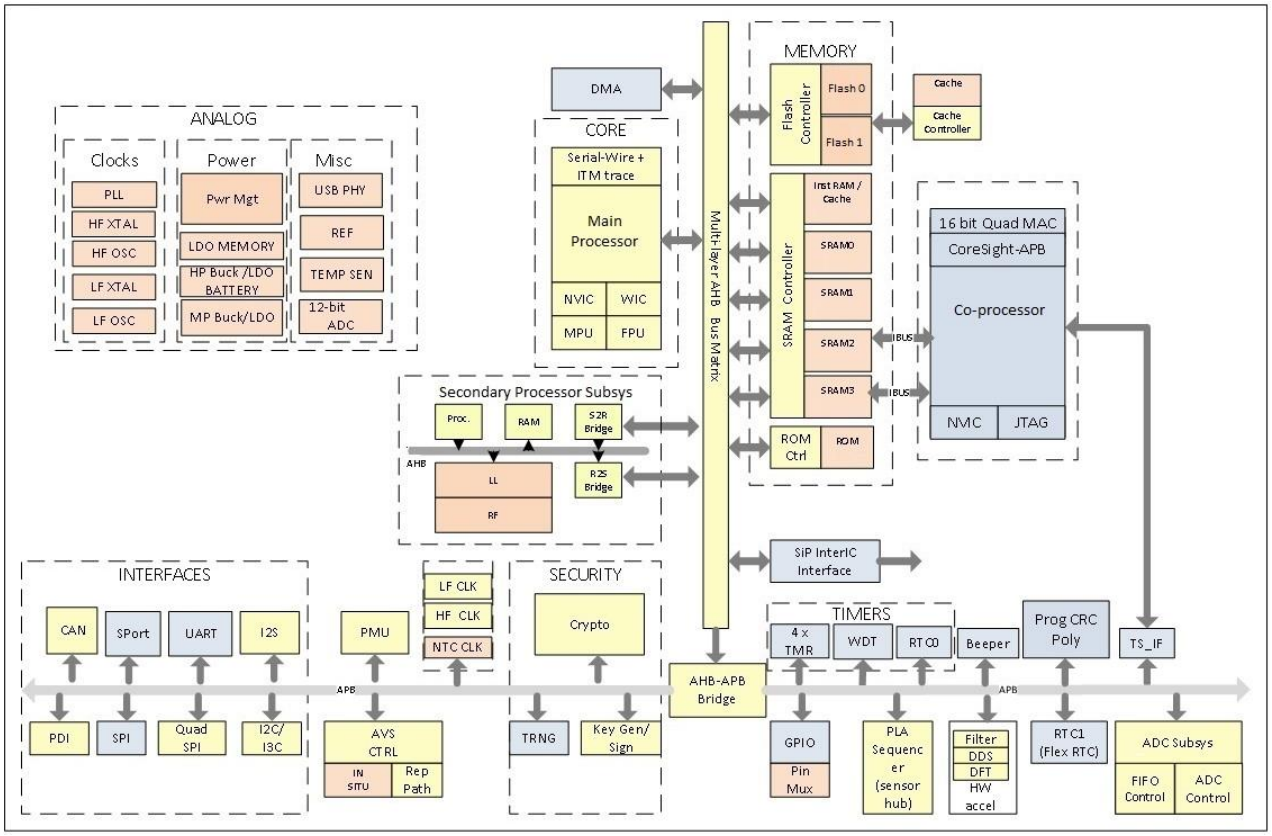

Figure 2: SoC Architecture

We worked on an AHB-based SoC, the architecture of which is shown in Figure 2. It consists of masters like main processor, secondary processor and DMA. Interfaces for communication, internal peripherals, controllers for Flash, SRAM and ROM are the among the slaves in the system. The task in hand was to understand the system security architecture, how the design implements the requirements specified in the architecture and comeup with a plan as to how each security feature needs to be verified. This section discusses the approaches adopted and the challenges faced in each of the approaches.

A. Simulations

The traditional approach is to simulate and prove that all our security requirements are met. It involves developing testcases for applying the stimulus as well as checkers to verify that the behavior is as intended.

SoC that we worked on had secure software in secure ROM memory. For the sake of illustration, consider the requirements of SoC shown in Figure 1. When Master 1 is in its secure state, it can execute the secure software from secure memory 1. No other master or even Master 1 in its non_secure state would be able to execute secure software. Security verification using simulations requires the following testcases:

1. Testcases to access all secure locations from both secure and nonsecure masters. Make sure the secure data is not getting leaked and the nonsecure master is not able to access the same.
2. Testcases to make sure that nonsecure software(when security-aware master is in nonsecure mode) cannot access any secure location.
3. Testcases to make sure that configuration of secure peripherals cannot be tampered by nonsecure software or any other nonsecure master.

We made these tests self-checking and thereby avoided developing the testbench checkers.

Verifying hardware security features using simulations was painful due to many reasons. We found that writing tests for certain scenarios was difficult. For example, in the case of requirement 4 in which secure master is accessing secure data, proving that there isn't any data leakage to nonsecure master 2 or nonsecure software is difficult. Structuring the testcases had to be different. A particular test scenario had to be analyzed and split into secure and nonsecure portions. Handling cross-security state function calls and exceptions in DV environment was challenging. Design supports a lot of configurable options: memory regions and their granularity, peripherals and their interrupts etc. With each configurable option, the number of test scenarios grows exponentially and writing test cases to cover all the scenarios is impossible. The presence of security-aware master (Master 1) and secure software, in secure memory, necessitated us to develop a different flow for testcase development. Compilation flow and related control scripts also had to be modified. This was to ensure that the secure software was compiled separately and loaded on to secure memory and nonsecure software to nonsecure memory.

Tests mentioned above are SoC-level tests and are directed at a particular scenario or requirement. Hence room for randomization is very little. An approach we used to increase the randmoness is by replacing masters with AHB-Verification IP and RAL-based test sequences to generate random accesses. The procedure followed is:

a. Preload a known key to all secure locations
b. Configure the masters and slaves as either secure or non-secure, depending on the application scenario
c. Use AHB RAL-based test and access all locations randomly.
    a. The same test was driven by both secure and nonsecure masters.
d. Assertions were coded to make sure that the key is not observed in nonsecure master interface.

Placing assertions at every other master's interface to check that the data is not reaching unintended destinations involved a lot of effort. This RAL-based test was helpful in generating random traffic. But the approach of generating random accesses and developing checkers was limited to a particular security configuration. Randomizing the security configuration of masters, slaves and other configurable options was not possible. Moreover, the test failed when the preloaded key from secure location is getting split and reaching nonsecure master interface.

In addition to all these challenges, there are no signoff metrics for security verification using simulations. Hence it is difficult to conclude whether the security verification is complete. Nevertheless, simulations were helpful to verify certain important use case scenarios involving secure and nonsecure softwares.

B. Formal Verification

While simulations can verify the presence of the paths, it does not verify their absence. This led us to explore formal verification as it excites all paths exhaustively. Even in terms of bring-up, it is formal verification setup which is easy and quick when compared to simulation setup.

Formal Verification requires assertions to prove that the design is meeting all the security requirements. Verification engineer has to translate the security requirements into SV Assertions. However, writing SVA assertions is not straightforward. For example, consider requirement 4. We would need to prove that a path exists from secure Master1 to secure memory 1 and at the same time, there are no paths from the slave to any other masters. Assertions must be coded at the interface of each nonsecure master such that the data from secure slave is not reaching them. A simple security requirement can result in a set of complicated assertions. In a complex design, with numerous such requirements, coding assertions would be labourious. These assertions if not properly constrained would take days of run time. The assertion can even go "explored", which means that during its run time, the tool was unable to prove or disprove the assertion. In the end, outcome is inconclusive. Also, these assertions fail if the data splits and a part of it is reaching the unintended master.

We decided to try out a new formal tool from Cadence, JasperGold Security Path Verification App. It is a dedicated formal app that identifies all paths between a given source and destination. It comes with two distinctive advantages:

1. Translating security requirements to assertions is fairly easy.
2. It can find paths between source and destination signals even if data mutates or splits.

The formal tool supports simple assertions to check if data can go from source A to destination B.

e.g. check_spv –create –from A –to B

As shown in figure 3, SPV injects a "taint" at Source, activates all paths using path sensitization technology and checks if the "taint" reaches the Destination. When a path is reported, it can be analyzed either using graphs or using waveforms. Figure 3 shows a graph to depict the path between source and destination points. For more information on this tool, please refer [1].

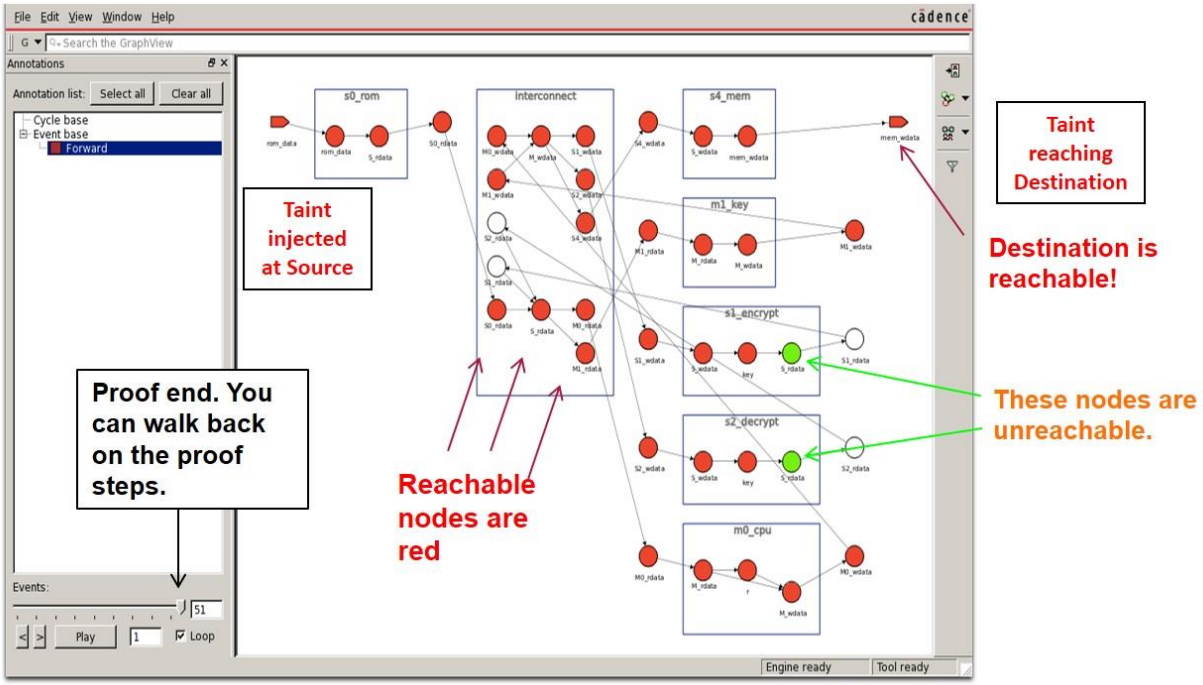

Figure 3: SPV explores paths between Source and Destination; reported using a detailed graph.

We employed the SPV app on the entire SoC. Its setup is similar to that of any regular formal verification tool. It requires a TCL control script where we essentially specify the following:

1. Define clocks and resets along with their constraints.
2. Non-synthesizable IPs like memory arrays were black-boxed.
3. AHB masters were blackboxed and replaced with AHB Assertion-based VIPs, provided by vendor

Any non-synthesizable IPs would be black-boxed by the formal tool. Because of blackboxing of memory arrays, say ROM array, the interface between ROM controller and ROM array becomes available for the formal tool for exhaustive toggling. Similarly, any master can be blackboxed, if we are interested only at its interface. Assertion-based AHB VIP were used instead of actual masters. If we want to find out a path from within the main processor to a point in the rest of the system, we should not blackbox main processor. In our work, we replaced main processor with AHB VIP, since we did not go into it. Point A, in that case, can be the interface of main processor. On the other hand, AHB VIPs reduce the formal app's run time. So it is a call that verification engineer has to take depending on the security verification requirement.

Once the setup is ready, the next step is to code "*create_spv*" assertions as per our security requirement. Points A and B are the RTL signals. SPV allows to set preconditions on some signals, using which we can force the signals to fixed values, if we want. The formal tool checks against data leaks as well as data overwrites. We listed out a set of illegal paths derived from system security architecture, wrote assertions and run the tool on the SoC design.

SPV assertion identifies all paths between A and B; be it an expected valid path or an unexpected path. It is these unexpected paths that give formal approach, an advantage over simulations. These paths could be invalid/false paths or be potential bugs. Proper analysis of the paths can result in identifying the bugs and fix them.

It took 6 weeks for setting up the simulation environment and to get the first testcase up and running; whereas the formal app required only a week for its bringup and to code the first assertion. SPV was able to quickly uncover some design bugs. Moreover, one such assertion replaced multiple testcases eliminating the need for positive and negative testing. On the other hand, some of the assertions took 3 days for completion. Assertions must be properly constrained in order to reduce the run time. This requires indepth understanding of the design. We tried measures such as constraining the address range of memories, replacing masters with VIPs etc. Nonetheless, formal verification brought in a sense of completeness into our security verification effort as against simulations which was inconclusive. Yet, simulation test cases were required as some usecase scenarios containing secure and nonsecure softwares were to be exercised. We generated coverage information from simulations.

IV. Design Bugs Found

This sections explains the bugs that we found out using Jasper SPV App. One of our requirements was that secondary processor should not have access to ROM memory. SPV assertion was written to check if there is any path from ROM controller to secondary processor, as shown in figure 4.

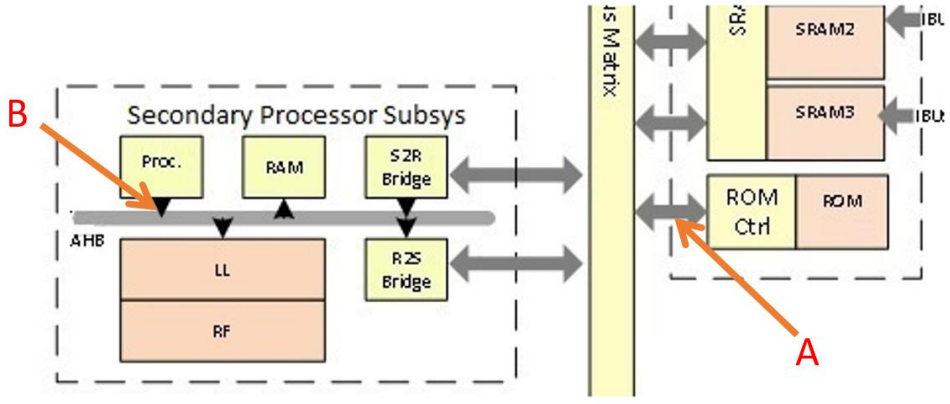

Figure 4: Assertion from Secure ROM Controller to Nonsecure Secondary Processor

SPV Assertion:

```
check_spv -create -name ROM_to_PROC2 -from $ROM_HRDATA -to $PROC2_HRDATA
```

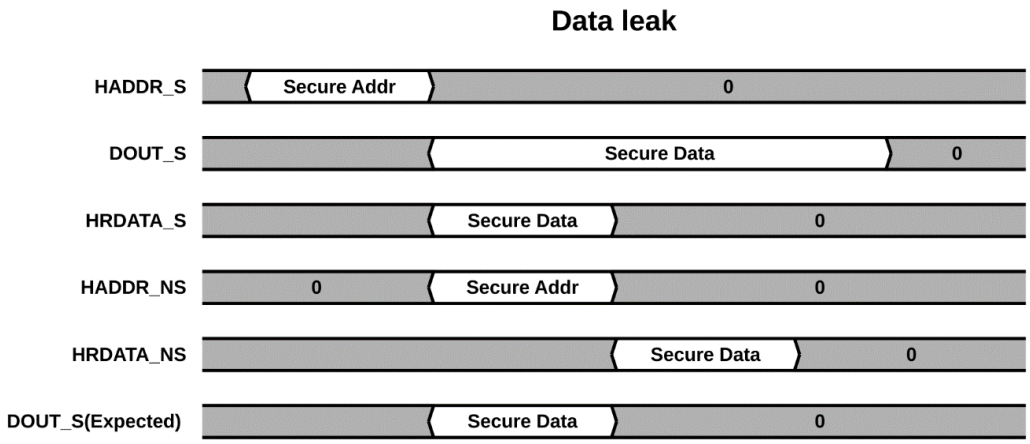

Figure 5: Waveforms illustrating the data leak.

A test case was written to check if secondary processor can access ROM contents. By simulations, we could not find any paths. But SPV was able to find a path between secondary processor and ROM data. This happened when secure access by secure master, to ROM was immediately followed by a nonsecure access by secondary processor. As can be seen from Figure 5, data from secure memory(DOUT_S) was reaching nonsecure master (HRDATA_NS). Though there was an error response by generating an error for the unsuccessful access, the data from the previous secure access was available at the interface. The issue was fixed by clearing the data right after the completion of secure access. Bugs like this can easily escape from simulations or manual review.

Another requirement was that the secondary processor should not have access to Crypto, when it is configured as a secure peripheral. This is as shown in figure 6. SPV assertion was coded for this requirement. Formal tool reported that nonsecure access from secondary processor to crypto's configuration registers was going through and hence secondary processor was allowed to overwrite the configuration registers of a secure peripheral.

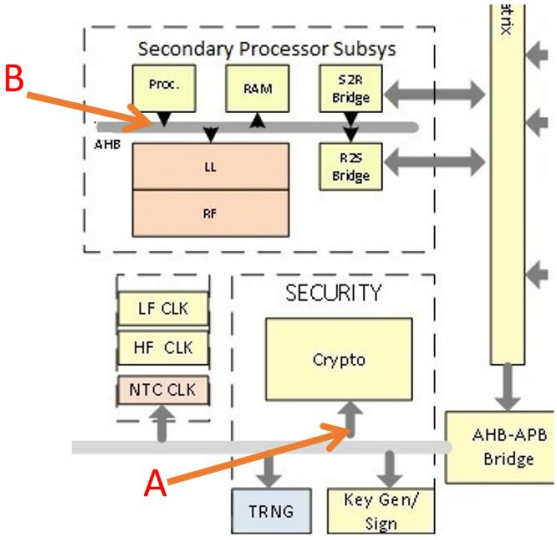

Figure 6: Assertion from Secure Crypto peripheral to Nonsecure Secondary Processor

SPV Assertion:

```
check_spv -create -name crypto_to_proc2 -from $crypto_prdata -from_precond { u_crypto.PADDR >= 12'h000 &&
u_crypto.PADDR <= 12'h0FF } -to $PROC2_HRDATA
```

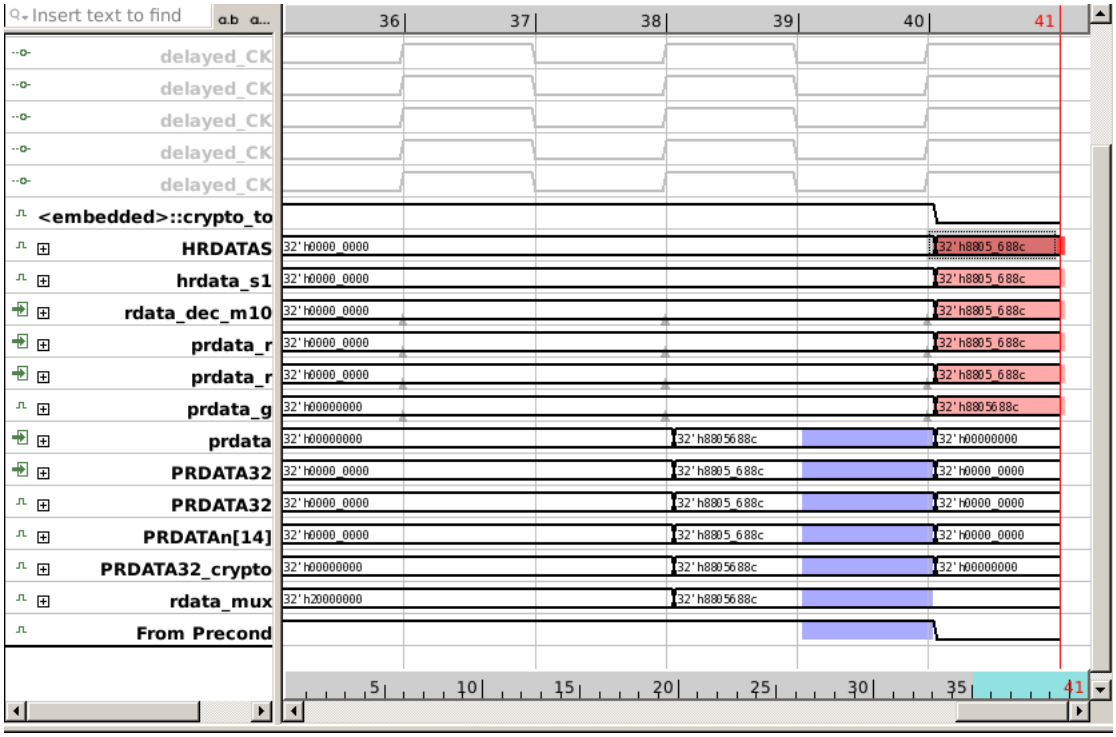

Figure 7: Assertion from Secure Crypto peripheral to Nonsecure Secondary Processor

Figure 7 is a snapshot of the waveforms reported by the formal tool. The assertion was run to identify paths through which data can leak. Taint, 0x8805688c, injected at crypto interface was reaching the nonsecure secondary processor, thereby proving that there is be a possibility of data leak. Simulations would have eventually caught this bug. But the point to note here is that we could catch the bug early and with minimal infrastructure support.

V. Conclusion

In the end, we had to overcome the following challenges posed by security verification:

- There is no tried-and-tested methodology towards security verification
  - It is difficult to conclude that the design is indeed secure, as there are no metrics
- Security cannot be verified at block level as we need the complete system for many of the scenarios
  - Limited room for randomization
- Complex design with lots of configuration options
  - Leads to exhaustive test scenarios
- Presence of Security-Aware Masters
  - Requires “security-aware” software development flow for verification.

We leveraged the capabilities of formal verification to a maximum extent in addressing these challenges. Equally, we used simulations to cover important usecase scenarios using secure and nonsecure softwares.

Given the nature of security verification, the traditional simulation approach will never prove to be sufficient. The Formal approach can expose hidden paths in the system and uncover critical bugs; thus complementing our simulation effort and making our security verification truly secure.

ACKNOWLEDGMENT

We would like to acknowledge Ameya Pangarkar for his help in understanding the security requirements of the design. We like to acknowledge Ravindrareddy Pulicharla and Nitin Neralkar for their contribution towards setting up of Jasper SPV tool and their help in coding assertions and debugging them. Finally, we would like to acknowledge Rohit Pandharipande for root-causing SPV failures and fixing the design bugs.

REFERENCES

[1] https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html