INVITED: Formal Verification of Security Critical Hardware-Firmware Interactions in Commercial SoCs

Sayak Ray, Nishant Ghosh, Ramya Jayaram Masti, Arun Kanuparthi, Jason M. Fung
{sayak.ray,nishant.ghosh,ramya.jayaram.masti,arun.kanuparthi,jason.m.fung}@intel.com

ABSTRACT

We present an effective methodology for formally verifying security-critical flows in a commercial System-on-Chip (SoC) which involve extensive interaction between firmware (FW) and hardware (HW). We describe several HW-FW interaction scenarios that are typical in commercial SoCs. We highlight unique challenges associated with formal verification of security properties of such interactions and discuss our approach of property-specific abstraction and software model checking to circumvent those challenges. To the best of our knowledge, this is the first exposition on formal co-verification of security-specific HW-FW interactions in the context and scale of a commercial SoCs. Despite traditional scalability challenges, we demonstrate that many such flows are amenable to effective formal verification.

KEYWORDS

HW-FW co-verification, System Security, Model Checking

1 INTRODUCTION

Security evaluation of system-on-chips (SoCs) involves opportunistic use of different techniques such as simulation, emulation, formal verification, post-silicon debugging as well as manual design/code review. Manual review is particularly important for security analysis of SoCs as currently available automated verification tools are often defeated by quirky security bugs which can only be uncovered through careful manual reasoning. The manual review process, however, has its own challenges. It is hardly scalable or repeatable. It tends to be tedious and error-prone, particularly while dealing with hardware-firmware (HW-FW) interactions. HW-FW semantic differences, concurrency and asynchronous nature of transactions make manual co-verification of HW-FW interactions extremely challenging. While automated verification tools could have aided manual review of HW-FW interactions, there is a serious void in the tool space as all such tools available today apply either to software or to hardware, but not across their boundary.

This gap has led to preliminary research on automated HW-FW co-verification techniques that would effectively complement manual analysis [6, 7, 9, 13, 15, 16]. These approaches are divided into two categories – (i) hardware is abstracted as software and the subsequent interaction is analyzed as a software verification problem [6, 16] (ii) firmware is compiled into assembly code and composed with hardware or its instruction level abstraction; the subsequent interaction is analyzed as a hardware verification problem [7, 9, 13, 15]. While the second approach leverages the engineering advances in hardware verification technologies, capturing a typical HW-FW interaction at the instruction level abstraction is often too detailed to scale for commercial SoCs. In contrast, the first approach abstracts away many hardware-specific details and retains only those firmware-visible architectural and micro-architectural states that are relevant for a particular security property [6]. We believe that the first approach is more promising than the second one to prove security properties of various HW-FW interactions in commercial scale SoCs. It is particularly true for the security properties for which cycle-accurate models are not needed.

In this paper, we present a case study of formally verifying security properties of several security-critical flows in a commercial SoC, by leveraging and improving upon the software verification approach presented in [6]. We focus on the model checking approach despite its known scalability issues because of its effectiveness in uncovering corner case bugs that are otherwise hard to find manually. This makes model checking suitable for bug hunting in security-critical modules such as bootROMs that may not be patchable in-field and could result in challenges for chip manufacturers if such vulnerabilities were to be discovered post-production. The performance and modeling costs associated with model checking in this case are justified by the higher level of security assurance.

Every commercial SoC includes a bootROM that plays a crucial role in ensuring that the SoC boots securely. Its primary objective is to ensure that only an authenticated bootloader is loaded and executed in the system. In that sense, it is the very first stage of the chain of trust which guarantees that the SoC will eventually run an authenticated operating system. Typical bootROM flows involve close interactions with hardware that are difficult to verify manually. This makes bootROM an ideal candidate for security analysis through a HW-FW co-verification framework using model checking.

Formal verification, particularly its bounded model checking capability, is typically used for bug hunting. In our case study, however, we emphasize its utility as a framework for regression. We successfully ran the model checking experiments on our target SoC and uncovered counterexamples to our defined security properties. Through manual analysis, we confirmed that the counterexamples we obtained were indeed security vulnerabilities and worked with designers to mitigate them before tape in. However, it may be prohibitively challenging to repeat the same manual analysis on different derivatives of the same products and over their successive generations. We highlight that our formal verification framework demonstrates the ability to automate this regression for known security issues on related products. More specifically, we make the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3323478

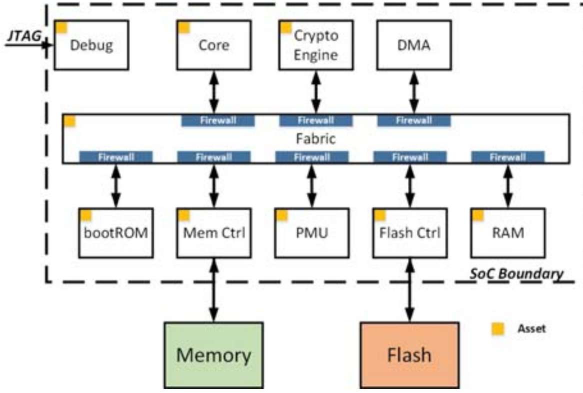

Figure 1: Typical System-On-Chip (SoC) Architecture

- We describe specific HW-FW interaction patterns that we observed during secure boot, runtime, and secure debug of a commercial SoC. We also identify specific challenges in formal verification of such interaction patterns.
- We demonstrate that property-specific abstraction of hardware operations in software do offer a viable methodology for formal verification of HW-FW interactions.

We are the first to identify and document the common patterns that most HW-FW co-verification frameworks should handle in order to provide security assurance. We believe that our approach offers a robust security assurance framework when used in tandem with existing HW-only and FW-only verification techniques.

2 RELATED WORK

Security architecture of SoCs is an active area of research [10, 19]. As new security architectures are being proposed to mitigate new security threats, need for their verification is also growing at equal pace. However, being a new paradigm, security co-verification of HW-FW interactions is not so well-understood as its functional counterpart [7, 9, 13, 16, 20]. Preliminary research results available for security co-verification can be divided into two categories, *viz.* information flow analysis [12, 14] and property verification through model checking [6, 8, 15]. Our work belongs to the second category as we use a software model checker to verify security properties of HW-FW interactions. The general approach of abstracting hardware-specific details into software and then using software model checkers on the composition to verify HW-FW interactions is not new [1, 6, 16]. However, the patterns of HW-FW interactions that we encountered in our experiments are not yet explored. Similarly, our focus on concrete model checking experiments distinguishes our work from previous work on security architecture of commercial SoCs [18] and its formal modeling for security analysis [17]. Abstraction is critical for successful verification in our approach. While various abstraction methods for HW-FW co-verification are available [9, 13], we found them not readily applicable to our problem. We thus propose our own property-directed abstraction method. Similar to prior work [2, 6], our abstraction method also avoids expensive bit-precise reasoning to speed up verification.

3 HW-FW SECURITY FLOWS

Figure 1 shows the high-level architecture of a generic SoC. The main processing core is accompanied by a cryptographic engine and a DMA engine. An interconnect fabric connects the bus masters to the peripherals, such as local RAM, a flash controller, a power management unit (PMU), a memory controller, and the ROM. The debug module provides an interface for debugging the SoC through interfaces such as JTAG. We consider software adversary and an adversary who can access the debug interface to be in scope. Table 1 shows a set of fundamental security-critical flows observed in such SoCs and the typical verification approach for each flow[1]. Flow 1 through 4 in Table 1 have a purely HW implementation and are verified using HW-only verification techniques such as RTL simulation, FPGA-based emulation, HW model checking, etc. Flow 5 has a purely FW implementation and these type of properties are verified using FW-only verification techniques such as static and dynamic code analysis tools, fuzzing, etc. Flow 6 and 7, on the other hand, can be implemented either using purely HW mechanism or using purely FW mechanism. If they are implemented using purely HW mechanism, HW-only verification techniques are used to verify their security claims. Otherwise, FW-only verification techniques are used. Industry-strength verification tools with well-established methodologies are available for verifying such HW-only and FW-only security flows. These methodologies, however, do not directly apply to security flows that involve HW-FW interactions. Flows 8 through 12 in Table 1 belong to this category. Effective verification methodology for such properties is an open and active area of research. In the subsequent sections, we present our methodology and experiments on formally verifying the security flows marked in bold in Table 1 (Flow 10-12). Below we explain niche of these flows in securing the system during boot, runtime, and debug.

3.1 HW-FW Interactions in Bootloaders

Bootloaders play a critical role in security of commercial SoCs [3–5, 11]. To help limit the impact of malicious code, bootloaders verify both the integrity and provenance of the software that an SoC executes. SoC bootloaders contain a chain of loaders with each one verifying the integrity and authenticity of the next. This creates a chain of trust, which stems from an immutable root of trust. In our case, this immutable root of trust is the bootROM. As the core in the SoC is powered on, it executes the immutable boot code that is stored in the bootROM. Primary tasks of this boot code include:

(*i*) Reading system security configuration from the fuses and initializing the SoC by writing (and locking) configuration values into numerous registers through MMIO write operations (Flow 11).

(*ii*) Loading the primary boot loader (PBL) either from the external flash or from an IO channel to the on-chip RAM (Flow 12).

(*iii)* Supporting test mode boot (described later) to allow for chip manufacturer debug during the early boot stage (Flow 10).

We will now discuss these tasks in detail below.

3.1.1 Configuration Register Locking. The configuration registers set by the bootROM are used in several critical hardware security

[1] Note that this is neither a comprehensive description of the SoC architecture nor the security property list is exhaustive. For the sake of brevity, we only described modules and properties that fall within the scope of this work.

Index	**Security Property**	**HW Only**	**FW Only**	**HW-FW**
1	Overlapping MMIO or memory ranges should follow principle of least privilege	✓	✗	✗
2	When lock bit is set, locked register cannot be modified	✓	✗	✗
3	Write-once registers cannot be set to 0 without a reset	✓	✗	✗
4	Untrusted entities cannot trigger IP reset to release the lock	✓	✗	✗
5	No related ranges are configured with conflicting settings	✗	✓	✗
6	Firewalls should implement access control policy	✓	✓	✗
7	Untrusted entities cannot bypass authentication for debug	✓	✓	✗
8	No asset is exposed before protected ranges are properly configured	✗	✗	✓
9	Signature verification from ROM to PBL or PBL to next boot stage cannot be bypassed	✗	✗	✓
10	**Authorized users cannot access more debug capabilities than allowed**	✗	✗	✓
11	**Trusted FW locks sensitive registers before execution of untrusted FW**	✗	✗	✓
12	**Correct image (download/copied) is selected based on boot configuration**	✗	✗	✓

Table 1: Various security flows in commercial SoCs and their verification strategy

flows. They can be used to store keys, configuration of access control filters, temperature sensor configuration, thermal trip settings, etc. The content of these registers are considered assets and need to be protected. BootROM, therefore, locks these registers after writing the configuration values and before control is transferred to untrusted firmware. Locks provide the benefit of protection against modification of assets by untrusted agents in case of a compromise through privilege escalation. Bit fields in registers can also be used as locks to protect other assets in the system or to protect themselves from further modification. Once a lock bit is set, the protected assets are locked off from future write accesses and are modifiable only when the lock is released, which usually happens when the system is reset.

3.1.2 Control Transfer to PBL. After signatures of PBL images are verified, whether control will be transferred to a PBL image copied from local memory (*eg.* flash memory) or downloaded from an IO interface (*eg.* PCIe interface) depends on various boot configurations. BootROM reads various fuse values and configuration registers through MMIO read operations to make this decision. Since the decision depends on tens of such parameters, it is easy to make implementation mistakes and deviate from the intended decision based on boot configuration. It is, therefore, important to formally verify that the implementation matches designers' intent.

3.1.3 Test Boot Mode. To facilitate debug of the boot process, certain SoCs introduce a test boot mode. This allows a privileged user to run test code of their choice to boot the system. In order to prevent the test code from making persistent changes to the system, this mode is time-limited by a watchdog timer.

4 METHODOLOGY

4.1 Source-level Modeling and Verification

All firmware modules in our case study are written in C. We capture all hardware behavior relevant to our flows as C subroutines. We observe that for our case study, the software abstraction is not required to be cycle accurate or bit-precise for proving the security properties under consideration. This results in significant improvement in both modeling and verification efficiency compared to hardware-based approaches [7, 9, 13]. The HW-FW interactions considered in this paper rely heavily on MMIO write operations. Our software modeling thus captures firmware-visible functional behavior of pertinent MMIO write operations and their side effects.

4.2 Verification Framework and Engines

The high-level security properties are broken down to assertions at the source code level. All HW-FW interactions are replaced by software abstraction models and assertions are compiled into an intermediate representation using LLVM compiler framework. The resulting code is then analyzed with SMACK model checker [21]. We note that any scalable model checker that supports verification of multi-threaded C code can be used for our purpose.

4.3 Verification Flow

Our verification flows begins with the given C code for firmware, RTL for hardware and supporting documents such as security architecture specification, register description etc. We analyze the HW-FW interaction under verification and its security objective. We then construct a software model of the underlying hardware using property-specific abstraction. The software model is composed with the firmware and the security objective is translated into assertions which are inserted in the composed model. The assertions are then verified using a software model checker.

5 SECURITY FLOW VERIFICATION RESULTS

We first consider Flow 10 from Table 1 which is associated with test boot mode. We present the security property associated with this flow and the corresponding abstraction and verification run time. We then present the same for two other flows, *viz.* hardware locking performed by firmware (Flow 11) and invocation of the proper authenticated bootloader (Flow 12).

5.1 Test Boot Mode

Certain SoCs offer two boot modes, *viz.* normal boot mode and test boot mode. In normal mode, the system boots up for normal operations by executing system boot code. Test boot mode, on the other hand, is a privileged mode that allows an authorized user to run test code of his/her choice to boot the system. Since the user gets full control of the system in this mode, security architecture of the SoC must ensure that the security objectives of the system are not violated in case a privileged user runs malicious test code. This is particularly important for protecting OEMs' secrets from the possibility of *insider attacks.* In this flow, a watchdog timer is

triggered to ensure that the system is restarted after the test code execution ends and the following property must be satisfied:

5.1.1 Security Property ϕ. In test boot mode, if execution of the user-supplied test code ends before the watchdog timer expires, operation of the system must halt.

5.1.2 Result. As SMACK manages to hit the property in 3 minutes, it demonstrates that it is possible to continue normal execution after user code execution completes. This indicates that not enough protection is placed in the implementation to prevent control from flowing into normal boot code, thus violating ϕ. We confirmed this bug through manual review and mitigated it subsequently.

5.2 Verification of register locks

Register locking is a hardware access control mechanism where a register (target register) or certain bit field within a register (target bits) is protected by another bit (lock bit) in such a way that the target register/bits can be written only when the lock bit is in reset state. Locking is heavily used during secure boot where the trusted boot code writes security-critical data to configuration registers and then locks them by setting designated lock bits to protect them from malicious overwriting during runtime.

5.2.1 Security Property. Security of the system relies on the assumption that the security sensitive configuration registers are properly locked by calling appropriate functions by the firmware. It is, therefore, required to guarantee that the respective locking functions are always called in every secure boot flow.

5.2.2 Result. SMACK takes less than 25 seconds to find conditions for bypassing lock functions we studied. In all such cases, SMACK has to analyze no more than 500 lines of user code (excluding library functions). While SMACK finds these conditions as counterexample to a reachability property, this counterexample is not a security bug in the implementation. An analysis of the counterexample cross-checks that the implementation indeed matches with designer's intent.

5.3 Signature verification flow

The bootROM decides which bootloader image to execute based on a number of factors including the availability and validity of the images retrieved from local storage and downloaded over the IO interface. The function first retrieves the bootloader image, verifies its integrity and authenticity, and finally invokes a deeply nested decision tree to call the appropriate function to execute the validated image based on its selection logic.

5.3.1 Security Property. The primary security requirement on this control flow is that the decision tree must implement the bootloader selection logic correctly. For example, under no circumstances, the bootloader retrieved from the local storage shall be executed when the image that is downloaded over the IO interface should be executed or no image should be executed at all (and vice versa).

5.3.2 Result. This flow amounts to analyzing around thousand lines of code. SMACK takes around 400 seconds on an average to derive a trace to a target call-site. Again, the counterexamples produced by SMACK are not implementation bugs. They are rather paths in the decision tree which cross-checks that the decision tree implementation is correct.

6 DISCUSSION, NEXT STEPS & CONCLUSION

Our approach of model checking is more effective when the underlying HW-FW interactions do not require extensive hardware modeling, handling of nested interrupts, or heavy multi-threading at the firmware level. In our observation, many fundamental security flows do not involve such complex HW-FW interactions. They involve only up to a few hundred lines of code (including both RTL and C) and can be analyzed by scoping out a bulk of context code. Such flows are, therefore, amenable to formal analysis through our methodology. The software models created in our methodology can be used beyond formal verification. For example, they can be used for emulation testing before the SoC is fabricated.

To conclude, we presented a low-cost methodology for formally verifying security properties of various HW-FW interactions. We verified important security flows taking no more than 400 seconds. Abstracting hardware into software models for all flows discussed in the paper together took only one day of work by one engineer. We argue that formal verification, though not scalable in general, has strong potential in verifying interesting security properties for HW-FW interactions in commercial SoCs. We hope that our approach will serve as a baseline for the development of new and improved verification techniques in the future.

REFERENCES

[1] A. Horn, *et al.* 2013. Formal co-validation of low-level hardware/software interfaces.. In *FMCAD.* 121–128.
[2] A. Lal, *et al.* 2012. A solver for reachability modulo theories. In *CAV.* Springer, 427–443.
[3] AMD. 2012. AMD SB800-Series Southbridges BIOS Developer's Guide. https://www.amd.com/system/files/TechDocs/45483.pdf.
[4] Apple. 2018. Apple BootROM 574.4. https://goo.gl/q3m5Ky.
[5] ARM. 2012. Principles of ARM® Memory Maps. https://goo.gl/oPVv55.
[6] B. Huang *et al.* 2018. Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware. *55th DAC* (2018).
[7] B. Schmidt, *et al.* 2013. A computational model for SAT-based verification of hardware-dependent low-level embedded system software. In *18th ASP-DAC.*
[8] Cook, B. *et al.* 2018. Model Checking Boot Code from AWS Data Centers. In *Computer Aided Verification (CAV) (LNCS)*, Vol. 10982. Springer, 467–486.
[9] D. Gro*β*e *et al.* 2006. HW/SW co-verification of embedded systems using bounded model checking. In *16th ACM GLSVLSI.* ACM, 43–48.
[10] E. Peeters, *et al.* 2015. SoC Security Architecture: Current Practices and Emerging Needs. In *52nd DAC.* Article 144, 6 pages.
[11] Intel. 2010. Minimal Intel Architecture Boot Loader. https://goo.gl/KLdq6a.
[12] M. Balliu, *et al.* 2014. Automating Information Flow Analysis of Low Level Code. In *ACM CCS.* 1080–1091.
[13] M. D. Nguyen, *et al.* 2011. Formal hardware/software co-verification by interval property checking with abstraction. In *48th DAC.* 510–515.
[14] P. Subramanyan, *et al.* 2016. Verifying Information Flow Properties of Firmware Using Symbolic Execution. In *DATE.* 337–342.
[15] P. Subramanyan *et al.* 2017. Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).
[16] R. Mukherjee, *et al.* 2017. Formal Techniques for Effective Co-verification of Hardware/Software Co-designs. In *54th DAC.* Article 35, 6 pages.
[17] S. Krstic, et al. 2014. Security of SoC firmware load protocols. In *HOST.* IEEE, 70–75.
[18] S. Ray, *et al.* 2015. Security Policy Enforcement in Modern SoC Designs. In *ICCAD (ICCAD '15).* 345–350.
[19] S. Ray, *et al.* 2018. System-on-Chip platform security assurance: architecture and validation. *Proc. IEEE* 106, 1 (2018), 21–37.
[20] Y. Abarbanel, *et al.* 2014. Validation of SoC Firmware-Hardware Flows: Challenges and Solution Directions. In *51st DAC (DAC '14).* Article 2, 4 pages.
[21] Z. Rakamarić, *et al.* 2014. SMACK: Decoupling source language details from verifier implementations. In *CAV.* Springer, 106–113.