Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware*

Bo-Yuan Huang[1], Sayak Ray[2], Aarti Gupta[1], Jason M. Fung[2], Sharad Malik[1]
[1]Princeton University [2]Intel Corporation

ABSTRACT

Formal security verification of firmware interacting with hardware in modern Systems-on-Chip (SoCs) is a critical research problem. This faces the following challenges: (1) design complexity and heterogeneity, (2) semantics gaps between software and hardware, (3) concurrency between firmware/hardware and between Intellectual Property Blocks (IPs), and (4) expensive bit-precise reasoning. In this paper, we present a co-verification methodology to address these challenges. We model hardware using the Instruction-Level Abstraction (ILA), capturing firmware-visible behavior at the architecture level. This enables integrating hardware behavior with firmware in each IP into a single thread. The co-verification with multiple firmware across IPs is formulated as a multi-threaded program verification problem, for which we leverage software verification techniques. We also propose an optimization using abstraction to prevent expensive bit-precise reasoning. The evaluation of our methodology on an industry SoC Secure Boot design demonstrates its applicability in SoC security verification.

1 INTRODUCTION

Contemporary Systems-on-Chip (SoC) contains a combination of different Intellectual Property Blocks (IPs) communicating through on-chip interconnect, as illustrated in Figure 1. Within an individual IP, there are specialized accelerators for performance critical functions, e.g. encryption, and also hardware assisted security solutions like secure storage. Meanwhile, firmware (FW) runs on a programmable processor and accesses specialized hardware (HW) through memory-mapped I/O (MMIO). The two levels of interaction, IP/IP and FW/HW, make their co-verification challenging.

The first challenge is dealing with the FW/HW interactions. Figure 2 shows an example of firmware accessing hardware to lock a key (lines 5 and 6) before enabling a cryptographic accelerator, where the *locking* of register write access is implemented in hardware. Since the *hardware functions are not captured by program semantics*, no off-the-shelf software verifier, e.g., CBMC [3], can be used for sound verification of these interactions. Further, formally verifying firmware and hardware components together using bit-precise cycle-accurate models does not scale for multiple IPs. What

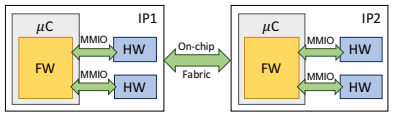

Figure 1: System-on-Chip High-Level Overview

```
1  uint32_t status = *ADDR_STATUS; // mmio read
2  if ((status >> 8) == INIT)
3    for(int i=0; i<KEY_SIZE; i++)
4      *(ADDR_KEY+i) = KEY[i];     // mmio write
5  status |= 1;                    // set lock bit
6  *ADDR_STATUS = status;          // mmio write & lock
7  *ADDR_ENABLE = 1;               // mmio write & enable
```

Figure 2: Firmware Setting up a Cryptographic Accelerator

is needed here is a suitable level of abstraction that bridges the FW/HW semantic gap, while providing scalability in verification.

Another verification challenge is in handling concurrency. Within an individual IP, firmware runs on the processor in parallel with the accelerators. Among IPs, firmware sends and processes messages to/from other components. Previous works have addressed reasoning about FW/HW concurrency [7, 15]. However, to the best of our knowledge, concurrent firmware in multiple IPs has not been formally verified. This is especially important in a heterogeneous environment where IPs potentially have *different* processors and implement distinct message handling and synchronizing mechanisms, which increases the vulnerability to security attacks.

Bit-precise reasoning poses yet another scalability challenge. Firmware often uses bit-wise operations, e.g., shifting and masking, to access hardware states stored in aligned hardware registers. For example, line 2 in Figure 2 shows a right shift to extract a 24-bit value, and line 5 shows a bit-wise OR operation to set the first bit. Checking such operations precisely requires expensive bit-wise reasoning, e.g., via bit-blasting.

In this paper, we present a co-verification methodology to address these challenges. Specialized hardware features, e.g., access control and direct memory access (DMA), are abstracted using an Instruction-Level Abstraction (ILA) to handle FW/HW interactions [21]. These ILAs are semi-automatically generated from high-level specifications, whereas firmware programs are modeled at the source level to avoid detailed modeling of various Instruction-set architectures (ISAs) in a heterogeneous SoC. This enables representing the FW/HW behavior of an IP as a *single program thread*. We can then model co-verification of interacting IPs as a *multi-threaded program*, where each thread corresponds to the firmware/interrupt handler of an IP or the attacker. This allows us to leverage techniques in software verification, such as context-bounding, to manage the concurrent interleavings. We further improve scalability using an optimization to abstract certain bit-wise operations, which lowers the cost of expensive bit-precise reasoning.

We report the application of our co-verification methodology on an industry SoC Secure Boot design where two heterogeneous IPs communicate with each other. We explore a vulnerability where

*This work was supported in part by Semiconductor Research Corporation (SRC). It was performed during the first author's internship at Intel Corp.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196055

an attacker is able to spoof commands in absence of access control protection. Further, when access control protection is deployed to mitigate this attack, we prove several properties relating to message handling and DMA. Our experimental results show significant improvement due to our proposed optimization for abstracting bit-wise operations.

Overall this paper makes the following contributions:

- It shows the use of ILAs for FW/HW co-verification of multiple IPs and demonstrates its applicability on an industry-scale SoC design.
- It leverages software verification techniques for scalable co-verification of parallel firmware and specialized hardware in heterogeneous SoCs.
- It proposes an optimization that abstracts bit-wise operations to lower the cost of expensive bit-precise reasoning.

The paper is organized as follows. In Section 2, we discuss the system architecture of our case study for an industry SoC, which serves as the running example through this paper. We then explain the threat model and security concerns of the SoC Secure Boot in Section 3. Section 4 describes our co-verification methodology and the proposed optimization. Experimental results, related work and conclusions are discussed in Sections 5, 6, and 7, respectively.

2 SYSTEM OVERVIEW FOR CASE STUDY

Contemporary SoCs contain multiple interacting IPs that communicate through an on-chip interconnect, as illustrated in Figure 1. An IP typically contains a programmable processor and several specialized hardware components for implementing performance-critical functions or hardware-assisted security, e.g., DMA, encryption, access control. The firmware runs on the processor and interacts with this hardware through MMIO to implement the IP's functionality.

In this section, we describe the system architecture for our case study of an industry SoC design. We apply our co-verification methodology for security verification of its Secure Boot feature.

2.1 System Architecture

Two interacting IPs are relevant to the Secure Boot flow, the security engine (SE) and a functional module (FM), similar to the set-up in Figure 1. FM is a security critical IP, e.g., a camera unit, power management unit, or radio unit, whose run-time image needs to be authenticated by SE before being booted. SE and FM communicate through the on-chip interconnect, to which other IPs can also send messages. The access control policy ensures that only valid (white-listed) IPs can access the SE/FM communication channel.

2.1.1 Security Engine (SE). To reduce the trusted computing base, cryptographic accelerators and security critical assets like encryption keys are maintained in SE. Its firmware runs on a single-threaded IA-32 processor, where the interrupt is masked during the boot phase. SE maintains a secure memory that is hardware isolated and internal-access-only, with the exception of the DMA engine in FM. The firmware uses MMIO to configure internal routing logic for controlling the accessibility of the secure memory.

2.1.2 Functional Module (FM). FM's firmware runs on a 32-bit RISC processor with non-nested interrupts. Messages to/from the on-chip interconnect are buffered and processed by specialized hardware, which may trigger interrupts to the processor when

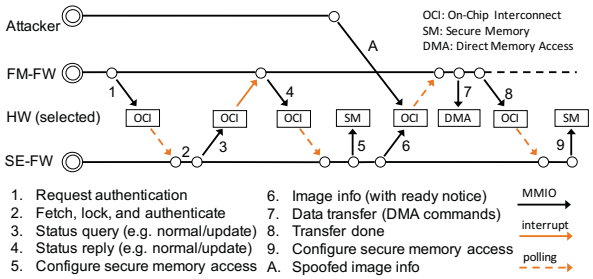

Figure 3: Simplified Boot Flow and Possible Attack

handling some types of commands. Thus, message handling in FM is a combination of interrupt routines and polling firmware. Further, there is a DMA engine for boosting data transfer, such as copying the authenticated run-time image. The DMA engine in FM can access the secure memory in SE. FM firmware configures DMA and message handling logic using MMIO.

2.1.3 On-chip Interconnect and Communication Protocol. IPs interact through an on-chip interconnect following a prescribed communication protocol. The protocol defines the interface, handshake mechanism, and the architecture-visible behavior. For example, all IPs are required to have a *doorbell* for each communication channel to implement the handshake mechanism. The doorbells at the two ends of a channel are mirrored, i.e., any write to one side will propagate to the other side. Further, accessing a doorbell can trigger an interrupt to the processor if the interrupt is not masked.

Note that the protocol only specifies the required functionality, but not its implementation. In our case study, the protocol implementation can be a combination of firmware programs and physical registers within hardware logic, as in FM. Alternatively, it can be implemented using specialized hardware only, as in SE. This heterogeneity in the FW/HW boundary increases the difficulty in abstracting this interface, and raises the importance of co-verification, especially when interacting IPs have different synchronizing methods.

2.1.4 Access Control Policy. The on-chip interconnect routes and transmits messages to IPs, and the receiving IPs should incorporate access control protection. The designs in the case study implement a white-list-based protection. All messages sent to the on-chip fabric are tagged with a source hardware ID (HID), and each IP maintains a **valid list** of trusted HIDs. The protection logic blocks accesses from untrusted entities not in the valid list. A failure in access control may result in privilege escalation, enabling messages from unauthorized entities being accepted. Correct configuration and integrity of the white-list is a critical security objective. In this work, we assume that hardware IDs are not spoof-able. The verification of HID uniqueness is beyond the scope of this paper.

3 SOC SECURITY AND SECURE BOOT

Modern SoCs implement a multitude of mechanisms to meet their security requirements. For example, hardware protected secure memory is used for storing confidential assets. Security primitives can be implemented using a combination of hardware accelerators and firmware programs. Further, access control protection can be employed to prevent inter-IP communication from being attacked. Verifying these security mechanisms requires not only checking the

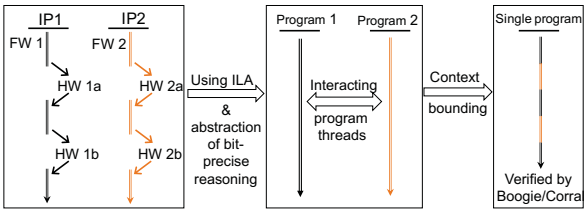

Figure 4: Verification Methodology

hardware logic, but also reasoning about the FW/HW interaction and the parallel execution between communicating IPs.

In this work, we study the security verification of an industry SoC with a Secure Boot flow, the primary guarantee of trust. This case study features complex FW/HW interactions and concurrent execution between FW/HW within an IP and across IPs. We first introduce the high-level aspects of the Secure Boot flow, then discuss the security concerns and the threat model under consideration.

3.1 Secure Boot

Firmware integrity, one of the most fundamental security objectives [4], specifies that `all devices, at any time, can run only authenticated firmware`. In Secure Boot, a feature to help resist attacks from malware, every piece of the firmware image should be authenticated before being booted if it is updated or fetched from an untrusted source.

To reduce the size of the trusted computing base, there is usually a trusted component implementing all security primitives, such as SE in Section 2. The trusted component authenticates the digital signatures of all images to boot in the system, then sends the image to the destination devices following the firmware load protocol. All components should enforce proper access control and lock/unlock mechanisms to ensure that only a valid image can be executed [9].

Steps 1 to 9 in Figure 3 show a simplified boot flow where FM requests SE to authenticate its runtime image, and uses DMA to transmit the authenticated image from the secure memory in SE. Note the various FW/HW interactions and interactions between different IPs in the system.

3.2 Security Concerns

Access control is critical for ensuring confidentiality and integrity of secured assets and blocking untrusted external messages. However, as images and signatures are large, transferring data with DMA is common to boost this process. This enlarges the attack surface because DMA engines often bypass normal access control mechanisms for performance reasons. The enlarged attack surface and FW/HW interplay raise security concerns and require thorough co-verification.

Concurrency is yet another concern for security. The support for communication protocol and message handling between IPs often differs from design to design. Some functionalities are implemented in hardware, while others are programmed in firmware. Some IPs have interrupt routines handling the events, while others poll for updates. Designs can even switch between them to improve performance [18]. Such heterogeneity is considered error-prone, especially in the context where IPs (and some potential attackers) execute in parallel. Failure of incorrect synchronization and the

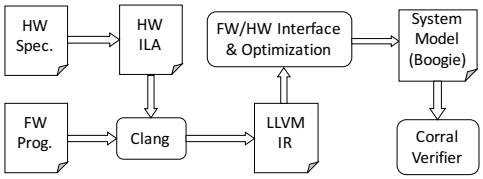

Figure 5: Verification Toolchain

Time-of-Check/Time-of-Use (TOC/TOU) [2] issue are particularly of interest for security verification.

In Figure 3, we also illustrate an example attack, where the attacker spoofs the command between SE and FM while running in parallel with them. Without a proper mechanism to block invalid messages, the attacker can spoof SE and send an instrumented command (step A) notifying FM for image ready with an incorrect source address and/or size of image. The DMA engine then accesses the secure memory with an invalid configuration, bypassing normal access control.

3.3 Threat Model

The two classes of security objectives we consider are *integrity* and *confidentiality* of firmware assets. For example, we wish to ensure that only the authenticated image will be transferred. We also wish to ensure that no data in the secure memory can be read by untrusted IPs due to mis-configuration of the DMA engine.

Our threat model covers the perspective of multiple components, beyond an individual IP. We consider an attacker, e.g., an untrusted IP, that has access to the on-chip interconnect, and can send commands to other IPs, at any given time, with any arbitrary message. Here we assume that the attacker has no hard-wired access to critical internal registers, and can only attack through sending commands to access states (registers) of the communication interface. We also assume that the attacker cannot spoof the hardware IDs used in the access control policy. This can be verified separately.

4 METHODOLOGY

In this section, we describe our methodology for system modeling, abstraction, and verification. A pictorial overview of the overall verification methodology is shown in Figure 4, where the ultimate goal is to model the co-verification of interacting IPs as a multi-threaded program verification problem, in which each thread corresponds to the firmware/interrupt handler of an IP or the attacker. First, we abstract specialized hardware components, e.g., access control and DMA (shown as HW 1a, HW 1b, etc. in the figure), by using Instruction-Level Abstraction (ILA). Next, these ILA models are composed with FW *within each IP*, to model each IP as a separate program thread. Finally, to verify the concurrent interactions of multiple IPs, we leverage context-bounding techniques from software verification to verify their concurrent interleavings.

Figure 5 shows the verification toolchain that implements our methodology. First, ILA models of specialized hardware are semi-automatically generated from a register-level hardware specification. Along with hardware ILAs, described in C, the firmware and interrupt routines of all IPs are compiled into an LLVM intermediate representation (LLVM-IR) using the LLVM front-end Clang [1]. We compose the firmware programs and hardware ILAs at the LLVM-IR level by substituting each MMIO access in firmware with

the corresponding ILA instruction (based on the address mapping). Thus, the FW/HW interaction gets modeled as a software program.

We then use the SMACK translator [17] to translate this model into a Boogie program [5]. Encoding of property assertions and optimization are also performed during this step. The final system is checked using the software verifier Corral [11]. As shown in Figure 4, we integrate FW/HW interaction within an IP using hardware ILAs, and model the interaction across IPs as interacting parallel programs. We now discuss salient aspects of our methodology.

4.1 Instruction-Level Abstraction

Co-verification of firmware and hardware with low-level finite state machine hardware models is too detailed and does not scale. Here, we found the Instruction-Level Abstraction (ILA) a perfect fit for FW/HW co-verification [8, 21]. ILA is a high-level abstraction for functional behavior of accelerators *at the architecture level*. It extends the familiar notion of instructions to accelerators and provides an abstraction for modeling firmware-visible behavior while abstracting low-level implementation details. It specifies hardware behavior as a set of instructions, in which each instruction corresponds to a command on the interface, e.g., MMIO writes, and defines how it affects the *architectural state*. In other words, for memory-mapped accelerators, an ILA instruction is essentially a functional specification that captures the firmware-visible behavior of an MMIO instruction.

In this work, we constructed the abstraction semi-automatically by translating from the machine-readable hardware specification that comes with every hardware design in the industrial environment where we investigated the case study. It describes the firmware-visible behavior when accessing each of the architectural registers. With relatively low manual effort to complete the model, as will be shown in Section 5, the translation generates ILA models where each ILA instruction is represented as a C function.

4.2 Source-Level Modeling and Verification

Firmware programs of all interacting IPs are modeled at the source level, C in our case, where the properties to verify are expressed as assertions. We model and verify the firmware at source level for several reasons.

- Verifying firmware at the instruction level, e.g., [6, 16, 19], requires modeling the instruction semantics of the underlying processor. Due to growing heterogeneity in SoCs, this would require a lot of effort in modeling various instruction sets.
- Preserving the source level program structure enables better utilization of software verification techniques, e.g., inferring loop invariants, abstraction by function summary, etc.
- Rich support in the LLVM ecosystem facilitates rapid development of a verification toolchain.

Beyond an individual firmware, we model the communicating IPs as a multi-threaded program. Each firmware program or interrupt routine (if not masked) of an IP is initiated in a program thread, as is the attacker. Architectural states of the hardware interface between firmware threads are modeled as shared global variables.

The access control mechanism blocks invalid accesses based on the tagged hardware ID (HID). In this work, we abstract the HID generation logic, and perform the checking on the program thread ID returned by system calls.


```
1  uint32_t global_hw_reg;
2
3  void lockChannel() {
4    uint32_t x;
5    x = mmioRead(ADDR);
6    x |= 1; // set lock bit
7    mmioWrite(ADDR, x);
8  }
9
10 uint32_t readVersion() {
11   uint32_t x;
12   x = mmioRead(ADDR);
13   return (x >> 8) & 255;
14 }
15
16 uint32_t readID() {
17   uint32_t x;
18   x = mmioRead(ADDR);
19   return (x >> 8) & 31;
20 }
```

Figure 6: Register Bit-Fields and Firmware Example

```
1  typedef struct _reg_t {
2    int lock;
3    int sts;
4    int rsv;
5  } reg_t;
6
7  void lockChannel() {
8    reg_t x;
9    x = mmioRead(ADDR);
10   x.lock = 1;
11   mmioWrite(ADDR, x);
12 }
13
14 int readVersion() {
15   reg_t x;
16   x = mmioRead(ADDR);
17   return x.rsv;
18 }
19
20 int readID(){
21   reg_t x;
22   x = mmioRead(ADDR);
23   return x.rsv;
24 }
```

Figure 7: Optimized Firmware

In our case study, the FW interacts with HW through MMIO instructions, where the instructions can be viewed as function calls to hardware that are blocking. In general, such calls may be non-blocking, requiring additional concurrent exploration of threads for hardware instructions, as in [7].

4.3 Verification Engine

The system model represented in LLVM-IR is converted into a monolithic Boogie program (an intermediate verification language) [5], which is then be checked by the software verifier Corral [11]. The LLVM-IR to Boogie conversion is extended from SMACK [17] to support bit-precise checking for parallel programs. The Corral/Boogie verifier generates verification conditions for procedures across the program, which are then checked by a theorem prover/SMT solver. This general approach has been applied successfully to industry-scale designs [10]. In this paper we leverage it with our FW/HW modeling to address SoC security verification challenges.

One of the critical challenges in our problem is to verify all possible interleavings of different IPs, which is notoriously difficult due to state explosion. Here, Corral sequentializes the concurrent execution by bounding the number of thread contexts [12]. This improves scalability in bug finding and provides bounded-proofs up to a given bound.

4.4 Optimization for Bit-Wise Operations

We propose an optimization where we over-approximate bit-wise operations on bit-fields of hardware registers based on two observations in our case study.

- Security properties relying on hardware-assisted access control usually depend only on *configuration registers*, and on a few bits in the registers.
- Most bit-wise operations in the firmware are used for accessing bit-fields in the aligned hardware register.

Multiple hardware architectural states are often packed into one aligned register as different **bit-fields**. Therefore, accessing these hardware states requires the firmware to use bit-wise operations for masking and shifting. Figure 6 shows an example of a 32-bit register

containing three bit-fields and the firmware functions accessing it. Bit-wise operations are used in lines 6, 13, and 19.

In SoC security verification, especially at the FW/HW boundary, precisely checking bit-wise operations is expensive, e.g., via bit-blasting. Therefore, we over-approximate some variables and bit-wise operations based on their hardware specification in the ILA and by analyzing firmware patterns. We now informally explain the optimization with an example. Figure 7 shows the optimized version of the firmware in Figure 6. Note that the actual optimization is implemented at the Boogie program level, here we show the C version for ease of understanding.

(1) Hardware registers and dependent program variables are normally modeled as fix-width bit-vectors to enable bit-precise reasoning. In the optimization, we model these variables as arithmetic integers. Registers containing multiple bit-fields are modeled as a set of integers, representing the architectural state defined in the hardware specification. For example, in Figure 7, type *reg_t* is used to model three architectural states.
(2) Bit-wise operations for accessing architectural states are substituted by expressions in linear integer arithmetic, if applicable. For example, masking then shifting is replaced by an assignment to the associated architectural state. In cases where the operation is not confined to bit-field manipulation for hardware states, e.g., firmware defined usage, we can selectively apply over-approximation. In our example, line 6 in Figure 6 is replaced by the equivalent assignment, whereas the firmware-defined bit-field manipulations in lines 13 and 19 are further abstracted.

The replacement for hardware defined bit-fields and their manipulations is sound and complete if the value to assign does not exceed the maximum capacity of the bit-field. This property can be checked separately. In the case study, we identified the bit-fields and the manipulations by analyzing firmware patterns based on the ILAs.

In practice, security critical states are implemented using dedicated hardware bits, e.g. *lock* in the example, and are well-defined in the hardware specification. Meanwhile, firmware-defined data manipulations are usually security irrelevant and are apt to be abstracted. These design characteristics enable the applicability of the optimization, as demonstrated in our case study.

5 EVALUATION

This section describes our case study of verifying the Secure Boot flow in the industrial SoC design discussed in Section 2. We wish to verify the system when there is an untrusted IP (attacker) running in parallel with the security engine (SE) and the functional module (FM). We will describe the properties being checked and the verification results.

5.1 Experimental Results

We performed the experiments on a 32-bit virtual machine with an Intel i5 2.4 GHz core and 4GB memory. Table 1 summarizes the experimental results of verifying the case study with and without applying bit-precise abstraction. Columns 2 and 3 show the upper bounds for thread context switches [12] and loop unrolling used by the verifier, determined based on design knowledges. Verification time and memory usage are reported in columns 4 and 5, respectively. Columns 6 to 8 show the (source level) length, number of thread contexts, and the number of hardware interactions (MMIO) in the counter-example trace if the property is violated.

The system model comprises two main parts: (1) firmware programs, and (2) hardware abstractions, i.e., ILAs. Firmware programs of FM and SE, including relevant interrupt routines, are implemented in C with 18.5k lines of code. Hardware ILAs, e.g., the DMA engine and on-chip fabric, are semi-automatically generated from hardware specifications. The specifications are available in a machine readable format (14k lines) and describe the firmware-visible behavior when accessing each of the architectural registers. The generated ILAs are represented in C, where about 500 lines are manually constructed in a total of 9500 lines for the abstraction models. As a reference, after the unused functions are removed by LLVM passes, the final integrated model has about 8.9k lines of LLVM-IR. Further, we abstract some parts of the operations as uninterpreted functions, for improving scalability and reducing modeling effort. They can be verified in a separate step.

5.1.1 Access Control Mitigation. We consider an attacker trying to spoof the messages between SE and FM by sending commands to the on-chip interconnect. We first verified the integrity of the message between SE and FM without modeling the access control protection. The property was violated, and the verifier generated a counter-example, denoted as $A1$ in Table 1. The trace showed a path where the attacker modified the value of an interface register during SE/FM interaction, hence spoofing the intended message. We then modeled the mitigation that blocks the invalid access to the interface states. The tool gave a context-bounded proof for end-to-end command integrity and non-accessibility of the interface state from untrusted IPs, denoted as property $A2$ and $A3$, respectively.

5.1.2 DMA and Message Handling. The second set of properties we checked is to ensure correct configuration of DMA commands and the message handling mechanisms, denoted as properties D_i and I_i, respectively. No invalid address should be accessed and only dedicated messages will be processed. We verified these two sets of properties without modeling the attacker, i.e., only SE and FM. As shown in the table, no bug was found up to the given bounds.

5.1.3 Scalability of Bug Finding. To further explore how the methodology scales, we instrumented the design, and placed error states in tricky corner cases. The traces reaching the error states are denoted as R_i in the table. The results show the applicability of our methodology in finding bugs that heavily interleave between different IP firmware, interrupt routines, and hardware operations. Such reasoning for concurrency among deep function calls in a huge code base is especially hard for human inspection and simulation-based validation.

5.1.4 Improvements due to Optimization. As Table 1 shows, there are significant improvements in both the verification time, memory usage, and also the quality of proofs (higher bounds) when our optimization is applied. With less than 75 lines in the model being manually modified (other parts are automated), the optimization enables better scalability in not only proving properties but also in bug finding.

6 RELATED WORK

Previous works address the importance of verifying SoC security by applying formal verification at different levels, from firmware

Table 1: Verification Results

	Without Optimization							With Optimization						
	Bounds		Resource Usage		Counter Example			Bounds		Resource Usage		Counter Example		
Prop	Ctx	Unroll	Time (s)	Mem (Mb)	Length	Ctx	MMIO	Ctx	Unroll	Time (s)	Mem (Mb)	Length	Ctx	MMIO
A1	-	-	-	-	-	-	-	12	5	45.6	124.1	395	7	4
A2	-	-	-	-	-	-	-	12	5	5308.9	139.5	-	-	-
A3	-	-	-	-	-	-	-	12	5	5190.0	157.2	-	-	-
D1	30	5	8.9	134.9	-	-	-	45	10	4.7	153.9	-	-	-
D2	30	5	44686.0	2780.7	-	-	-	45	10	133.9	472.2	-	-	-
D3	30	5	45094.9	3033.3	-	-	-	45	10	135.8	477.9	-	-	-
I1	30	5	14.6	126.9	-	-	-	45	10	4.9	150.2	-	-	-
I2	30	5	1075.9	411.1	-	-	-	45	10	5.7	158.8	-	-	-
R1	30	5	1673.2	313.7	235	4	1	30	10	30.3	269.8	240	4	1
R2	30	5	42641.2	913.7	332	4	3	30	10	69.0	387.9	349	4	3
R3	30	5	timed out (24hr)					30	10	977.2	559.4	709	7	12
R4	30	5	timed out (24hr)					30	10	922.5	591.2	817	7	16
R5	30	5	timed out (24hr)					30	10	4053.6	655.7	1153	11	23

load protocols [9] to the underlying hardware RTL logic [20]. In our work, we address formal co-verification of the Secure Boot implementation, checking parallel firmware programs of multiple communicating IPs, along with the hardware accelerators.

There have been efforts for co-verifying single-threaded firmware with hardware at instruction level, using bounded model checking [6, 19] and interval property checking [16]. To reason about FW/HW concurrency, Kroening *et al.* perform symbolic execution on the firmware with hardware RTL models [15] and virtual prototypes (software models serving as hardware proxies) [7]. They too model firmware programs at a higher level (CIL) to avoid detailed ISA modeling, similar to our use of LLVM-IR as an intermediate representation. However, our methodology verifies *concurrent* firmware of *multiple* interacting IPs, and leverages software verification techniques for scalable co-verification.

Other related works use concolic execution for finding information flow bugs [22] and use automata-theoretic verification techniques to verify PCI drivers [13]. To reduce the effort in verification, use of abstractions [6, 14, 16] and avoiding expensive bit-precise reasoning [10] are the keys.

7 CONCLUSIONS

This paper presents a formal co-verification methodology for verifying firmware of interacting IPs along with specialized hardware, and demonstrates its applicability in security verification in a heterogeneous SoC. We model the co-verification as a multi-threaded program verification problem (as shown in Figure 4), and leverage software verification techniques. Firmware programs are modeled at the source level to avoid detailed ISA modeling, and specialized hardware components are modeled using the Instruction-Level Abstraction (ILA) for better scalability in verification. ILAs capture firmware-visible behavior at the architecture level, and are semi-automatically generated from hardware specifications. We propose an optimization that uses abstraction to further avoid expensive bit-precise reasoning.

Our proposed verification methodology applies broadly to the general SoC security verification problem shown in Figure 1, and we demonstrate it on a specific case study of an industrial SoC Secure Boot implementation. Our experiments explore an attack where commands can be spoofed in the absence of access control protection, and prove command integrity when its mitigation is deployed. The experimental results show significant improvements due to our proposed optimization for handling bit-precise operations in accessing bit-fields. In future work, we will investigate use of program synthesis techniques for finding suitable linear integer-based abstractions for expensive bit-precise operations.

REFERENCES

[1] Clang: a C Language Family Frontend for LLVM. https://clang.llvm.org
[2] S. Bratus, N. D. Cunha, E. Sparks, and S. W. Smith. 2008. TOCTOU, Traps, and Trusted Computing. In *TRUST*. 14–32.
[3] E. Clarke, D. Kroening, and F. Lerda. 2004. CBMC - A Tool for Checking ANSI-C Programs. In *TACAS*, Vol. 2988. 168–176.
[4] A. Cui, M. Costello, and S. J. Stolfo. 2013. When Firmware Modifications Attack : A Case Study of Embedded Exploitation. In *NDSS*.
[5] R. Deline and K. R. M. Leino. 2005. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs of Program and Types.* Technical Report.
[6] D. Große, U. Kühne, and R. Drechsler. 2006. HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In *GLSVLSI*. 43–48.
[7] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening. 2013. Formal Co-Validation of Low-Level Hardware/Software Interfaces. In *FMCAD*. 121–128.
[8] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *arXiv preprint arXiv:1801.01114* (2018).
[9] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor. 2014. Security of SoC Firmware Load Protocols. In *HOST*. 70–75.
[10] A. Lal and S. Qadeer. 2014. Powering the Static Driver Verifier Using Corral. In *FSE*. 202–212.
[11] A. Lal, S. Qadeer, and S. Lahiri. 2012. Corral : A Solver for Reachability Modulo Theories. In *CAV*. 427–443.
[12] A. Lal and T. Reps. 2009. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *Formal Methods in System Design* 35, 1 (2009), 73–93.
[13] J. Li, F. Xie, T. Ball, V. Levin, and C. Mcgarvey. 2010. An Automata-Theoretic Approach to Hardware/Software Co-verification. In *FASE*. 248–262.
[14] S. Malik and P. Subramanyan. 2016. INVITED: Specification and Modeling for Systems-on-Chip Security Verification. In *DAC*. 1–6.
[15] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening. 2017. Formal Techniques for Effective Co-verification of Hardware/Software Co-designs. In *DAC*. 1–6.
[16] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz. 2011. Formal Hardware/Software Co-Verification by Interval Property Checking with Abstraction. In *DAC*. 510–515.
[17] Z. Rakamari and M. Emmi. 2014. SMACK : Decoupling Source Language Details. In *CAV*. 106–113.
[18] J. H. Salim, R. Olsson, and A. Kuznetsov. 2001. Beyond Softnet. In *ALS*. 18–18.
[19] B. Schmidt, C. Villarraga, J. Bormann, D. Stoffel, M. Wedler, and W. Kunz. 2013. A Computational Model for SAT-based Verification of Hardware-dependent Low-Level Embedded System Software. In *ASPDAC*. 711–716.
[20] P. Subramanyan and D. Arora. 2015. Formal Verification of Taint-propagation Security Properties in a Commercial SoC Design. In *DATE*. 1–2.
[21] P. Subramanyan, B.-Y. Huang, Y. Vizel, A. Gupta, and S. Malik. 2017. Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst* (2017).
[22] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung. 2016. Verifying Information Flow Properties of Firmware using Symbolic Execution. In *DATE*. 1393–1398.