INVITED: Specification and Modeling for Systems-on-Chip Security Verification

Sharad Malik and Pramod Subramanyan*
Department of Electrical Engineering, Princeton University.

ABSTRACT

This paper describes a methodology for system-level security verification of modern Systems-on-Chip (SoC) designs. These designs comprise interacting firmware and hardware modules which makes verification particularly challenging. These challenges relate to (i) specifying security verification properties, and (ii) verifying these properties across firmware and hardware. We address the latter through raising the level of abstraction of the hardware modules to be similar to that of instructions in software/firmware. This abstraction, referred to as an instruction-level abstraction (ILA), plays a similar role to the instruction set architecture (ISA) definition for general purpose processors and enables high-level analysis of SoC firmware. In particular, the ILA can be used instead of the cycle-accurate bit-precise hardware implementation for scalable verification of system-level security properties in SoCs.

We introduce techniques to semi-automatically synthesize the ILA using a template abstraction and directed simulations of the SoC hardware. We describe techniques to ensure that the ILA is a correct abstraction of the underlying hardware implementation. We then show how the ILA can be used for SoC security verification by designing a specification language for security properties and an algorithm based on symbolic execution to verify these properties. Our case studies apply ILA-based verification to an example SoC built out of open source components as well as part of a commercial SoC. The methodology discovers several bugs in the hardware implementation, simulators and firmware.

CCS Concepts

•Security and privacy → Security in hardware; Logic and verification; •Hardware → Functional verification;

*This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA and a research gift from Intel Corporation. Part of this work was performed when Pramod Subramanyan was an intern with Intel's Security Center of Excellence in Hillsboro, OR.

DAC 2016 Austin, Texas USA

DOI: http://dx.doi.org/10.1145/2897937.2911991

1 Introduction

Today's integrated circuits are complex Systems-on-Chip (SoC) devices consisting of multiple programmable cores, accelerators, sensors and I/O devices [23, 25]. Specialized firmware executes on the programmable cores and *orchestrates* the operation of various accelerators and the functionality of the system is implemented by this combination of hardware and firmware. To verify that the system-level security requirements of SoCs are met, we need to analyze both hardware and firmware as well as the hardware/firmware interface. Clearly, bugs can exist in the hardware or firmware themselves, but some bugs may also be due to incorrect assumptions made by hardware/firmware about the other component.

For a specific example, consider the runtime binary authentication protocol discussed by Krstic et al. [15] The objective of the protocol is to read a binary from an I/O device, verify it is signed by a trusted RSA public key, and if so, load the binary into local memory for execution. Krstic et al. demonstrate that such a protocol is vulnerable to various attacks; *e.g.*, a malicious entity may modify the loaded binary after its signature is verified, but before it is loaded for execution. To prevent this, firmware needs to configure the memory management unit (MMU) to "lock" the pages containing the binary during and after signature verification. Verifying that this protection works requires precise specification of the hardware/firmware interface for the MMU, ensuring that hardware correctly implements the protection, and verifying that firmware sets the MMU configuration correctly. A mistake in any of these steps could violate the security requirements of the SoC.

The above example demonstrates the need for verification that analyzes the hardware and firmware together. Unfortunately, formal verification of the cycle-accurate and bit-precise register transfer level (RTL) hardware description along with the firmware is not feasible for even small SoCs due to scalability limitations of formal tools. And as we argue above, verifying hardware and firmware separately can miss bugs at the hardware/firmware interface.

1.1 Challenges in SoC Security Verification

Scalable co-verification of hardware and firmware in SoCs requires the use of *abstractions* of SoC hardware instead of the bit-precise and cycle-accurate RTL description. Such abstractions omit low-level microarchitectural details, thus enabling formal analysis.

1.1.1 Challenges in Constructing Abstractions: Although the idea of constructing abstractions for SoC verification is promising and has been proposed by past work [20, 33, 34], there are three challenges in applying this in practice. First, for an abstraction to be generally useful, it must all capture all firmware/hardware interactions as well as updates to all firmware-visible states. We argue that manually constructing an abstraction that captures all of

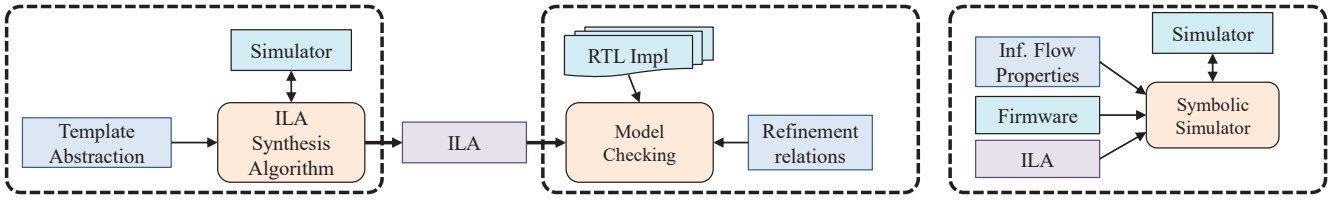


Figure 1: Block Diagram of Template-Based Synthesis of Instruction-Level Abstractions)

these details is both error-prone and tedious.

Completeness of an abstraction is extremely important for security verification. Finding security vulnerabilities requires reasoning about *all* inputs and states of the system including invalid/illegal inputs. For instance, [28] describes a bug affecting certain misaligned store instructions in a commercial SoC. A misaligned store instruction will cause an exception and therefore should not be executed by a well-behaved program. However, malicious code may specifically execute this instruction in order to exploit the above bug and corrupt MMU state. If verification were limited to "legal" inputs and states, or if an abstraction did not precisely model the behavior under illegal inputs, such violations will be missed.

Abstraction soundness is also very important. If the abstraction does not accurately capture of hardware behavior, proofs of system-level properties made using the abstraction are invalid.

1.1.2 Specifying Security Properties: Another challenge is *property specification*. Commonly-used property specifications based on temporal logics – such has linear temporal logic (LTL) and computation tree logic (CTL) – cannot express the security requirements of confidentiality and availability [17]. Confidentiality and integrity can be specified using *information flow properties*.

State-of-the-art techniques for *verifying* information flow properties are based on dynamic taint analysis and secure type systems [2, 14, 19, 21, 24, 26]. Dynamic taint analysis cannot systematically search over all possible inputs and states while new languages with secure type systems cannot verify existing and legacy firmware. Adoption of new programming languages is also complicated by the fact that significant parts of firmware are still written in assembly language. These reasons point to the need for tools that can perform exhaustive analysis of information flow properties on *binary code*.

1.2 SoC Verification Using Instruction-Level Abstractions

In this paper, we introduce a principled technique for the construction of abstractions for verification of system-level security properties in SoCs. The insight underlying our work is that firmware can only view changes in system state at the granularity of instructions. Therefore, it is sufficient to construct an abstraction which models hardware components of the SoC at this granularity. We call this an *instruction-level abstraction* (*ILA*) [30].

1.2.1 ILA Synthesis and Verification: To help easily construct the abstraction in a semi-automated manner, we build on recent progress in syntax-guided synthesis [1, 12]. Instead of manual construction of the complete ILA, the verification engineer constructs a *template* abstraction, which can be regarded as an abstraction with "holes." Our synthesis framework fills in the "holes" through directed simulation of hardware components.

A key advantage of our methodology is that the ILA can be verified to be a correct over-approximation of the hardware implementation. This uses model checking and ensures the ILA accurately captures the behavior of the RTL description. When model checking is complete and all properties are verified, we have a strong guarantee that all properties proven using the ILA are valid.

1.2.2 Security Verification Using the ILA: To address the problem of security property specification, we introduce a specification language for information flow properties of firmware. These properties specify that information cannot "flow" from a given source to a given destination and can be used to verify confidentiality when the source is a secret and the destination is any untrusted location. Integrity can be verified with if the source is an untrusted location and the destination is a sensitive firmware register.

Using the ILA as a formal model of the underlying hardware, we introduce an algorithm based on symbolic execution to verify these information flow properties. The algorithm exhaustively explores all paths in the program and creates symbolic expressions corresponding to the computation along each path. It then uses a constraint solver to check whether two different values at the source can result in different values at the destination. If yes, it means information flow can occur from source to destination because the destination value depends on the source. This means the property is violated. If such values cannot be found for the source, the property holds for this particular path.

1.2.3 Summarizing ILA-based Verification: An overview of the complete methodology is shown in Figure 1. ILA synthesis, shown in Figure 1(a), enables semi-automatic synthesis of the ILA from a template abstraction. Verification of the ILA against the implementation is shown in Figure 1(b) while use of ILA to verify security properties is shown in Figure 1(c).

2 Instruction-Level Abstractions

In this section, we describe what an instruction-level abstraction is, how it can be synthesized, techniques for verifying the correctness of an ILA and an overview of our experimental results related to constructing and verifying ILAs. As we are limited by space, this section provides only a brief overview of each of these topics. A detailed and formal description of the algorithms and methodology may be found in [30].

2.1 ILA Definition

An instruction-level abstraction (ILA) is an abstraction that captures the firmware-visible hardware behavior in the context of an SoC. Typically, accelerators and I/O devices on an SoC interact with firmware through memory-mapped I/O (MMIO). Firmware reads and writes to MMIO are commands/requests sent from firm-

ware to an accelerator. Each command either sets a particular configuration or requests the accelerator to perform a certain operation.

The key insight is to view MMIO reads/writes as part of an extended instruction set architecture (ISA) specification, referred to as an ILA. For example, in an AES accelerator, an MMIO write to the `Start` register corresponds to the `StartEncryption` instruction. The bit-pattern for the write: the write-enable signal and the address of the register form the opcode for this "instruction."

The ILA models the effects of its abstracted "instructions" much like an ISA. It specifies the computation performed by each instruction and what updates to architectural state will occur. For ILAs, the architectural state now includes all software-visible state that is accessible to its execution in the system, including memory-mapped registers, shared memory buffers, accelerator scratchpads, etc. For example, in the accelerator from [30] implementing the SHA-1 hashing algorithm, the `ComputeHash` instruction reads the source data from memory, computes its hash and writes this result back to memory. The location of this instruction's input and output data in memory is set by previous configuration instructions.

2.2 ILA Synthesis

For ILAs to be useful, one must be able to generate them correctly and preferably automatically. Due to the prevalent use of third-party component IPs, SoC hardware blocks often exist *before* the ILA is constructed, and so ILAs need to be constructed *post hoc* for existing accelerators and I/Os. To address the error-prone and tedious aspects of this construction, we have developed an algorithm for *template-based synthesis* of ILAs.

Instead of manual construction, ILAs can be synthesized from partial descriptions referred to as template abstractions using techniques drawn from program synthesis [1, 12]. The synthesis algorithm is able to "fill-in-the-blanks" in an incomplete abstraction by using data obtained from directed simulations of the accelerator.

2.2.1 Template Abstraction: Figure 2 shows a pedagogical example of ILA synthesis for a very simple ALU-based processor. Notice that the template abstraction does not specify low-level details such as the mapping between opcodes and operations, the bitfields in the opcodes which correspond to source registers and bitfields which correspond to immediate values, and so on. The template uses one of the *synthesis primitives* introduced in [30]: the `choice` operator. This operator tells the synthesis tool that the result computed by the instruction is *one of* the operands of the `choice` operator and the synthesis algorithm must use directed simulations to figure out which one it is. We introduced two more synthesis primitives in [30] and these operators can be nested or cascaded to arbitrary depths to form powerful synthesis constructs.

2.2.2 Synthesis Algorithm: Since the template abstraction is a partial description, it describes not one ILA but a family of ILAs. The intuition behind the synthesis algorithm is to pick some two different ILAs in this family and use a constraint solver to find a *distinguishing input* for which the behavior of these two ILAs is different. Once the distinguishing input is found, we can use the simulator to find the "correct" output for this input and rule out at least one of these two ILAs. In the next iteration, we repeat the above process, but this time we find distinguishing inputs among ILAs that are consistent with the input/output behavior we observed in the first iteration. This process repeats until no more distinguishing inputs can be found and we have an ILA consistent with the input/output behavior observed thus far.

2.3 ILA Verification

Once an ILA has been synthesized, its it is important to ensure that it is correct and is an accurate overapproximation of hardware implementation behavior. These checks are bidirectional; they can either be used to check that the ILA itself was correctly synthesized, or they can be used to verify that RTL description of hardware block abides by the ILA. We have introduced a methodology and a set of tools which can be used to ensure that the behavior of an ILA matches the RTL [30]. The approach requires specifying a set of *refinement relations* which "connect" the behavior of the ILA with the behavior of the RTL in terms of which state values need to match and when [13, 18]. A model checker is then used to verify that the refinement relations hold.

2.4 Practical Case Study

This section describes the evaluation methodology, the example SoC used as a case study, and then briefly describes the synthesis and verification results.

2.4.1 Methodology: The template-based synthesis framework was implemented as a Python library using the Z3 SMT solver [7]. A modified version of Yosys was used to synthesize netlists from behavioral Verilog [32]. We used ABC for model checking [3]. The synthesis framework, template abstractions, synthesized ILA, and other experimental artifacts are available online [8].

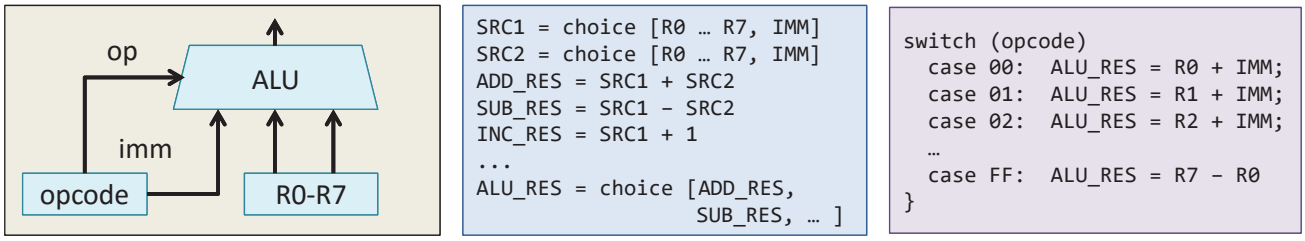
2.4.2 Example SoC Structure: Experiments were conducted on an example SoC constructed from open source components containing the 8051 microcontroller and two cryptographic accelerators. One accelerator implements encryption/decryption using the Advanced Encryption Standard (AES) while the other implements the SHA-1 cryptographic hash function [11, 27]. The RTL description of the 8051 is from OpenCores.org [31]. We also used *i8051sim* for instruction-level simulations of the 8051 [16].

2.4.3 Summary of Synthesis Results: As an indication of the effort involved in building the model, the size of the template ILA for the 8051 is about 800 lines, while the simulator is about 3000 lines of C++ code and the RTL is about 10,000 lines of Verilog code. The template ILA for the accelerator is was about 600 lines of Python code. These numbers demonstrate that the template ILA can be written with relatively little effort. Execution time for various elements of architectural state ranges between a few seconds to an hour, with most runs completing in just a few seconds. ILA synthesis found 5 bugs in *i8051sim*.

2.4.4 Typical ILAs: Tables 1 and 2 show "instructions" in the ILAs for the accelerators. Firmware first *configures* the accelerator with the appropriate instructions and then starts operation with the `StartEncryption` and `StartHash` instructions. It can then poll for completion using the `GetStatus` instruction.

Instruction	Description of operation
`Rd/Wr DataAddr`	Get/set the address of data to encrypt.
`Rd/Wr DataLen`	Get/set the length of data to encrypt.
`Rd/Wr Key0 <index>`	Get/set specified byte of key0.
`Rd/Wr Key1 <index>`	Get/set specified byte of key1.
`Rd/Wr Ctr <index>`	Get/set specified byte of counter.
`Rd/Wr KeySel`	Get/set the current key (key0/key1).
`StartEncryption`	Start the encryption state machine.
`GetStatus`	Poll for completion.

Table 1: ILA instructions for AES accelerator.



(a) Simple ALU (b) Template ILA (c) ILA

Figure 2: Pedagogical Example of a Template Abstraction and corresponding ILA

Instruction	Description of operation
`Rd/Wr DataInputAddr`	Get/set address of data to be hashed.
`Rd/Wr DataLength`	Get/set length of data to be hashed.
`Rd/Wr DataOuptutAddr`	Get/set the address of output.
`StartHash`	Start the SHA1 state machine.
`GetStatus`	Poll for completion.

Table 2: ILA instructions for SHA1 accelerator.

2.4.5 Summary of Verification Results: We generated Verilog "golden models" from the ILAs and defined a set of refinement relations specifying that the the golden models were equivalent to the RTL. We then used bounded and unbounded model checking to verify these refinement relations. This process found 6 bugs in the RTL description of the 8051 microcontroller from OpenCores.org.

3 Security Verification Using ILAs

The ILA is a complete formal specification of hardware behavior and enables scalable system-level verification. Firmware that interacts with accelerators and I/O devices can now be analyzed in terms of firmware-visible behavior as specified by the ILA instead of *ad hoc* manually constructed models, or at the other extreme, very detailed bit-precise cycle-accurate RTL descriptions.

In this section, we describe how ILAs can be used to verify confidentiality and integrity properties of firmware using symbolic execution. We first describe a specification language for firmware security properties, briefly provide an overview of an algorithm based on symbolic execution for verifying these properties and then show experimental results. Due to limited space, this section provides only a brief overview of this work. Details may be found in [29].

3.1 Specifying Information Flow Properties

The property specification language for firmware security properties is based on the following insights. First, security requirements such as confidentiality and integrity are essentially statements about information flow. These express the requirement that either a firmware secret must not "flow" to an untrusted value (confidentiality), or an untrusted value must not "flow" to a sensitive asset (integrity). Note such properties cannot be expressed using specification languages based on temporal logic [17].

Second, almost all interesting firmware security assets, such as secret keys, sensitive configuration registers and untrusted input registers are accessed through memory-mapped I/O (MMIO). Therefore, firmware address ranges and architectural registers are first class entities in the property specification language.

Third, a mechanism for *declassification* is required [22]. This allows information flow when certain conditions hold during execution; information flow is disallowed if these do not hold.

Based on the above requirements, we introduce a specification language for information flow properties consisting of:

1. A `src` which is a range of firmware memory addresses.
2. A predicate `srcpred` associated with the source which specifies when the data at `src` is valid. For example, we may allow a register to be programmed from an input port during the boot process, but not afterwards with the predicate $\neg boot$.
3. A `dst` which is an element of the ILA state.
4. A predicate `dstpred` for `dst` which specifies when data at `dst` is valid similar to (2) above.

The property holds if data read from `src` when `srcpred`=1 never influences a value written to `dst` when `dstpred`=1. `srcpred` and `dstpred` are evaluated at the time of the read and write respectively.

3.2 Verifying Information Flow Properties

This section provides an intuitive description of how information flow properties can be verified using symbolic execution.

3.2.1 Overview of Algorithm: Intuitively, when verifying a information flow property from `src` to `dst`, the algorithm attempts to answer the following question: can *some* two different values at `src` result in different values at `dst`? If so, then information *does* flow from `src` to `dst`. If not, then `dst` is not influenced by `src` and the property holds. The algorithm poses this question to a constraint solver, and this allows us to consider all possible values for `src` and `dst` on every path in the program.

The algorithm performs a depth-first search (DFS) of all reachable instructions. Two main enhancements over past symbolic execution algorithms enable verification of information flow properties in SoCs. First, the engine maintains *two* symbolic copies of the state of the program along the path that is being executed allowing us to test whether assigning different values to `src` results in different values at `dst`. The other enhancement is handling MMIO instructions using selective symbolic execution.

3.2.2 Example of Algorithm Execution: To understand the algorithm, consider its execution on the code shown in Figure 3.[1] The property states that the untrusted value `r1` must not influence the value of `IO_REG`. Suppose, due to a typo `N`=3 instead of the correct value 2. The symbolic state computed by the algorithm when it reaches the assignment to `IO_REG` would be as follows:

[1]We show the algorithm in C-like pseudocode to make understanding easier but the analysis is done on binary code.

```
#define N 3                  // should be 2
uint8_t tbl[] = { 1, 1 };    // address of tbl = 0x100
uint8_t data = 3;            // &data=0x102
uint8_t IO_REG = 1;          // &IO_REG=0x200.

void foo(int r1) {
    if (r1 < 0 || r1 >= N) return;
    IO_REG = tbl[r1];
}
```

Figure 3: Integrity property example: src=r1, dst=*dataout*, srcpred=true and dstpred=*memaddr* = 0x200 ∧ *memop* = WR.

$P^1 = \neg(x^1 < 0 \vee x^1 \geq 3)$	$P^2 = \neg(x^2 < 0 \vee x^2 \geq 3)$
$dataout^1 = \mathcal{M}^1[\texttt{0x100} + x^1]$	$dataout^2 = \mathcal{M}^2[\texttt{0x100} + x^2]$
$\mathcal{M}^1 = \mathcal{M}^2 = [\texttt{0x100} \mapsto 1, \texttt{0x101} \mapsto 1, \texttt{0x102} \mapsto 3, \ldots]$	
$memaddr^1 = memaddr^2 = \texttt{0x200}$	
$memop^1 = memop^2 = \mathsf{WR}$	

In the above, x^1 and x^2 are the new variables created to represent the untrusted value r1. For each variable, the superscripts 1 and 2 refer to the values of these variables in the corresponding "copies" of the program state. P^1 and P^2 are *path conditions* that determine the constraints under which this particular path is taken. $\mathcal{M}^1$ and $\mathcal{M}^2$ are represent memory state. $\mathcal{M}^1[\texttt{0x100}+x^1]$ refers to the result of reading address $\texttt{0x100}+x^1$ from the memory. *memaddr*, *memop*, *dataout* are respectively the address, type and data being written by the current memory operation.

When the solver evaluates whether $dataout^1 \neq dataout^2$, P^1, P^2, srcpred and dstpred are all satisfiable, it will find $x^1 = 1, x^2 = 2$ and report this error. Once we fix the bug and N=2, then $(P^1, P^2) = (x^1 \geq 0 \wedge x^1 < 2, x^2 \geq 0 \wedge x^2 < 2)$. Now it is not possible make $dataout^1 \neq dataout^2$ while satisfying P^1 and P^2, so the algorithm will not report in an error.

Now let us consider a different property. Suppose src is data, while dst and dstpred are the same as before. This property states that the secret value data must not influence untrusted register IO_REG. Clearly, a violation exists if N=3 and this is detected.

3.3 Evaluation

We evaluated our approach by examining part of the firmware of an upcoming commercial phone/tablet SoC. The SoC consists of a number of IPs for various functions such as display, camera, touch sensing, etc. This evaluation examined a single component IP, called the PTIP which is involved in security sensitive "flows" such as secure boot. It contains a proprietary 32-bit microcontroller which executes the firmware. The firmware interacts with the other IPs in the SoC through hardware registers accessed using MMIO.

3.3.1 Methodology: We synthesized an instruction-level abstraction (ILA) of the PTIP microcontroller and then used the ILA to generate a symbolic execution engine. Z3 v4.3.2 was the constraint solver [7] used. This symbolic execution engine was integrated with a pre-existing simulator for this microcontroller to model MMIO reads and writes to other parts of the SoC.

3.3.2 Security Objectives: The PTIP firmware interacts with system software, devices drivers and other untrusted IPs. Since these entities, especially the system software and drivers, may be compromised by malware, these are all untrusted. We explored two main security objectives as part of the evaluation. First, the PTIP memory holds a sensitive cryptographic key called the *IPKEY*. We verify that these untrusted entities cannot not access *IPKEY*. Second, we verify control-flow integrity of the PTIP firmware. Three representative information flow properties we formulated to capture these security requirements.

The total size of the PTIP firmware is approximately a few tens of thousands of static instructions. Due to limited time, this evaluation focused on a set of message handler functions which send and receive commands/messages from the (untrusted) system software, drivers and other IPs. The size of these handler functions was approximately several hundred static instructions.

3.3.3 Summary of Verification Results: In terms of scalability, the symbolic execution engine could explore up to about half a million instructions within the assigned time limit of 30 minutes. This was sufficient for exploring all possible paths in 4 out of 6 handlers examined in the evaluation. Full exploration of paths could not be completed for the other two handlers. While these results are promising and show that some real-world firmware can be examined, further improvements in scalability are likely possible with more sophisticated analysis techniques.

The PTIP firmware had previously undergone simulation-based testing and manual code review. However, we were still able to identify a tricky security bug that could lead to *IPKEY* exposure. Symbolic analysis involving reasoning over all possible input values was essential in helping discover this bug.

4 Discussion and Related Work

This section briefly describes other uses of the ILA and connections to related research.

4.1 Discussion

The ILA is a complete formal specification of hardware behavior that precisely defines the hardware/software interface. In this paper, we showed how the ILA can be used to verify security properties of firmware using symbolic execution. However, the ILA-based verification methodology is helpful in enabling many diverse design and verification tasks.

4.1.1 ILA-Based Design: Since the ILA precisely defines the firmware/hardware interface, it enables portability among SoCs with slightly different accelerator and I/O devices. Firmware code that targets one ILA can be "translated" into code that targets an accelerator that implements a different ILA.

Similarly, the ILA enables synthesis/design optimizations to make hardware changes "under-the-hood" while retaining firmware compatibility. For example, one chip generation might implement only one substep of an important routine in hardware, while the next chip generation might implement the entire important routine as a single hardware accelerator. The ILA enables us to make such changes with the guarantee that system-level security and functionality requirements are preserved.

4.1.2 ILA-based Verification: On the verification side, ILAs can verify equivalence of two accelerators that implement the same firmware-visible behavior. They also offer various avenues for formal analysis of system-level properties. In this paper, we described symbolic execution, but other techniques such as bounded and unbounded model checking as well as interactive theorem proving are also possible using an ILA. From the security point-of-view, this paper introduced a specification language for confidentiality and integrity properties. Extending the specification language to cover other security requirements such as availability and non-repudiation and using ILA-based verification to analyze these properties is another promising avenue for further research.

4.2 Related Work

There is a rich body of literature studying synthesis and verification. We survey some of the most closely related work below.

4.2.1 Synthesizing Abstractions: Our work builds on recent progress in syntax-guided synthesis which is surveyed in [1]. Our synthesis algorithm is based on oracle-guided synthesis from [12]. Our contribution is the use of synthesis for constructing abstractions of SoC hardware. Also related is the work of Godefroid *et al*. [10] They synthesize a model for a subset of the x86 ALU instructions using I/O samples. In comparison, our contributions are strong guarantees about the correctness of the synthesized abstraction and general abstractions for SoC hardware, not just ALU outputs.

4.2.2 SoC Verification: *Refinement relations*, used in proving the abstraction and the implementation match are from [13, 18]. One approach to compositional SoC verification is by Xie *et al*. [33, 34] They suggest manually constructing a "bridge" specification that along with a set of hardware properties can be used to verify software components that rely on these properties. Our methodology makes it easy to construct the equivalent of the bridge specifications. Most importantly, it ensures correctness of the abstraction.

4.2.3 Symbolic Execution and Taint Analysis: The DART and KLEE projects are the precursors of subsequent work in symbolic execution [4, 9]. They combined modern constraint solvers and dynamic analysis to generate tests for software programs. Subsequent projects, such as FIE and S^2E, have applied symbolic execution to firmware and low-level software [5, 6]. The most important difference between these frameworks and our work is that they only verify safety properties, *not* confidentiality and integrity.

A large body of work also studies dynamic taint analysis (DTA) [2, 14, 24, 26]. DTA suffers from both false positives and false negatives due to the problems of under- and over-tainting. Our work does not result in false positives. An overview of DTA and symbolic execution is presented in [24]. We show how symbolic execution can be used to verify information flow; this is missing from [24] which treats DTA and symbolic execution separately.

5 Conclusion

In this paper, we described a principled methodology for security verification of SoCs. The first component of the methodology is the construction of Instruction-Level Abstractions (ILAs) of SoC hardware components. The ILA of a hardware component is an abstraction that treats commands sent from firmware to the component as the equivalent of "instructions" and models all firmware-visible state updates due to these instructions. We described how ILAs can be semi-automatically synthesized and verified to be correct abstractions of hardware components.

The second component of the methodology is using the ILA for the verification of system-level security properties. We introduced a property specification language that can express requirements like confidentiality and integrity and an algorithm based on symbolic execution to verify these properties. Experimentally, we found that both components – ILA construction and verification using symbolic execution – helped find several bugs in both an SoC built out of open source components and parts of a commercial SoC design.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, 2013.

[2] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *Security and Cryptography*, 2013.

[3] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2014.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Operating Systems Design and Implementation*, 2008.

[5] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[6] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Conference on Security*, 2013.

[7] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[8] Experimental artifacts and synthesis framework source code. https://bitbucket.org/spramod/fmcad-15-soc-ila, 2015.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Programming Language Design and Implementation*, 2005.

[10] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Programming Language Design and Implementation*, 2012.

[11] H. Hsing. http://opencores.org/project,tiny_aes, 2014.

[12] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, 2010.

[13] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *Computer-Aided Verification*, 2001.

[14] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Network and Distributed System Security Symposium*, 2011.

[15] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor. Security of SoC firmware load protocols. In *Hardware-Oriented Security and Trust*, pages 70–75, 2014.

[16] R. Lysecky, T. Givargis, G. Stitt, A. Gordon-Ross, and K. Miller. http://www.cs.ucr.edu/~dalton/i8051/i8051sim/, 2001.

[17] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 79–93. IEEE, 1994.

[18] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods*. Springer, 2001.

[19] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Principles of Programming Languages*, 1999.

[20] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz. Formal Hardware/Software Co-verification by Interval Property Checking with Abstraction. In *Design Automation Conference*, 2011.

[21] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Selected Areas in Communications*, 2003.

[22] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[23] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov. System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, 94(6), 2006.

[24] E. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Security and Privacy*, 2010.

[25] R. Sinha, P. Roop, and S. Basu. The AMBA SOC Platform. In *Correct-by-Construction Approaches for SoC Design*. Springer New York, 2014.

[26] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, 2008.

[27] J. Strömbergson. https://github.com/secworks/sha1, 2014.

[28] P. Subramanyan and D. Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Design, Automation and Test in Europe*, 2014.

[29] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung. Verifying Verifying Information Flow Properties of Firmware using Symbolic Execution. In *Design Automation and Test in Europe*, 2016.

[30] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik. Template-based Synthesis of Instruction-Level Abstractions for SoC Verification. In *Formal Methods in Computer-Aided Design*, 2015.

[31] S. Teran and J. Simsic. http://opencores.org/project,8051, 2013.

[32] C. Wolf. http://www.clifford.at/yosys/, 2015.

[33] F. Xie, X. Song, H. Chung, and R. N. Translation-based Co-verification. In *Formal Methods and Models for Co-Design*, 2005.

[34] F. Xie, G. Yang, and X. Song. Component-based Hardware/Software Co-verification for Building Trustworthy Embedded Systems. *Journal of System Software*, 80(5), May 2007.