Chapter 13
Verifying Security Properties in Modern SoCs Using Instruction-Level Abstractions

Pramod Subramanyan and Sharad Malik

13.1 Introduction

In the era of Dennard-scaling, advances in computing performances were driven by transistor scaling. With each succeeding generation, transistors were smaller, faster, and more power-efficient than before [7, 17]. These gains were used to increase integrated circuit (IC) performance by designing more complex chips with a higher frequency of operation. The end of Dennard-scaling led to the multicore era where increases in performance were driven by parallelism rather than increased operating frequency [31]. However power and thermal constraints still limit performance and we are now in the dark silicon era where significant parts of an IC must be powered-off in order to stay within its power and thermal budgets [18].

Despite the technological limitations imposed by the dark silicon era, the demand for increased application performance and power-efficiency has not subsided and this has led to the rise of *accelerator-rich* system-on-chip (SoC) architectures [12]. Application-specific functionality is implemented using fixed-function or semi-programmable accelerators to obtain increased performance and power-efficiency. These accelerators are controlled and orchestrated by *firmware* which executes on programmable cores. Overall system functionality is implemented by this combination of firmware, programmable cores, and hardware components.

Figure 13.1 shows the structure of a modern SoC. It contains multiple programmable cores, accelerators, memories, and I/O devices connected by an on-chip interconnect [41, 44]. These components are organized into hierarchical modules, also referred to as intellectual property (IP) blocks. Typically each IP block contains a processor core, private memory, and several accelerators and I/O devices. Simpler IPs may contain only accelerators and/or I/O devices.

P. Subramanyan (✉) • S. Malik
Department of Electrical Engineering, Princeton University, Princeton, NJ, USA
e-mail: psubrama@princeton.edu; sharad@princeton.edu

© Springer International Publishing AG 2017
P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_13

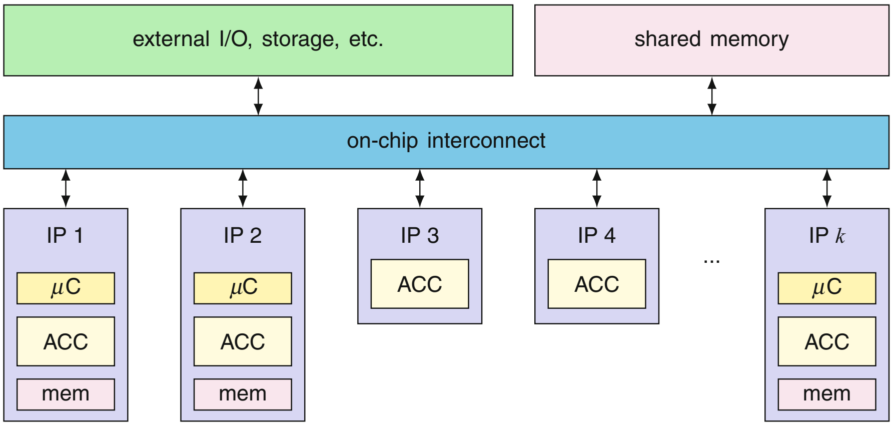


Fig. 13.1 Overview of a system-on-chip design

Due to time-to-market pressures imposed by the competitive environment, IP blocks are often obtained from semi-trusted or untrusted third-party vendors. This has two effects on the security verification problem: (1) the functionality of these blocks may not be fully or clearly specified (e.g., verification engineers may only have access to gate-level netlists instead of RTL designs) making verification more challenging, and (2) due to their untrusted nature, the designs may have subtle bugs which violate system-level security requirements of the SoC.

It is important to emphasize the role of firmware in modern SoC designs. Firmware is code that executes on programmable cores. It is shipped with the hardware and sits "below" the operating system and interacts closely with hardware accelerators, the on-chip interconnect, and I/O devices. System-level functionality including security features and protocols are *orchestrated* by firmware—firmware initiates operation of the hardware accelerators involved, collects and coordinates their responses, and manages resource acquisition and release.

13.1.1 Challenges in SoC Security Verification

The prevalence of firmware in today's SoCs poses an important challenge in SoC security verification. Both firmware and hardware make many assumptions about the other component. Since functionality is implemented by a combination of hardware and firmware, verifying these two components separately requires explicitly enumerating these assumptions and ensuring the other component satisfies them. Bugs may exist in hardware or firmware alone, but some bugs may also be caused by incorrect assumptions made by hardware/firmware about the other component.

13.1.1.1 Need for Hardware/Firmware Co-verification

As an illustrative example, consider the runtime binary authentication protocol discussed by Krstic et al. [30] 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 this is the case, 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 such that untrusted hardware components cannot write to locked pages, 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: (1) the need for verification that analyzes the hardware and firmware together and (2) the need for precise specification of the hardware/firmware interface so that assumptions made by either side about the other are explicit. One way of achieving these two objectives is formal verification of firmware along with the cycle-accurate and bit-precise register transfer level (RTL) model of SoC hardware. Unfortunately, this naïve approach does not work in practice: formal verification of the RTL description along with the firmware is not feasible even for very small SoCs due to scalability limitations of formal tools.

13.1.1.2 SoC Verification Through Abstraction

A general technique for making SoC verification tractable is to use an abstraction that accurately models all updates to firmware-accessible hardware states [24, 33, 38, 49, 54, 55]. When verifying properties involving firmware, the abstraction is used instead of the bit-precise cycle-accurate hardware model.

Although the idea of constructing abstractions for firmware verification is attractive, there are several challenges in applying this technique. Firmware interacts with hardware components in a myriad of ways. For the abstraction to be useful, it needs to model all interactions and capture all updates to firmware-accessible states.

- Firmware usually controls accelerators in the SoC by writing to memory-mapped registers within the accelerators. These registers may set the mode of operation of the accelerator, the location of the data to be processed, or return the current state of the accelerator's operation. The abstraction needs to model these "special" reads and writes to the memory-mapped I/O space correctly.
- Once operation is initiated, the accelerators step through a high-level state machine that implements the data processing functionality. Transitions of this state machine may depend on responses from other SoC components, the

acquisition of semaphores, external inputs, etc. These state machines have to be modeled to ensure there are no bugs involving race conditions or malicious external input that cause unexpected transitions or deadlocks.

- Another concern is preventing compromised/malicious firmware from accessing sensitive data. To prove that such requirements are satisfied, the abstraction needs to capture issues such as a sensitive value being copied into a firmware-accessible temporary register.

Completeness of the abstraction is very important for security verification as finding security vulnerabilities requires reasoning about *all* inputs and states of the system including invalid/illegal inputs. A specific example of this is given in [48] which 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 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.

Manual construction of abstractions which capture these details, as proposed, for example, in [54, 55], is not practical because it is error-prone, tedious, and time-consuming. Manual construction can be especially challenging for third-party IPs because RTL “source code” may not be available so the abstraction has to be “reverse-engineered” from a gate-level netlist. If the manually constructed abstraction is *incorrect*, i.e., the hardware implementation is not consistent with the abstraction, properties proven using it are not valid.

13.1.1.3 Challenges in Specifying Security Properties

Another challenge in security verification is property specification. Traditional property specification languages based on temporal logic cannot express security requirements involving information flow: e.g., confidentiality and integrity [34]. As a result, existing formal tools such as model checkers and symbolic execution engines cannot verify information flow properties.

Confidentiality and integrity can be intuitively specified using *information flow* properties. One method for verifying these properties is through dynamic taint analysis [3, 13, 14, 29, 42, 46]. Dynamic taint analysis (DTA) associates a “taint bit” with each object in the program/design. Taint propagation rules then set the taint bit of the output of a computation if its input(s) are also tainted. Confidentiality violations are detected by tainting the secret and raising an error when the taint propagates to an untrusted object. Integrity violations are detected by tainting untrusted objects and raising an error when the taint propagates to a trusted location.

While DTA enables intuitive property specification, it has deficiencies in the verification aspect. DTA can determine if the property has been violated given an instruction trace. However, since it is a dynamic analysis, it cannot be used to exhaustively search over the space of all possible instruction sequences to prove the absence of property violations. Secondly, creating taint propagation rules is challenging due to the problems of under- and over-tainting [3, 29]. Over-tainting can result in a deluge of false-positives [29] while under-tainting can miss bugs [42].

Static taint analysis can ensure information flow properties are satisfied in programs. However, current static taint analysis techniques are based on programming languages with secure type systems [37, 39]. These techniques cannot be applied on existing firmware because significant parts are written in assembly language. This necessitates analysis at the level of binary code rather than a high-level programming language. The above discussion points to the need for tools that can perform exhaustive analysis of information flow properties on binary code in the context of hardware/firmware co-verification.

13.1.2 SoC Security Verification Using Instruction-Level Abstractions

In this chapter, we introduce a principled methodology 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*) [49].

13.1.2.1 ILA Synthesis and Verification

To help easily construct the abstraction in a semi-automated manner, we build on recent progress in syntax-guided synthesis [2, 26, 27, 43]. 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.

The synthesis algorithm requires a simulator that can be used in the following manner: start simulation at a specified initial state, execute a given instruction, and return the state after the state update(s) corresponding to this instruction are carried out. This only requires minor modifications to an architectural, RTL, or gate-level simulator. The key advantages of ILA synthesis are that it significantly reduces manual effort in ILA construction and allows ILAs to be constructed easily and correctly even in the case of third-party IPs for which RTL descriptions may not be available.

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. If the model checking is completed, we have a strong guarantee that all properties proven using the ILA are valid.

13.1.2.2 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 when 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 of computation performed on 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 as the destination value depends on the source. This means the property is violated. If no such value can be found, the property holds along this particular path.

13.1.2.3 Summarizing ILA-Based Verification

An overview of the complete methodology is shown in Fig. 13.2. The methodology has two parts: (a) synthesis and verification of correctness of the ILA and (b) using the ILA to verify security properties. The rest of this chapter is organized as follows. The definition of the ILA, ILA synthesis, and verification of ILA correctness are described in Sect. 13.2. The security property specification language and use of the ILA for verification properties is described in Sect. 13.3. A discussion

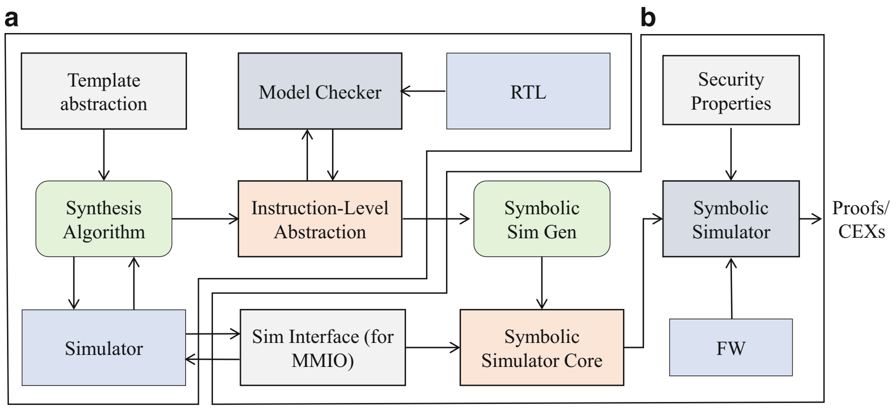


Fig. 13.2 Overview of ILA-based security verification methodology. (**a**) ILA synthesis and verification of correctness. (**b**) Security verification using ILAs

of potential extensions, limitations of the methodology, and related work is in Sect. 13.4. Section 13.5 provides concluding remarks.

13.2 Instruction-Level Abstractions

An instruction-level abstraction (ILA) is an abstraction that captures the firmware-visible hardware behavior in the context of an SoC. The key insight underlying the ILA is to model all firmware-visible state and associated state updates, and nothing more. Therefore, microarchitectural states, such as pipeline registers and reorder buffers, *are not* modeled in the ILA. However, all firmware-visible registers, including MMU state and on-chip network state *are* modeled in the ILA. This results in an abstraction which can be used for co-verification of hardware and firmware, but omits unnecessary detail that can cause scalability bottlenecks.

In the rest of this section, we first provide an intuitive overview of ILAs. We then formally define ILAs and describe how ILAs can be semi-automatically synthesized and verified to be correct abstractions of SoC hardware.

13.2.1 ILA Overview

Recall that the ILA is an abstraction of the firmware-visible behavior of a hardware module. Therefore, in the case of programmable cores, the ILA is the same as the instruction-set architecture (ISA) specification of the programmable core. An important aspect of the ILA that it is a formal, machine-readable abstraction that models all architectural state and associated state updates performed by the programmable core upon the execution of each instruction supported by it. Formal specifications of ISAs can be quite difficult to construct in practice [22, 23]. Our methodology mitigates this challenge by enabling *semi-automated synthesis* of ILAs.

The ILA models semi-programmable and fixed-function accelerators using an abstraction that is similar to an ISA. Accelerators in today's SoC designs are event-driven. Computation is performed in response to commands sent by programmable cores and is typically bounded in length. Our insight here is to view commands from the programmable cores to the accelerators as analogous to "instruction opcodes" and state updates in response to these commands as "instruction execution." The key insight here is to model accelerators using the same fetch/decode/execute sequence as a programmable core. The command is analogous to "fetch," the case-split determining how the command is processed is "decode," and the state update is instruction "execution."

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 [49] 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.

Imposing this structure on an abstraction for accelerators has two advantages. First, the ILA becomes a precise and formal specification of the HW/SW interface for accelerators. It can be used in various design tasks such as full-system simulation to support FW/SW development and system-level verification. Second, interactions of FW with accelerators can be modeled using well-understood instruction-interleaving semantics, enabling use of standard tools such as model checkers for verification. Third, verification of SoC hardware also becomes more tractable because conformance with an ILA can be done compositionally on a "per-instruction" basis leveraging the body of work in microprocessor verification [28, 35].

13.2.2 ILA Definition

We now provide a formal definition of an instruction-level abstraction.

13.2.2.1 Notation

Let $\mathbb{B} = \{0, 1\}$ be the Boolean domain and let $bvec_l$ denote all bitvectors of width l. $M_{k \times l} : bvec_k \mapsto bvec_l$ maps from bitvectors of width k to bitvectors of width l and represents memories of address width k bits and data width l bits. Booleans and bitvectors are used to model state registers while the memory variables model hardware structures like scratchpads and random access memories. A memory variable supports two operations: $read(mem, a)$ returns data stored at address a in memory mem while $write(mem, a, d)$ returns a new memory which is identical to mem except that address a maps to d, i.e., $read(write(mem, a, d), a) = d$.

13.2.2.2 Architectural State and Inputs

The architectural state variables of an ILA are modeled as Boolean, bitvector, or memory variables. As with ISAs, the architectural state refers to state that is persistent and visible across instructions.

Let S represent the vector of state variables of an ILA consisting of Boolean, bitvector, and memory variables. In an ILA for a microprocessor, S contains all the architectural registers, bit-level state (e.g., status flags), and data and instruction memories. In an accelerator, S contains all the software-visible registers and memory-structures. A state of an ILA is a valuation of the variables in S.

Let vector W represent the input variables of the ILA; these are Boolean and bitvector variables which model input ports of processors/accelerators.

Let $type_{S[i]}$ be the "type" of state variable $S[i]$; $type_{S[i]} = \mathbb{B}$ if $S[i]$ is Boolean, $type_{S[i]} = bvec_l$ if $S[i]$ is a bitvector of width l and $type_{S[i]} = M_{k \times l}$ if $S[i]$ is a memory.

13.2.2.3 Fetching an Instruction

The result of fetching an instruction is an "opcode." This is modeled by the function $F_o : (S \times W) \mapsto bvec_w$, where w is the width of the opcode. For instance, in the 8051 microcontroller, $F_o(S, W) \triangleq read(\text{IMEM}, \text{PC})$ where IMEM is the instruction memory and PC is the program counter.[1]

Programmable cores repeatedly fetch, decode, and execute instructions, i.e., they always "have" an instruction to execute. However, accelerators may be event-driven and execute an instruction only when a certain trigger occurs. This is modeled by the function $F_v : (S \times W) \mapsto \mathbb{B}$. Suppose an accelerator executes an instruction when either Cmd1Valid or Cmd2Valid is asserted, then $F_v(S, W) \triangleq \text{Cmd1Valid} \vee \text{Cmd2Valid}$.

13.2.2.4 Decoding an Instruction

Decoding an instruction involves examining an opcode and choosing the state update operation that will be performed. We represent the different choices by defining a set of functions $D = \{\delta_j \mid 1 \leq j \leq C\}$ for some constant C where each $\delta_j : bvec_w \mapsto \mathbb{B}$. Recall $F_o : (S \times W) \mapsto bvec_w$ is a function that returns the current opcode. Each δ_j is applied on the result of F_o. The functions δ_j must satisfy the condition: $\forall j, j' : j \neq j' \iff \neg(\delta_j \wedge \delta_{j'})$.

For convenience let us also define the predicate $\text{op}_j \triangleq \delta_j(F_o(S, W))$. When op_j is 1, it selects the jth instruction. For example, in the case of the 8051 microcontroller, $D = \{\delta_1(f) \triangleq f = 0, \delta_2(f) \triangleq f = 1, \ldots, \delta_{256}(f) \triangleq f = 255\}$.[2] Recall we had defined F_o for this microcontroller as $F_o(S, W) \triangleq read(\text{IMEM}, \text{PC})$. Therefore, $\text{op}_j \iff read(\text{IMEM}, \text{PC}) = j - 1$. We are "case-splitting" on each of the 256 values taken by the opcode and each of these possibly performs a different state update. The functions δ_j choose which of these updates is to be performed.

[1]Note $F_o(S, W)$ must contain the instruction opcode. It may also (but is not required to) contain additional data such as the arguments to the instruction.

[2]We are writing bitvectors of width 8 (elements of $bvec_8$) as $0 \ldots 255$.

13.2.2.5 Executing an Instruction

For each state element $S[i]$ define the function $N_j[i] : (S \times W) \mapsto type_{S[i]}$. $N_j[i]$ is the state update function for $S[i]$ when $\mathrm{op}_j = 1$. For example, in the 8051 microcontroller, opcode `0x4` increments the accumulator. Therefore, $N_4[\mathrm{ACC}] = \mathrm{ACC} + 1$. The complete next state function $N : (S \times W) \mapsto S$ is defined in terms of the functions $N_j[i]$ over all i and j.

Defining the next state function N compositionally in terms of the functions $N_j[i]$ has an important advantage: it enables compositional verification [28, 35]. The behavior of the RTL can now be verified separately for each state element $S[i]$ and for each opcode op_j. This results in a simpler verification problem and enables scalable verification of large designs.

13.2.2.6 Syntax

The language of expressions allowed in F_o, F_v, D, and $N_j[i]$ is shown in Fig. 13.3. Expressions are quantifier-free formulas over the theories of bitvectors, bitvector arrays, and uninterpreted functions. They can be of type Boolean, bitvector, or memory, and each of these has the usual variables, constants, and operators with

$\langle exp \rangle$ ::= $\langle bv\text{-}exp \rangle$ | $\langle bool\text{-}exp \rangle$ | $\langle mem\text{-}exp \rangle$ | $\langle func\text{-}exp \rangle$
  | $\langle choice\text{-}exp \rangle$

$\langle bv\text{-}exp \rangle$ ::= var $\langle id \rangle$ *width* | cnst `val` *width*
  | bvop $\langle exp \rangle$ ...
  | read $\langle memexp \rangle$ $\langle addrexp \rangle$
  | read-block $\langle memexp \rangle$ $\langle addrexp \rangle$
  | apply $\langle func \rangle$ $\langle bv\text{-}exp \rangle$ ...
  | **extract-bitslice** $\langle bv\text{-}exp \rangle$ *width*
  | **extract-subword** $\langle bv\text{-}exp \rangle$ *width*
  | **replace-bitslice** $\langle bv\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$
  | **replace-subword** $\langle bv\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$
  | **in-range** $\langle bv\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$

$\langle bool\text{-}exp \rangle$ ::= var | true | false
  | boolop $\langle exp \rangle$ ...

$\langle mem\text{-}exp \rangle$ ::= $\langle id \rangle$ |
  | write $\langle mem\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$
  | write-block $\langle mem\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$ $\langle bv\text{-}exp \rangle$

$\langle func\text{-}exp \rangle$ ::= func $\langle id \rangle$ $width_{out}$ $width_{in_1}$ ...

$\langle choice\text{-}exp \rangle$ ::= **choice** $\langle exp \rangle$ $\langle exp \rangle$ ...

Fig. 13.3 Syntax for expressions

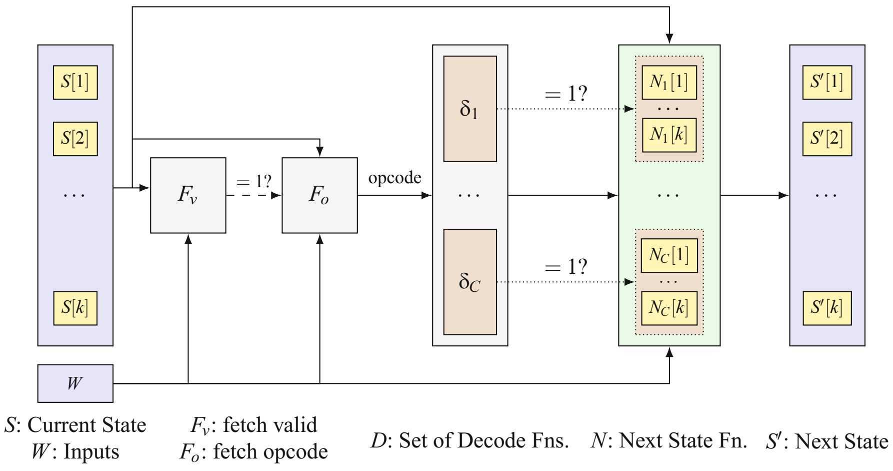


Fig. 13.4 Pictorial overview of the ILA definition

standard interpretations. The synthesis primitives, shown in **bold**, will be described in Sect. 13.2.3.

13.2.2.7 Putting It All Together

To summarize, an instruction-level abstraction (ILA) is the tuple: A $= \langle S, W, F_o, F_v, D, N \rangle$. S and W are the state and input variables. F_o, F_v, D, and N are the fetch, decode, and next state functions, respectively.

A pictorial overview of this definition is shown in Fig. 13.4. The leftmost box shows the state vector S. The function F_v ("fetch valid") examines the state vector S and input vector W and determines if there is an instruction to execute. If this is the case, i.e., if $F_v(S, W) = 1$, then $F_o(S)$ is used to get the opcode from the current state and input vectors. This opcode is used to evaluate each of the functions $\delta_1, \delta_2, \ldots, \delta_j, \ldots, \delta_C$. If $\delta_j = 1$, then $N_j[i]$ is evaluated for each $S[i]$ to get the next state $S'[i]$. The vector S' then defines the next state of the ILA and the above process is repeated again from this state to execute the next "instruction."

13.2.3 ILA Synthesis

For ILAs to be useful, one must be able to generate them correctly and preferably automatically. Due to the prevalence of third-party IPs, SoC hardware blocks often exist *before* the ILA is constructed, and so ILAs need to be constructed *post hoc* without IP designer support. To address the error-prone and tedious aspects of this construction, we have developed an algorithm for *template-based synthesis* of ILAs.

To semi-automatically synthesize the functions $N_j[i]$ in the ILA, we build on the counterexample guided inductive synthesis (CEGIS) algorithm [26, 27]. We assume availability of a simulator that models state updates performed by the hardware. This simulator can be gate-level, RTL, or a high-level C/C++/SystemC simulator of the design; it is only used as a black-box. Its implementation does not matter except that it be possible to set the initial state, execute an instruction, and read out the final state after execution.

13.2.3.1 Notation and Problem Statement

Let $Sim : (S \times W) \mapsto S$ be the I/O oracle for the next state function N. Define Sim_i as the function that projects the state element $S[i]$ from Sim; $Sim_i : (S \times W) \mapsto type_{S[i]}$. In order to help synthesize the function implemented by Sim_i, the SoC designers write a *template next state function*, denoted by $\mathscr{T}_i : (\Phi \times S \times W) \mapsto type_{S[i]}$.

Φ is a set of *synthesis* variables, also referred to as "holes" [45], and different assignments to (interpretations of) Φ result in different next state functions. Unlike $N[i]$, $\mathscr{T}_i$ is a partial description and is therefore easier to write. It can omit certain low-level details, such as the mapping between individual opcodes and operations, opcode bits and source and destination registers, etc. These details are filled-in by the counterexample guided inductive synthesis (CEGIS) algorithm [26, 27] by observing the output of Sim_i for carefully selected values of (S, W).

(Problem Statement: ILA Synthesis): For each state element $S[i]$ and each op_j, find an interpretation of Φ, $\Phi_j[i]$, such that $\forall S, W : op_j \implies (\mathscr{T}_i(\Phi_j[i], S, W) = Sim_i(S, W)))$.

The synthesis procedure is repeated for each instruction (each j) and each state element (each i), and the synthesis result $\Phi_j[i]$ is an interpretation of Φ for i and j.

13.2.3.2 Template Language

The synthesis primitives in the currently implemented template language are shown in bold in Fig. 13.3. It is important to emphasize that our algorithm and methodology are not dependent on the specific synthesis primitives used. The only requirement placed on the primitives is that it be possible to "compile" them to some quantifier-free formula over bitvectors, bitvector arrays, and uninterpreted functions.

Synthesis Primitives

The expression **choice** ε_1 ε_2 asks the synthesizer to replace the **choice** primitive with either ε_1 or ε_2 based on simulation result. **choice** ε_1 ε_2 is translated to the formula $\text{ITE}(\phi_b, \varepsilon_1, \varepsilon_2)$ where $\phi_b \in \Phi$ is a new Boolean variable associated with this instance of the choice primitive. Its value is determined by the synthesis procedure.

The primitives **extract-slice** and **extract-subword** synthesize bitvector extract operators. The synthesis process determines the indices to be extracted from. These operators are used to extract bit-fields from a register. The advantage of using these operators instead of an extract operator is that the indices of the bitfield need not be specified. This is advantageous for two reasons: (1) it reduces manual effort when constructing the ILA and (2) it also eliminates errors in specifying indices, as these are now automatically determined by the synthesis procedure. The **replace-slice** and **replace-subword** are the counterparts of these primitives; they replace a part of the bitvector with an argument expression.

The primitive **in-range** synthesizes a bitvector constant that is within the specified range. We note that adding new synthesis primitives is easy and straightforward in our language.

13.2.3.3 An Illustrative Example

We illustrate the definition of an ILA and template next state function using the processor shown in Fig. 13.5. The instruction to be executed is read from the ROM. Its operands can either be an immediate from the ROM or from the 4-entry register file. For simplicity, we assume that the only two operations supported by the processor are addition and subtraction.

The architectural state for the processor is $S = \{\text{ROM}, \text{PC}, \text{R}_0, \text{R}_1, \text{R}_2, \text{R}_3\}$ and input set W is empty. $type_{\text{ROM}} = M_{8\times8}$ while all the other variables are of type

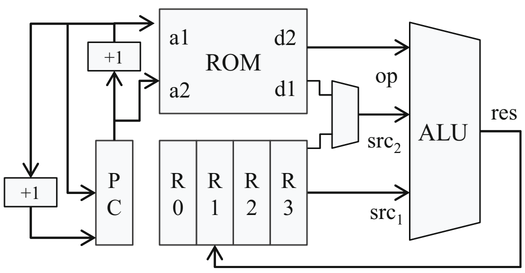


Fig. 13.5 A simple processor for illustration

$bvec_8$. The opcode which determines the next instruction is stored in the ROM and so $F_o \triangleq read(\text{ROM}, \text{PC})$; $F_v \triangleq true$. $D = \{\delta_j \mid 1 \leq j \leq 256\}$ where each $\delta_j(f) \triangleq f = j - 1$. The template next state functions, $\mathscr{T}_{\text{PC}}$ and $\mathscr{T}_{\text{R}_\text{i}}$, are as follows.

$$\begin{aligned}
\mathscr{T}_{\text{PC}} &= \textbf{choice } (\text{PC} + 1)\ (\text{PC} + 2)\\
imm &= read(\text{ROM}, \text{PC} + 1)\\
src_1 &= \textbf{choice } \text{R}_0\ \text{R}_1\ \text{R}_2\ \text{R}_3\\
src_2 &= \textbf{choice } \text{R}_0\ \text{R}_1\ \text{R}_2\ \text{R}_3\ imm\\
res &= \textbf{choice } (src_1 + src_2)\ (src_1 - src_2)\\
\mathscr{T}_{\text{R}_\text{i}} &= \textbf{choice } res\ R_i \quad (0 \leq i \leq 3)
\end{aligned}$$

13.2.3.4 Synthesis Algorithm

The counterexample-guided inductive synthesis (CEGIS) algorithm from [49] is shown in Algorithm 1. The inputs to the algorithm are the following.

1. op_j which determines the opcode for which synthesis is performed,
2. The template next state function $\mathscr{T}_i$.
3. The simulator Sim_i. The suffix i here denotes that the value of state element $S[i]$ is extracted from the output of the simulator.
4. A is a conjunction of assumptions under which synthesis is to be carried out. For example, suppose the opcode is $read(\text{ROM}, \text{PC})$, and functionality is only defined for opcodes in the range `0x0` to `0xF0`, then the assumption A would express this as $A \triangleq read(\text{ROM}, \text{PC}) \geq \texttt{0x0} \wedge read(\text{ROM}, \text{PC}) \leq \texttt{0xF0}$.

The algorithm is executed for each op_j and each $S[i]$. In each case it returns $\Phi_j[i]$ which is used to compute the next state function as $N_j[i](S, W) = \mathscr{T}_i(\Phi_j[i], S, W)$.

Algorithm 1 Synthesis algorithm

```
1:  function SYNCEGIS($op_j, \mathscr{T}_i, Sim_i, A$)
2:      $k \leftarrow 1$
3:      $R_1 \leftarrow A \wedge op_j \wedge (\theta \leftrightarrow (\mathscr{T}_i(\Phi_1, S, W) \neq \mathscr{T}_i(\Phi_2, S, W)))$
4:      while $sat(R_k \wedge \theta)$ do
5:          $\Delta \leftarrow \textsc{Model}_{(S,W)}(R_k \wedge \theta)$                ▷ get dist. input $\Delta$
6:          $O \leftarrow Sim_i(\Delta)$                                              ▷ simulate $\Delta$
7:          $O_1 \leftarrow (\mathscr{T}_i(\Phi_1, \Delta) = O)$
8:          $O_2 \leftarrow (\mathscr{T}_i(\Phi_2, \Delta) = O)$
9:          $R_{k+1} \leftarrow R_k \wedge O_1 \wedge O_2$                            ▷ enforce output $O$ for $\Delta$
10:         $k \leftarrow k + 1$
11:     end while
12:     if $sat(R_k \wedge \neg\theta)$ then
13:         return $\textsc{Model}_{\Phi_1}(R_k \wedge \neg\theta)$
14:     end if
15:     return $\bot$
16: end function
```

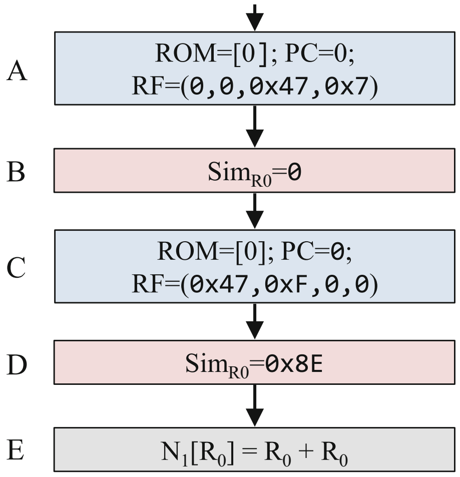


Fig. 13.6 Distinguishing inputs for the illustrative example. Notation $ROM = [0]$ means all ROM addresses contain the value 0

To understand the algorithm, observe that $\mathscr{T}_i(\Phi, S, W)$ defines a *family* of next state functions. Different functions are selected by different interpretations of Φ. The algorithm tries to find an interpretation of (S, W), say Δ, which for some two interpretations of Φ, $\Phi_1[i]$ and $\Phi_2[i]$ is such that $\mathscr{T}_i(\Phi_1[i], \Delta) \neq \mathscr{T}_i(\Phi_2[i], \Delta)$. Δ is called a *distinguishing input*. Once Δ is found, the simulation oracle Sim_i can be evaluated to find the expected output for this scenario. We then add constraints enforcing this input/output relation and try to find another distinguishing input. This process repeats until no more distinguishing inputs can be found. When this happens, it means that all remaining interpretations of Φ result in the same next state function, and any such interpretation is the solution.

Consider the computation of $N_1[\mathrm{R}_0]$ shown in Fig. 13.6. We are constructing the next state function for R_0 for opcode $\mathrm{op}_1 \iff read(\mathrm{ROM}, \mathrm{PC}) = \texttt{0x0}$. The first distinguishing input computed is node A in Fig. 13.6. This distinguishes between next state functions like $\mathrm{R}_0+\mathrm{R}_2$, $\mathrm{R}_0+\mathrm{R}_3$ and functions like $\mathrm{R}_0+\mathrm{R}_0$. Node B shows the output from the simulator for A is `0x0` ruling out $\mathrm{R}_0 + \mathrm{R}_2$ and $\mathrm{R}_0 + \mathrm{R}_3$. We now compute distinguishing input C. This input distinguishes among the functions $\mathrm{R}_0+\mathrm{R}_0$, $\mathrm{R}_0-\mathrm{R}_0$, $\mathrm{R}_0+\mathrm{R}_1$, $\mathrm{R}_0-\mathrm{R}_1$, and so on. When this input is simulated, it results in output D. At this point, we have a unique next state function: $N_1[\mathrm{R}_0] = \mathrm{R}_0 + \mathrm{R}_0$ and the algorithm terminates.

13.2.4 ILA Verification

Once we have ILA, the next step is to verify that it correctly abstracts the hardware implementation. Our first attempt at this might be to consider the ILA and the RTL as finite state transition systems executing in parallel and verify properties of the form $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$. This property says that state variable x_{ILA} in the ILA is equal in every time step[3] to the state variable x_{RTL} in the RTL. Unfortunately, this property is likely to be false for most state variables in hardware designs.

For example, consider a pipelined microprocessor with branch prediction. In this case, the processor may mispredict a branch and execute "wrong-path" instructions. Although these instructions will eventually be flushed, while they are being executed registers in the RTL will contain the results of these "wrong-path" instructions and so x_{RTL} will not match x_{ILA}.

13.2.4.1 Verifying Abstraction Correctness

When considering the internal state of the hardware components, we verify the ILA by defining *refinement relations* as proposed by McMillan [35]: $\mathbf{G}(\text{cond}_{ij} \implies x_{\text{ILA}} = x_{\text{RTL}})$. The predicate cond_{ij} specifies when the equivalence between state in the ILA and the corresponding state in the implementation holds. For example, in a pipelined microprocessor, we might expect that when an instruction commits, the architectural state of the implementation matches the ILA. Note the suffix ij here denotes that the refinement relation may be different for each state element $S[i]$ and each opcode op_j, i.e., the state update of the RTL may occur in a different cycle for each combination of $S[i]$ and op_j.

Defining the refinement relations as above allows compositional verification [28]. Consider the property $\neg(\phi\ \mathbf{U}\ (\text{cond}_{ij} \wedge (x_{\text{ILA}} \neq x_{\text{RTL}})))$ where ϕ states that all refinement relations hold until time $t-1$. This is equivalent to the above property, but we can abstract away irrelevant parts of ϕ when proving equivalence of x_{ILA} and x_{RTL}. For example, when considering op_j, we can abstract away the implementation of other opcodes $\text{op}_{j'}$ and assume these are implemented correctly. This simplifies the model but these proofs are still valid because we are still considering the correctness of all opcodes, albeit separately.

For state variables that are outputs of hardware components being modeled, we expect that ILA outputs always match the implementation. In this case, the property is $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$.

13.2.4.2 Discussion of Verification Issues

One part of our case study is a pipelined microcontroller with limited speculative execution. Our refinement relations are of the form $\mathbf{G}(\textit{inst\_finished} \implies (x_{\text{ILA}} = x_{\text{RTL}}))$, i.e., the state of the ILA and implementation must match when each instruction commits. The other part involves the verification of two cryptographic accelerators. Here the refinement relations are of the following form: $\mathbf{G}(\textit{hlsm\_state\_changed} \implies x_{\text{ILA}} = x_{\text{RTL}})$. The predicate *hlsm_state_changed* is

[3]This is the meaning of the **G** linear temporal logic (LTL) operator.

equal to 1 whenever the high-level state machine in the accelerator changes state. This refinement relation states that the high-level state machines of the ILA and RTL have the same transitions. The RTL state machine has some "low-level" states. These states do not exist in the ILA and are not visible to the firmware.

If we had to verify a superscalar processor, the ILA would execute multiple instructions in each transition. The exact number of instructions to be executed with each transition is an output of the implementation and an input to the abstraction. The property would state that after these many instructions are executed, the states of the ILA and implementation match.

Propagating auxiliary execution information from the implementation to the abstraction can help verify many complex scenarios. Consider the verification of weak memory models [1] where loads and stores in a microprocessor can be reordered in (well-defined) ways that affect program semantics. The information about the order in which load/store instructions are executed will be an output of the implementation and an input to the ILA. The refinement relations would verify that the execution of the ILA matches the implementation assuming instructions are executed in the same order.

13.2.4.3 Verification Correctness

If we prove the refinement relations for all outputs of the ILA and implementation: $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$, then we know that the ILA and implementation have identical externally visible behavior. Hence any properties proven about the behavior of the external inputs and outputs of the ILA are also valid for the implementation.

In practice, proving the property $\mathbf{G}(x_{\text{ILA}} = x_{\text{RTL}})$ for all external outputs may not be scalable, so we adopt McMillan's compositional approach. We prove refinement relations of the form $\neg(\phi \ \mathbf{U} \ (\text{cond}_{ij} \wedge x_{\text{ILA}} \neq x_{\text{RTL}}))$ for internal state and use these to prove the equivalence of the outputs.

If these compositional refinement relations are proven for all firmware-visible state in the ILA and implementation, then we know that all firmware-visible state updates are equivalent between the ILA and the implementation. Further, we know that transitions of the high-level of state machines in the ILA are equivalent to those in the implementation. These properties guarantee that firmware/hardware interactions in the ILA are equivalent to the implementation, ensuring correctness of the abstraction.

13.2.5 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.

13.2.5.1 Methodology

The template-based synthesis framework was implemented as a Python library using the Z3 SMT solver [16]. A modified version of Yosys was used to synthesize netlists from behavioral Verilog [53]. We used ABC for model checking [5]. The synthesis framework, template abstractions, synthesized ILA, and other experimental artifacts are available online [19].

13.2.5.2 Example SoC Structure

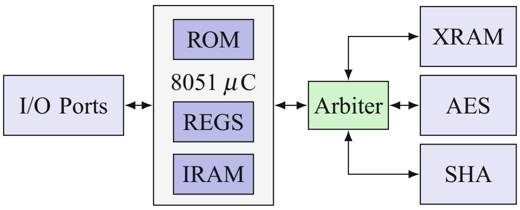
Experiments were conducted on an example SoC constructed from open source components containing the 8051 microcontroller and two cryptographic accelerators (Fig. 13.7). One accelerator implements encryption/decryption using the Advanced Encryption Standard (AES) while the other implements the SHA-1 cryptographic hash function [25, 47]. The RTL description of the 8051 is from OpenCores.org [51]. We also used *i8051sim* for instruction-level simulations of the 8051 [32].[4]

The firmware running on the 8051 initiates operation of the accelerators by writing the addresses of the data to be encrypted/decrypted/hashed to memory-mapped registers within the accelerators. Operation is started by writing to the start register which is also memory-mapped. Once the operation is started, the accelerators use direct memory access (DMA) to fetch the data from the external memory (XRAM), perform the operation, and write the result back to XRAM. The processor determines completion by polling a memory-mapped status register.

13.2.5.3 Summary of Synthesis Results

We constructed two ILAs: one for the 8051 microcontroller and another for the arbiter, XRAM, AES, and SHA modules. The insight here is that the 8051 communicates with the accelerators and XRAM by reading/writing to XRAM addresses. So from the perspective of the 8051, it is sufficient to show that all instructions that

Fig. 13.7 Example SoC block diagram



[4]We could have used the Verilog implementation itself for simulation. However, we chose to replicate the common scenario where simulators are developed and used for validation before the RTL design is complete.

modify the internal state of the 8051 are executed correctly and instructions which read/write XRAM produce the correct results at the external memory interface. What happens after these instructions "leave" the external memory interface—whether they modify the XRAM or start AES encryption, or return the current state of the SHA accelerator—need not be considered in this model. For the accelerators and the XRAM, we construct a separate ILA and the only instructions we need to consider here are reads and writes to XRAM addresses. In this ILA, we verify that these operations produce the expected results.

As an indication of the effort involved in building the model, the size of the template ILA, the simulator, and the RTL implementations for the two ILAs is shown in Table 13.1. The table shows that the template ILA is typically much smaller than the RTL. These numbers also demonstrate the template ILA can be written with relatively less effort.

Execution times of the synthesis algorithm for the 8051 ILA are shown in Table 13.2. We report the average and maximum values over all 256 opcodes. Except for the internal RAM, all other elements are synthesized with a few seconds. ILA synthesis found five bugs in *i8051sim*.

13.2.5.4 Typical ILAs

Tables 13.3 and 13.4 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. Each of these instructions

Table 13.1 Lines of code (LoC) and size in bytes of each model for the 8051 ILA

	8051		AES+SHA+XRAM	
Model	LoC	Size (KB)	LoC	Size (KB)
Template ILA	$\approx$ 650	30	$\approx$ 500	26
Instruction-level simulator	$\approx$ 3000	106	$\approx$ 400	14
Behavioral verilog implementation	$\approx$ 9600	360	$\approx$ 2800	87

Table 13.2 Synthesis execution time for 8051 ILA

	AVG	MAX		AVG	MAX
State	Time (s)		State	Time (s)	
ACC	4.3	8.5	B	3.6	5.1
DPH	2.7	5.0	DPL	2.6	4.4
IRAM	1245.7	14043.6	P0	1.8	2.7
P1	2.4	3.8	P2	2.2	3.5
P3	2.7	4.6	PC	6.3	141.2
PSW	7.3	15.9	SP	2.8	5.0
XRAM/addr	0.4	0.4	XRAM/dataout	0.3	0.4

Table 13.3 ILA instructions for AES accelerator

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 13.4 ILA instructions for SHA1 accelerator

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

is "triggered" by a memory-mapped I/O write to an appropriate address; F_v is of following form $F_v \triangleq ((\text{memop} = \texttt{WR} \lor \text{memop} = \texttt{RD}) \land \text{mem\_addr} = \texttt{REG\_ADDR})$ where `REG_ADDR` is the address of the corresponding configuration register.

13.2.5.5 Summary of Verification Results

We generated Verilog "golden models" from the ILAs and defined a set of refinement relations specifying that the golden models were equivalent to the RTL. We then used bounded and unbounded model checking to verify these refinement relations. These verification results are described below.

Verification of the 8051 ILA

We first attempted to verify the 8051 by generating a large monolithic golden model that implemented the entire functionality of the processor in a single cycle. The IRAM in this model was abstracted from a size of 256 bytes to 16 bytes. This abstracted golden model was generated automatically using the synthesis library. We manually implemented the abstraction reducing the size of the IRAM in the RTL implementation.

We used this golden model to verify properties of the form $\mathbf{G}(\textit{inst\_finished} \implies x_{\text{ILA}} = x_{\text{RTL}})$. For the external outputs of the processor, e.g., the external RAM address and data outputs, the properties were of the form $\mathbf{G}(\textit{output\_valid} \implies$

Table 13.5 Results with per-instruction golden model

Property	BMC bounds					Proofs
	CEX	≤ 20	≤ 25	≤ 30	≤ 35	
PC	0	0	25	10	204	96
ACC	1	0	8	39	191	56
IRAM	0	0	10	36	193	1
XRAM/dataout	0	0	0	0	239	238
XRAM/addr	0	0	0	0	239	239

$x_{\text{ILA}} = x_{\text{RTL}}$). Verification was done using bounded model checking (BMC) with ABC using the `bmc3` command. After fixing some bugs and disabling the remaining (17) buggy instructions, we were able to reach a bound of 17 cycles after 5 h of execution.

To improve scalability, we generated a set of "per-instruction" golden models which only implement the state updates for one of the 256 opcodes, the implementation of the other 255 opcodes is abstracted away. We then verified a set of properties of the form: $\neg(\phi\ \mathbf{U}(\mathit{inst\_finished} \wedge \text{op}_j \wedge x_{\text{ILA}} \neq x_{\text{RTL}}))$. Here ϕ states that all architectural state matches until time $t - 1$. We then attempted to verify five important properties stating that: (1) PC, (2) accumulator, (3) the IRAM, (4) XRAM data output, and (5) XRAM address must be equal for the golden model and the implementation.

Results for these verification experiments are shown in Table 13.5. Each row of the table corresponds to a particular property. Columns 2–6 show the bounds reached by BMC within 2000 s. For example, the first row shows that for 25 instructions, the BMC was able to reach a bound between 21 and 25 cycles without a counterexample; for 10 instructions, it achieved a bound between 26 and 30 cycles, and for the remaining 204 instructions, BMC reached a bound between 31 and 35 cycles. The last column shows the number of instructions for which we could prove the property. These proofs were done using the `pdr` command which implements the IC3 algorithm [8] with a time limit of 1950 s. Before running `pdr`, we preprocessed the netlists using the gate-level abstraction [36] technique with a time limit of 450 s.

Bugs Found During 8051 Verification

In the simulator, we found five bugs in total. Bugs in `CJNE`, `DA`, and `DIV` instructions were due to signed integers being used where unsigned values were expected. Another was a typo in `AJMP` and the last was a mismatch between RTL and the simulator when dividing by zero. These bugs were found during synthesis.

An interesting bug in the template was for the `POP` instruction. The `POP <operand>` instruction updates two items of state: (1) `<operand> = RAM[SP]` and (2) `SP = SP - 1`. But what if operand is `SP`? The RTL set `SP` using (1) while the ILA used (2). This was discovered during model checking and the ILA was changed to match the RTL. This shows one of the benefits of our methodology: all state updates are precisely defined and consistent between the ILA and RTL.

In the RTL model, we found a total of 7+1 bugs. One of these is an entire class of bugs related to the forwarding of special function register (SFR) values from an in-flight instruction to its successor. This affects 17 different instructions and all bit-addressable architectural state. We partially fixed this. A complete fix appears to require significant effort.

Another interesting issue was due to reads from reserved/undefined SFR addresses. The RTL returned the previous value stored in a temporary buffer. This is an example of the methodology detecting and preventing unintended leakage of information through undefined state.

Verifying the XRAM+AES+SHA ILA

We generated a Verilog golden model that combined the ILAs for the XRAM, AES, and SHA modules. We reduced the size of the XRAM in the ILA and the implementation to just one byte because we were not looking to prove correctness of reads and writes to the XRAM. We then attempted to prove a set of properties of the form $\mathbf{G}(\mathit{hlsm\_state\_change} \implies (x_{\mathrm{ILA}} = x_{\mathrm{RTL}}))$. We were able to prove that the *AES:State*, *AES:Addr*, and *AES:Len* in the implementation matched the ILA using the `pdr` command. For other firmware-visible state, BMC found no property violation up to 199 cycles with a time limit of 1 h.

13.3 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 the system and threat model. We then describe a specification language for firmware security properties and briefly provide an overview of an algorithm based on symbolic execution for verifying these properties.

13.3.1 System and Threat Model Overview

This section provides a brief overview of the system model and threat model considered in this work.

13.3.1.1 System-On-Chip Model

As described in Sect. 13.1 and shown in Fig. 13.1, we consider SoC designs consisting of a set of interacting *IPs*, a shared I/O space, an on-chip interconnect and possibly a shared memory. Each IP consists of a microcontroller, specialized hardware components, and private read-only and read-write memories (i.e., ROMs and RAMs). The firmware executes on the microcontroller interacts with the specialized hardware in its own IP as well as other IPs through memory-mapped I/O (MMIO).

The firmware's view of the system consists of a set of architectural registers, a special register known as the program counter, an instruction ROM which contains the firmware, and a data RAM which is used by the firmware during execution. We consider single-threaded firmware, however, other hardware and firmware interactions are also modeled and execute concurrently with the firmware thread. We use the ILA of the microcontroller to model the state updates performed by each instruction in the firmware.

13.3.1.2 Threat Model

The verification problem is tackled in a modular manner. Each IP is verified separately and the threat model is defined from the perspective of the individual IPs.

For each IP we identify the other IPs which are its *trust boundary*. The trust boundary is the subset of other IPs this IP fully trusts. For example, an IP involved in security critical functionality such as secure boot will likely *not trust* the camera and GPS IPs. Therefore, these IPs *will be outside* its trust boundary and any inputs it receives from these IPs are untrusted. Keeping the trust boundary small helps keep the attack surface small.

13.3.1.3 Security Objectives

The two classes of security objectives we consider are *confidentiality* and *integrity* of firmware assets. We wish to keep firmware secrets, e.g., encryption keys, confidential from untrusted IPs. Similarly, we wish to preserve the integrity of firmware assets. For example, we wish to ensure the integrity of firmware control-flow by ruling out stack smashing/buffer overflow attacks in the presence of arbitrary inputs from untrusted IPs. We also wish to ensure that untrusted IPs cannot modify the value of sensitive control/configuration registers, such as memory protection configuration registers. In this work, we do not consider other security requirements like availability and side channel attacks.

13.3.1.4 Modelling the Attacker

We assume that memory, I/O, and hardware registers controlled by untrusted IPs contain arbitrary values. In other words, outputs of an untrusted IP are *unconstrained* so reads from these locations return arbitrary values. Similarly, untrusted IPs may send invalid commands in an attempt to exploit, for example, buffer overflow bugs.

13.3.2 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 [34].

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 [40]. 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. An `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.

(continued)

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.

13.3.3 Firmware Execution Model

We now describe firmware state and the execution model.

13.3.3.1 Execution State

Firmware state is modeled using an ILA of the microcontroller hardware: A $= \langle S, W, F_o, F_v, D, N \rangle$. The firmware state S is assumed to contain the following special elements:

1. A set of bit-vector variables $S.mem = \{S.memop, S.memaddr, S.datain, S.dataout\}$ which contain information about the current memory operation. $S.memop =$ NOP means that the current instruction does not read/write from memory or memory-mapped I/O, $S.memop =$ RD denotes a read and WR is a write. $S.memaddr$ is the address of the current memory operation, $S.datain$ is the data read *from* memory (or I/O), and $S.dataout$ is the data being written to memory or I/O. These variables help model memory-mapped I/O and accesses to untrusted memory.
2. $S.ROM$ contains the read-only memory which contains the firmware to be executed. We assume that the instruction and data memories are separate.
3. The set $S.regs$ which contains all the remaining state variables of the ILA. $S.regs$ contains the special element $S.PC$ which refers to the program counter of the microcontroller.

The initial state of the microcontroller, i.e., the state from which execution starts is defined by the symbolic expression $init_S$. N is the next state function, and N_{mem} and N_{regs} are the projections of N to the sets $S.mem$ and $S.regs$, respectively.

13.3.3.2 A Review of Symbolic Execution

Our verification algorithm builds on symbolic execution using constraint solvers [9, 21]. In this section, we provide a brief overview of these techniques. Section 13.3.4 will describe the extensions required to verify information flow properties.

```
 1  void foo(int a, int b) {
 2      int c;
 3      if (a < 0 || b < 0)
 4          c = 0;
 5      else
 6          c = a+b;
 7  
 8      assert (c >= a && c >= b);
 9      return c;
10  }
```

Listing 13.1 Example code to demonstrate symbolic execution

Consider the code shown in Listing 13.1. To make understanding the algorithm easier, the code is shown in a C-like language but symbolic execution is actually performed on the equivalent binary code. For simplicity, let us assume that the line numbers shown in the listing correspond to program counter (PC) values. The following steps outline how a typical symbolic execution algorithm verifies the correctness of the assertions on line 8.

Execution starts in the symbolic state $init_S \triangleq S.PC = 3$. This means the initial state constrains the value of $S.PC$ to be 3 but all other variables/registers/memory values are unconstrained, i.e., they can take arbitrary values. The algorithm uses a constraint solver to enumerate all concrete values of $S.PC$ consistent with this symbolic state. In this case, this is just $S.PC = 3$. The conjunction of $S.PC = 3$ and $init_S$ is pushed onto a stack. This stack contains all paths in the program and associated symbolic states that need to be explored by the algorithm.

Symbolic Execution Main Loop

The main loop of the symbolic execution algorithm starts by popping the symbolic state at the top of the stack. Note symbolic states in the stack always have a specific (concrete) value for $S.PC$, so we know what instruction is to be executed next. This instruction is retrieved from the program memory and is executed symbolically, which means that we create symbolic expressions corresponding to the state update for each register and memory value in the program.

In our example, initially $S.PC = 3$, this instruction is a conditional branch which updates the PC according to the following expression: $ite(a < 0 \vee b < 0, 4, 6)$. In the above formula, ite is the if-then-else operator and the formula states that the next PC value will be 4 if $a < 0$ or $b < 0$, and 6 otherwise. All other variables and memory states are left unchanged; they retain the same values as in $init_S$.

Now the algorithm uses a constraint solver to evaluate all values of $S.PC$ consistent with these next state expressions. This gives us $S.PC = 4$ and $S.PC = 6$. The conjunction of each of these, $init_S$, and a *path condition* is now pushed onto

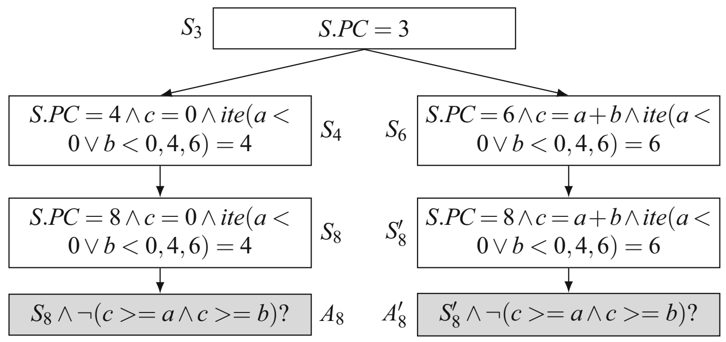

S_3 $S.PC = 3$
$S.PC = 4 \wedge c = 0 \wedge ite(a < 0 \vee b < 0, 4, 6) = 4$ S_4
S_6 $S.PC = 6 \wedge c = a + b \wedge ite(a < 0 \vee b < 0, 4, 6) = 6$
$S.PC = 8 \wedge c = 0 \wedge ite(a < 0 \vee b < 0, 4, 6) = 4$ S_8
S_8' $S.PC = 8 \wedge c = a + b \wedge ite(a < 0 \vee b < 0, 4, 6) = 6$
$S_8 \wedge \neg(c >= a \wedge c >= b)?$ A_8
A_8' $S_8' \wedge \neg(c >= a \wedge c >= b)?$

Fig. 13.8 State evolution in symbolic execution

the stack. The path condition records the constraints under which this branch is taken. Figure 13.8 shows that the path condition for symbolic state S_4 is $ite(a < 0 \vee b < 0, 4, 6) = 4$. This path condition can be syntactically simplified to $a < 0 \vee b < 0$ and records the fact that line 4 is only reached when $a < 0$ or $b < 0$. We now have two paths to explore, one with $PC = 4$ which is reachable when $a < 0 \vee b < 0$ and another with $PC = 6$ which is reachable under the path condition $\neg(a < 0 \vee b < 0)$. When we reach final instruction, in this case $PC = 9$, nothing is pushed onto the stack, signifying that no additional paths need to be explored.

This loop is repeated until the stack is emptied, i.e., all paths are visited.

Verifying Assertions

The symbolic state expressions constructed when verifying the code in Listing 13.1 are shown in Fig. 13.8. The white boxes show the symbolic state expressions and are labelled with their corresponding PC values. We see that $PC = 8$ is reached on both paths in the program with different path conditions.

The gray boxes show the assertions on line 8. Assertions are verified by negating the assertion condition, conjoining this with the state expressions and checking if the result is satisfiable in a constraint solver. In this example, A_8 is not satisfiable, so the assertion holds along this path. But the A_8' is satisfiable because the result of $c = a + b$ can overflow. This violation is reported by the algorithm.

The assertions shown in the listing are predicates on program states. Unfortunately, such predicates cannot be used to reason about information flow [34] and so the algorithm described above cannot be used to verify information flow properties. An algorithm that can verify information flow properties is given in Sect. 13.3.4.

13.3.4 Verifying Information Flow Properties

Algorithm 2 shows how information flow properties can be verified using symbolic execution. It performs a depth-first search (DFS) of all reachable instructions and checks whether the information flow property specified by $(src, dst, srcpred, dstpred)$ holds for all of them. The *stack* keeps track of paths to be visited and the *path constraints* P_A and P_B determine the conditions under which each path is taken.

Algorithm 2 Symbolic execution

```
Inputs: $init_S$, $\langle src, srcpred, dst, dstpred \rangle$
1: $stack.push(\langle init_S, init_S, \texttt{true}, \texttt{true} \rangle)$
2: while $\neg empty(stack)$ do
3:     ▷ $S_A$ and $S_B$ represent the current firmware state.
4:     $\langle S_A, S_B, P_A, P_B \rangle \leftarrow stack.pop()$
5:
6:     $T \leftarrow P_A \wedge P_B \wedge [\![dstpred]\!]_{S_A} \wedge [\![dstpred]\!]_{S_B} \wedge [\![dst]\!]_{S_A} \neq [\![dst]\!]_{S_B}$
7:     if $sat(T)$ then                                        ▷ check properties
8:         display violation
9:     end if
10:
11:    if $finished(S_A, S_B)$ then                            ▷ check for completion
12:        continue
13:    end if
14:
15:    ▷ $S'_A$ and $S'_B$ is state after this instruction is executed.
16:    $S'_A.mem \leftarrow N_{mem}(S_A)$
17:    $S'_B.mem \leftarrow N_{mem}(S_B)$
18:
19:    ▷ check for MMIO and handle it
20:    if $isMMIO(S'_A)$ then
21:        $\langle S'_A, S'_B \rangle \leftarrow execMMIO(S_A, S_B)$
22:    end if
23:    ▷ rewrite $datain$ if $memaddr$ matches the source
24:    if $overlaps(S'_A, src)$ then
25:        $S'_A.datain = \texttt{ite}(overlaps(S'_A.memaddr, src) \wedge srcpred(S_A), newVar(), S'_A.datain)$
26:        $S'_B.datain = \texttt{ite}(overlaps(S'_B.memaddr, src) \wedge srcpred(S_B), newVar(), S'_B.datain)$
27:    end if
28:    ▷ compute next value of the registers
29:    $\langle S'_A.regs, S'_B.regs \rangle \leftarrow \langle N_{regs}(S_A), N_{regs}(S_B) \rangle$
30:
31:    ▷ find all concrete values $\texttt{PC}_\texttt{i}$ consistent with the symbolic value $[\![PC]\!]_{S_A}$ and add to stack.
32:    for all $\texttt{PC}_\texttt{i} \models [\![\mathrm{PC}]\!]_{S_A}$ do
33:        $(S'_A.PC, S'_B.PC) = (\texttt{PC}_\texttt{i}, \texttt{PC}_\texttt{i})$
34:        $P_A \leftarrow P_A \wedge S_A.PC = \texttt{PC}_\texttt{i}$
35:        $P_B \leftarrow P_B \wedge S_B.PC = \texttt{PC}_\texttt{i}$
36:        $stack.push(\langle S'_A, S'_B, P_A, P_B \rangle)$
37:    end for
38: end while
```

```c
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];
}
```

Listing 13.2 Integrity property example: src=r1@, dst=*dataout*@, srcpred=`true` and dstpred=$memaddr = \texttt{0x200} \wedge memop = \mathsf{WR}$

There are two main enhancements over previous symbolic execution engines [4, 9, 10, 15]. The first is that the engine maintains *two* copies of the state in S_A and S_B. This is so that we can functionally test whether assigning different values to *src* results in different values at *dst*. This check is performed in line 7. The substitution of the source with "fresh" unconstrained variables is performed in lines 24–27. The other difference is the handling of MMIO instructions in lines 20–22 which are executed using simulation.

To understand Algorithm 2, let us consider its execution on the code shown in Listing 13.2. We show the algorithm in C-like pseudocode to make understanding easier but the analysis is done on binary code. The property here 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:

$P_A = \neg(x_A < 0 \vee x_A \geq 3)$	$P_B = \neg(x_B < 0 \vee x_B \geq 3)$
$S_A.dataout = \mathsf{M}_A[\texttt{0x100} + x_A]$	$S_B.dataout = \mathsf{M}_B[\texttt{0x100} + x_B]$
$S_A.memaddr = \texttt{0x200}$	$S_B.memaddr = \texttt{0x200}$
$S_A.memop = \mathsf{WR}$	$S_B.memop = \mathsf{WR}$
$S_A.\mathsf{M} = S_B.\mathsf{M} = [\texttt{0x100} \mapsto 1, \texttt{0x101} \mapsto 1, \texttt{0x102} \mapsto 3, \ldots]$	

In the above, P_A and P_B are the two path conditions which specify the conditions that must hold for this path in the program to the taken. $S_A.\mathsf{M}$ and $S_B.\mathsf{M}$ represent the values of the RAM. The state variables *dataout*, *memaddr*, and *memop* refer to the elements of *S.mem* which contain information about the memory operation being executed by this statement in each "copy" of the execution. x_A and x_B are the new variables created to represent the untrusted value `r1`. At this point, when the solver evaluates whether $dataout_A \neq dataout_B$ is possible along with P_A, P_B and the predicates, it will find $x_A = 1, x_B = 2$. This means that if $\texttt{r1} = x_A = 1$, then value stored in `IO_REG` is different from the value stored in `IO_REG` if $\texttt{r1} = x_B = 2$. This means the property being violated. Once we fix the bug and set `N`=2, then $(P_A, P_B) = (x_A \geq 0 \wedge x_A < 2, x_B \geq 0 \wedge x_B < 2)$. Now it is not possible make $dataout_A \neq dataout_B$ while satisfying P_A and P_B, so the algorithm will not report in an error.

An alternative technique for verifying information flow properties is *dynamic taint analysis* (DTA) [3, 13, 14, 29, 42, 46]. DTA works by associating a taint bit with each variable/object and propagating the taint bit from inputs to the outputs of a computation. DTA is a dynamic analysis: it is implemented by analysis of execution traces in which the source of the information flow property is tainted and the analysis checks if this taint propagates to the destination. DTA cannot search over the space of all possible executions.

Listing 13.2 is an example where DTA fails. Typically taints are tracked separately for memory addresses and memory data [42]. Therefore, if the address for a memory read is tainted but the data pointed to by the address is not tainted, the result of the memory read is *not* tainted. Under such a policy, the bug would not be detected by DTA as the taint would not propagate from `r1` to `IO_REG`. If we change the policy and taint the result of memory read when the address is tainted, we will have overtainting. DTA will report a problem even when the bug is fixed and `N`=2.

Now 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 it will be detected by Algorithm 2.

To detect this issue, DTA needs an instruction sequence where `r1`=2 in order to expose the violation. In other words, DTA cannot detect a violation without a trace that "activates" the problem. The above examples demonstrate the advantages of our technique: exhaustive static analysis of all program paths and states, and improved precision over DTA.

MMIO and Symbolic Execution

Memory-mapped I/O (MMIO) poses unique challenges for symbolic execution, because a read/write from/to MMIO might have "side-effects," e.g., initiating a hardware state machine. Ideally, we would model all these symbolically. However, this is difficult in practice because constructing a symbolic model for the entire MMIO space will be very time-consuming.

Therefore, we use selective symbolic execution to model the MMIO side-effects. Simulation models of the SoC are usually available during SoC design and these model MMIO accesses. We use these to execute the "side-effects" of MMIO reads/writes. This means we have to convert the symbolic expressions into sets of concrete values, execute the simulator for each concrete value, and then resume symbolic execution with the simulation results. This process is shown in Fig. 13.9b, which is contrast to fully symbolic execution shown in Fig. 13.9a. This process of conversion from symbolic to concrete states and back is tractable for firmware because typical accesses to only made to a small number of hardware registers.

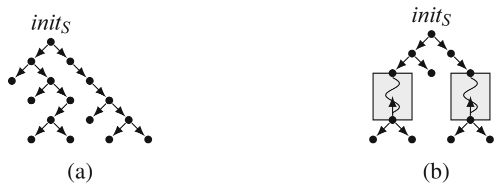


Fig. 13.9 Execution model. In (**b**) the *shaded boxes* show single-path (concrete) execution through the simulation model due to MMIO. (**a**) Fully symbolic execution. (**b**) Selective symbolic execution

13.3.5 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, and touch sensing. 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.

13.3.5.1 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 [16] 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.

13.3.5.2 Security Objectives

The PTIP firmware interacts with system software, device 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 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.

13.3.5.3 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 min. 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.

13.4 Discussion and Related Work

This chapter described a methodology for SoC security verification that is based on the construction of instruction-level abstractions and the use of symbolic simulation to verify information flow properties. We now discuss the applicability of potential extensions to this methodology. We also discuss related work.

13.4.1 SoC Security Verification

The ILA is a complete formal specification of hardware behavior that precisely defines the hardware/software interface. A key feature of the ILA is that it is machine-readable. Section 13.3 showed how symbolic execution on ILAs of microprocessors could be used to verify information flow properties in SoC designs. However, ILA-based verification is not limited to symbolic execution. Symbolic execution is just one way of carrying out the proof obligations posed by SoC security requirements. Other verification techniques, such as model checking and abstract interpretation, can also be used to carry out these proof obligations.

In particular, software model checkers [6, 11] can be more scalable than symbolic simulators because they avoid the path explosion problem by *implicitly* enumerating paths. However, current software model checkers do not support information flow properties. Therefore, extending model checkers with richer property specification schemes in order to verify information flow properties is an important area for further work. Using ILAs in other formal frameworks such as interactive theorem provers may also be beneficial.

13.4.2 Related Work

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

13.4.2.1 Synthesizing Abstractions

Our work [33, 49] builds on recent progress in syntax-guided synthesis which is surveyed in [2]. Our synthesis algorithm is based on oracle-guided synthesis from [26]. Our contribution is the use of synthesis for constructing abstractions of SoC hardware. Also related is the work of Godefroid et al. [22] 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. For example, they do not consider issues like the mapping between opcodes and instructions, the source registers, the memory addressing modes, how the PC is updated, etc. Our model considers all these details.

Also related is the work of Udupa et al. [52] They synthesize distributed protocols using a partial template specification and input/output traces. Similar to our work, they verify the correctness of the synthesized protocol using model checking. This idea of synthesis synergistically combining synthesis with model checking is also used by Gascon et al. [20] who synthesize cryptographic protocols which satisfy certain specifications.

13.4.2.2 SoC Verification

Refinement relations, used in proving the abstraction and the implementation match, are from [28, 35]. One approach to compositional SoC verification is by Xie et al. [54, 55] 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.

Horn et al. [24] suggest symbolic execution on a software model that contains both firmware and software models of hardware components. This approach is complementary to ours because it can be used for early design stage verification, when an RTL model may not be available. However, once the RTL model is constructed, there is no easy way of ensuring that the software model and the RTL are in agreement. This is the critical challenge addressed by our work.

13.4.2.3 Symbolic Execution and Taint Analysis

The DART and KLEE projects are the precursors of subsequent work in symbolic execution [9, 21]. 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 [10, 15]. 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) [3, 29, 42, 46]. DTA suffers from both false positives and false negatives due to the problems of under- and over-tainting. Our work [50] does not result in false positives. An overview of DTA and symbolic execution is presented in [42]. We show how symbolic execution can be used to verify information flow; this is missing from [42] which treats DTA and symbolic execution separately.

13.5 Conclusion

In this chapter, 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 to find several bugs in an SoC built out of open source components and part of a commercial SoC design.

References

1. S.V. Adve, K. Gharachorloo, Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
2. R. Alur, R. Bodik, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in *Formal Methods in Computer-Aided Design* (2013)
3. G.S. Babil, O. Mehani, R. Boreli, 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)

4. O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M.R. Tuttle, V. Zimmer, Symbolic Execution for BIOS Security, in *Proceedings of the 9th USENIX Conference on Offensive Technologies* (2015)
5. Berkeley Logic Synthesis and Verification Group, ABC: a system for sequential synthesis and verification (2014). http://www.eecs.berkeley.edu/~alanmi/abc/
6. D. Beyer, T.A. Henzinger, R. Jhala, R. Majumdar, The software model checker blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007)
7. M. Bohr, The new era of scaling in an SoC world, in *IEEE International Solid-State Circuits Conference-Digest of Technical Papers* (IEEE, New York, 2009), pp. 23–28
8. A.R. Bradley, SAT-based model checking without unrolling, in *Verification, Model Checking, and Abstract Interpretation* (2011)
9. C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in *Operating Systems Design and Implementation* (2008)
10. V. Chipounov, V. Kuznetsov, G. Candea, S2E: a platform for in-vivo multi-path analysis of software systems, in *Architectural Support for Programming Languages and Operating Systems* (2011)
11. E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in *Tools and Algorithms for the Construction and Analysis of Systems* (2004)
12. J. Cong, M.A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, G. Reinman, Accelerator-rich architectures: opportunities and progresses, in *Proceedings of the 51st Annual Design Automation Conference* (ACM, New York, 2014), pp. 1–6
13. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, P. Barham, Vigilante: end-to-end containment of internet worms, in *Symposium on Operating Systems Principles* (2005)
14. J.R. Crandall, F.T. Chong, Minos: control data attack prevention orthogonal to memory model, in *IEEE/ACM International Symposium on Microarchitecture* (2004)
15. D. Davidson, B. Moench, S. Jha, T. Ristenpart, FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution, in *USENIX Conference on Security* (2013)
16. L. De Moura, N. Bjørner. Z3: an efficient SMT solver, in *Tools and Algorithms for the Construction and Analysis of Systems* (2008)
17. R.H. Dennard, V. Rideout, E. Bassous, A. LeBlanc, Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J. Solid State Circuits **9**(5), 256–268 (1974)
18. H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in *Proceedings of the International Symposium on Computer Architecture* (IEEE, New York, 2011), pp. 365–376
19. Experimental artifacts and synthesis framework source code (2016). https://bitbucket.org/spramod/ila-synthesis
20. A. Gascón, A. Tiwari, A synthesized algorithm for interactive consistency, in *NASA Formal Methods* (2014)
21. P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in *Programming Language Design and Implementation* (2005)
22. P. Godefroid, A. Taly, Automated synthesis of symbolic instruction encodings from I/O samples, in *Programming Language Design and Implementation* (2012)
23. S. Heule, E. Schkufza, R. Sharma, A. Aiken, Stratified synthesis: automatically learning the x86-64 instruction set, in *Proceedings of Programming Language Design and Implementation* (2016)
24. A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, D. Kroening, Formal co-validation of low-level hardware/software interfaces, in *Formal Methods in Computer-Aided Design* (2013)
25. H. Hsing, http://opencores.org/project,tiny_aes (2014)
26. S. Jha, S. Gulwani, S.A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in *International Conference on Software Engineering* (2010)
27. S. Jha, S.A. Seshia, A theory of formal synthesis via inductive learning, in *CoRR*, abs/1505.03953 (2015)

28. R. Jhala, K.L. McMillan, Microarchitecture verification by compositional model checking, in *Computer-Aided Verification* (2001)
29. M.G. Kang, S. McCamant, P. Poosankam, D. Song, DTA++: dynamic taint analysis with targeted control-flow propagation, in *Network and Distributed System Security Symposium* (2011)
30. S. Krstic, J. Yang, D.W. Palmer, R.B. Osborne, E. Talmor, Security of SoC firmware load protocols, in *Hardware-Oriented Security and Trust*, pp. 70–75 (2014)
31. R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, D.M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction, in *Proceedings of International Symposium on Microarchitecture* (IEEE, New York, 2003), pp. 81–92
32. R. Lysecky, T. Givargis, G. Stitt, A. Gordon-Ross, K. Miller, http://www.cs.ucr.edu/~dalton/i8051/i8051sim/ (2001)
33. S. Malik, P. Subramanyan, Invited: specification and modeling for systems-on-chip security verification, in *Proceedings of the Design Automation Conference*, DAC '16, New York, NY (ACM, New York, 2016), pp. 66:1–66:6
34. J. McLean, A general theory of composition for trace sets closed under selective interleaving functions, in *IEEE Computer Society Symposium on Research in Security and Privacy* (IEEE, New York, 1994), pp. 79–93
35. K.L. McMillan, Parameterized verification of the FLASH cache coherence protocol by compositional model checking, in *Correct Hardware Design and Verification Methods* (Springer, Berlin, 2001)
36. A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, P. Nalla, GLA: gate-level abstraction revisited, in *Design, Automation and Test in Europe* (2013)
37. A.C. Myers, JFlow: practical mostly-static information flow control, in *Principles of Programming Languages* (1999)
38. M.D. Nguyen, M. Wedler, D. Stoffel, W. Kunz, Formal hardware/software co-verification by interval property checking with abstraction, in *Design Automation Conference* (2011)
39. A. Sabelfeld, A. Myers, Language-based information-flow security. IEEE Sel. Areas Commun. **21**, 5–19 (2003)
40. A. Sabelfeld, D. Sands, Declassification: dimensions and principles, J. Comput. Secur. **17**(5), 517–548 (2009)
41. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P.P. Pande, C. Grecu, A. Ivanov, System-on-chip: reuse and integration. Proc. IEEE **94**(6), 1050–1069 (2006)
42. E. Schwartz, T. Avgerinos, 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)
43. S.A. Seshia, Combining induction, deduction, and structure for verification and synthesis. Proc. IEEE **103**(11), 2036–2051 (2015)
44. R. Sinha, P. Roop, S. Basu, The AMBA SOC platform, in *Correct-by-Construction Approaches for SoC Design* (Springer, New York, 2014)
45. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat. Combinatorial sketching for finite programs, in *Architectural Support for Programming Languages and Operating Systems* (2006)
46. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, BitBlaze: a new approach to computer security via binary analysis, in *Information Systems Security* (2008)
47. J. Strömbergson, https://github.com/secworks/sha1 (2014)
48. P. Subramanyan, D. Arora, Formal verification of taint-propagation security properties in a commercial SoC design, in *Design, Automation and Test in Europe* (2014)
49. P. Subramanyan, Y. Vizel, S. Ray, S. Malik, Template-based synthesis of instruction-level abstractions for SoC verification, in *Formal Methods in Computer-Aided Design* (2015)
50. P. Subramanyan, S. Malik, H. Khattri, A. Maiti, J. Fung, Verifying information flow properties of firmware using symbolic execution, In *Design Automation and Test in Europe* (2016)
51. S. Teran, J. Simsic, http://opencores.org/project,8051 (2013)

52. A. Udupa, A. Raghavan, J.V. Deshmukh, S. Mador-Haim, M.M. Martin, R. Alur, TRANSIT: specifying protocols with concolic snippets, in *Programming Language Design and Implementation* (2013)
53. C. Wolf, http://www.clifford.at/yosys/ (2015)
54. F. Xie, X. Song, H. Chung, N. Ranajoy, Translation-based co-verification, in *Formal Methods and Models for Co-Design* (2005)
55. F. Xie, G. Yang, X. Song, Component-based hardware/software co-verification for building trustworthy embedded systems. J. Syst. Softw. **80**(5), 643–654 (2007)