Security of SoC Firmware Load Protocols

Sava Krstić, Jin Yang
Strategic CAD Labs
Intel Corporation

David W. Palmer, Randy B. Osborne
Product Development Group
Intel Corporation

Eran Talmor
Core CAD Technologies
Intel Corporation

***Abstract*—The security architecture of modern systems-on-a-chip (SoC) is complex and critical to be done right and quickly. SoC security architects feel an acute need for new tool-supported specification and validation technologies. Aiming to stimulate research into creation of these technologies, in this paper we provide some industrial insights and initial solutions. Focusing on a concrete non-trivial example of security sensitive *firmware load protocols*, we show how to: (1) concisely specify the communication between IP blocks; (2) model the adversary; (3) debug and verify the protocol.**

I. Introduction

Security of computation on phones, tablets and other platforms depends critically on the underlying System-On-Chip (SoC) fulfilling its security objectives. Since SoC is a network of communicating hardware components (IP blocks), any SoC security failure is a result of (1) some of the IP failing to meet its security objectives, or (2) error in some of the protocols governing inter-IP communication. A security architect designing a system-level protocol needs to provide a description of the protocol that communicates to the designers of all the participating blocks a sufficient and unambiguous set of requirements. At the same time, the system security architect needs to make sure that the protocol itself does not have security defects. Now, any systematic search for bugs in a security protocol demands clear enough notions of the security objectives of the protocol, and of the adversary. Thus, raising the current level of reasoning/debugging from pen-and-paper to tool-supported requires creation of abstract SoC system models that include the adversarial actions, and a formulation of security objectives as checkable system properties.

Current SoCs contain dozens of IP blocks that communicate over well-defined interfaces and interconnects. Some blocks are programmable devices (for example, camera and audio controllers) with their firmware loaded at the boot time from external storage, say eMMC card. Additionally, there is often a requirement that the host OS be able to stop the device, load an updated version of its firmware, and then restart the device. A fundamental SoC **security objective** is that, *at any time, the device may only run authenticated firmware.* Since authentication involves a set of cryptographic operations, it is typically not done by the device itself but by a specialized controller, which we will call CE (for crypto-engine). The major threat in this setup is that some agent manages to get inauthentic firmware to the place from where the device runs. We can trust CE and the device hardware with its bootloader ROM, but the driver (and software running on other microcontrollers) may be malicious.

To the security architect, the above poses a rather concrete **firmware load problem**: *Design a protocol specifying the roles of each participating agent (Device, its host OS Driver, and CE) so that the security objective is satisfied even in the presence of a malicious Driver.*[1]

As is commonly the case with security protocols, the most obvious solutions are flawed, and it is difficult to convincingly argue that a flawless-looking design is correct. Our goal in this paper is to demonstrate pitfalls in the design of firmware load protocols and show how a formal analysis can be done that will either prove that a protocol is correct, or produce an attack scenario.

The paper is organized as follows. In Section II we begin with the simplest protocol that may appear to be a solution to our firmware load problem. It has a bug and three ways to fix it suggest themselves, but all three turn out to be flawed too. Two possibilities to combine these fixes give us new candidate solutions at the next level of complexity and one of them finally works. In Section III, we describe how these protocols can be conveniently formalized and analyzed with model-checking tools. Modeling the adversary is a crucial part of our analysis. Section IV discusses related and future work, and then we conclude in Section V.

II. Flows, Flaws and Fixes

In any solution to the firmware load problem, when the Device starts running its firmware, that firmware must be in a memory that is not accessible to the host OS and thus rewriteable by the malicious Driver. Abstracting the situation, let us say there is a *system memory* (SM) that is accessible by all agents, and an *isolated memory* (IM), accessible by Device and not by Driver. By giving CE read access to IM, we may come up with the simple protocol F_b ("*b*" for *basic*) shown in Figure 1. Intel architects use the term *flows* for pictorial protocol representations as in Figure 1. We adopt the term and will give it a precise meaning in Section III.

In F_b, when Driver decides to start the protocol, it resets Device and copies the firmware it wants loaded to a place in SM. Then it sends the message $\mathsf{Load\_fw}$ to Device (which, after reset, is running from bootloader ROM), with the information about the location of the firmware. Device then copies firmware from that location to IM and sends the message $\mathsf{Auth\_req}$ to CE, providing the location of the copied firmware, and asking for authentication of it. CE serves the request and responds to Device with the $\mathsf{PASS}/\mathsf{FAIL}$ status (sts) of the authentication. Device forwards sts to Driver and if $sts = \mathsf{PASS}$, then it stops executing from ROM and jumps

[1] The problem is a precisely formulated instance of the more general *Device Integrity Problem*. In [3], device integrity is defined as "the absence of corruption in the hardware, firmware and software of a device".

978-1-4799-4112-4/14/$31.00 ©2014 IEEE

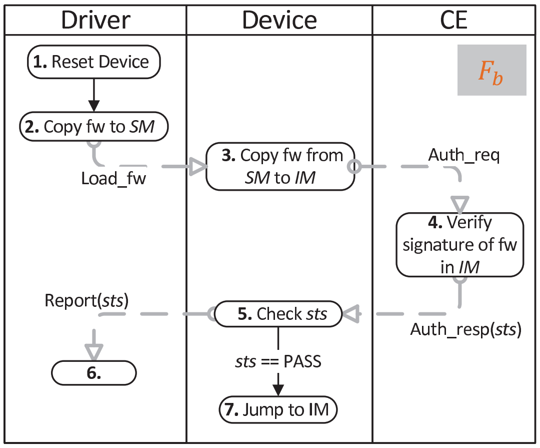

Fig. 1. Basic firmware load protocol.

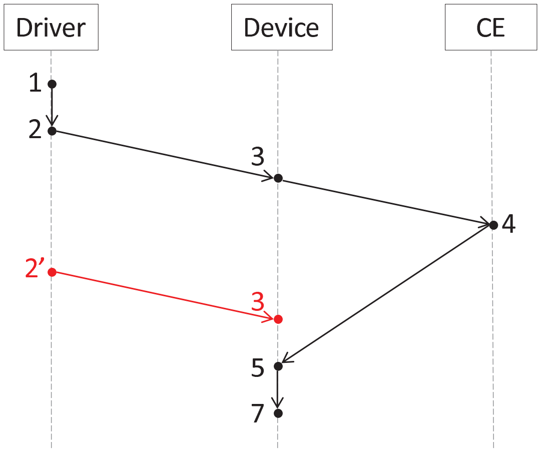

Fig. 2. A scenario showing that F_b does not meet its security objective.

to the firmware from IM. If sts = FAIL, Device does nothing (halts), but Driver may try a remedial action, which is irrelevant for us.

Our concern is the security objective of the protocol: at the moment the execution of the protocol gets to the "Jump to IM" stage (and until the next reset of Device), the firmware in IM must be authentic. In fact, it may not be, as shown in Figure 2. What is happening is that Driver first provides authentic firmware (task 2) and then overwrites it with an inauthentic one (the second instance of task 2). At the time Device gets Auth_resp(PASS), bad firmware has been prepared for it in IM.

The sequence

$$(1, 2, 3, 4, \underline{2}', \underline{3}, 5, 7)$$

encodes the counterexample in Figure 2: the numbers refer to the numbered "tasks" in Figure 1, underlining indicates the second occurrence of the same task in the sequence, and the prime symbol in $2'$ indicates copying of bad firmware. For the economy of space, in the rest of the section we will use this notation to present counterexample scenarios.

Protocol F_a in Figure 3 is an attempt to fix F_b; it introduces an internal boolean variable $active$ (initially and upon reset set to false) to Device in order to make it know when it is engaged in the protocol, so that it does not do the second copy to IM. Even though F_a positively strengthens F_b, it can still fail. A sequence encoding a counterexample trace is given in the caption of Figure 3.

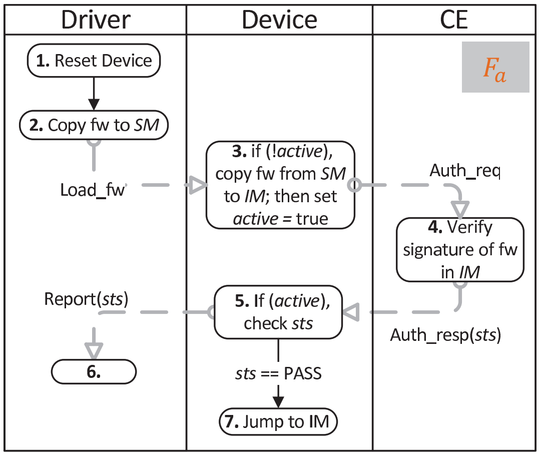
Driver
Device
CE
F_a
1. Reset Device
2. Copy fw to SM
Load_fw
3. if (!active), copy fw from SM to IM; then set active = true
Auth_req
4. Verify signature of fw in IM
Report(sts)
5. If (active), check sts
Auth_resp(sts)
6.
sts == PASS
7. Jump to IM

Fig. 3. Attempt to correct F_b by making Device aware of the protocol execution stage. It fails with counterexample trace $(1, 2, 3, \underline{1}, 4, \underline{2}', \underline{3}, 5, 7)$.

One might ask why not keep the $active$ state in CE instead of Driver, and let CE do the copying into IM? That would work, but let us say it is required to not add hardware to CE, and to minimize its knowledge about the rest of the system. Thus, we have to look for other solutions.

Protocols F_{lu} and F_{ul} in Figs. 4,5 (subscripts indicate "lock-unlock" and "unlock-lock") attempt to fix F_b by using the ability of CE to lock Device's access to its isolated memory. In F_{lu}, CE prevents writing to IM during authentication. In F_{ul}, write access to IM is restricted at all times except when Device needs to copy firmware from SM into it, and then it explicitly asks CE for premission. Both of these strengthenings of the basic protocol fail, as witnessed by traces given in the figure captions.

Our next attempts to arrive at a correct protocol combine the features of F_a with F_{lu} and F_{ul}. The resulting protocols F_{alu} and F_{aul} are in Figs. 6,8. Neither of them is perfectly safe, but breaking them requires more effort. In the attack on F_{alu} given in the caption of Figure 6, Driver generates three instances of the protocol. The first two instances load good firmware in SM, the third loads a bad one. As soon as the first instance passes task 3, Driver resets Device and starts the second instance. CE serves both authentication requests, but after the first authentication response is processed by Device (task 5), Device gets reset again and the third instance starts. After the second authentication, CE receives the Ack message from the first instance, and unlocks SM—just in time to enable Device to copy the bad firmware to IM (servicing the third Load_fw request). And then Device receives the second authentication response (a PASS) and moves on to run the (bad) firmware. The whole scenario is visualized in the message sequence chart in Figure 7.

Protocol F_{aul} is basically correct. The attack scenario in the

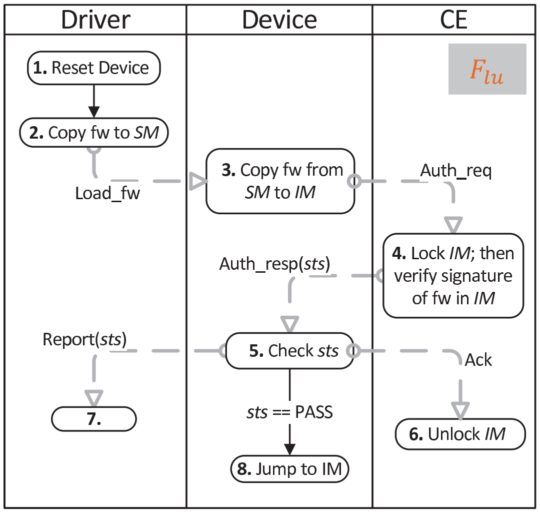

Fig. 4. CE locks IM before it performs authentication. Counterexample: $(1, 2, 3, 4, 5, 6, \underline{2}', \underline{3}, 8)$.

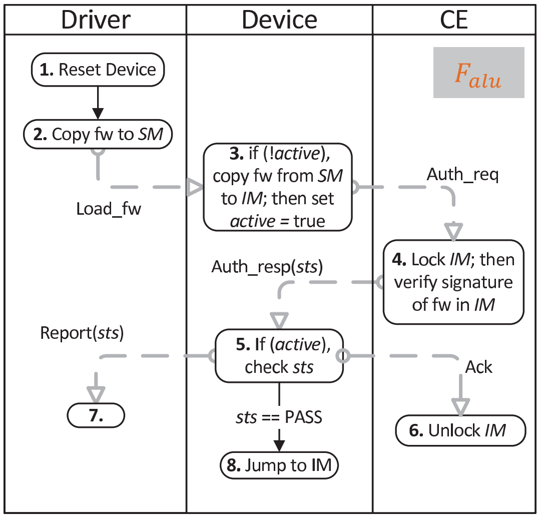

Fig. 6. A combination of F_a and F_{lu}. Figure 7 depicts the counterexample $(1, 2, 3, \underline{1}, \underline{2}, \underline{3}, 4, 5, \bar{1}, \underline{4}, 6, \bar{2}', \bar{3}, \underline{5}, \underline{8})$.

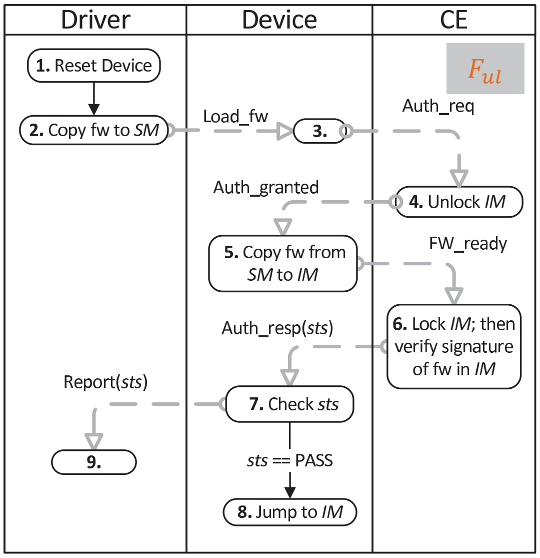

Fig. 5. CE keeps IM locked except for an interval when task 5 is performed. Counterexample: $(1, 2, 3, 4, 5, 6, 7, \underline{2}', \underline{3}, \underline{4}, \underline{5}, 8)$.

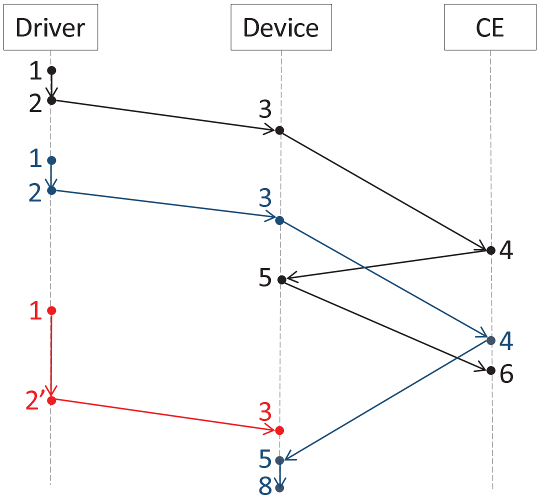

Fig. 7. Message sequence chart showing a security flaw of protocol F_{alu}.

caption of Figure 8 shows Device sending messages Auth_req and FW_ready in tasks 5 and $\underline{3}$ that CE executes out-of-order in tasks $\underline{4}$ and 6. There is a similar attack scenario in which Device executes two messages from CE out-of-order. However, if both Device and CE execute in-order (which should be a realistic assumption), F_{aul} does not fail.

Finally, we should note that industrial firmware loading protocols are an order of magnitude more complex than the examples we have discussed. Our toy examples, however, are skeletons of protocols that have been considered by Intel architects, and the vulnerabilities they expose are realistic. Let us also point out that the problem set-up we used is the simplest, but not the only one relevant. For example, the architect may start with the assumption that Device will be executing firmware from its *local memory* (SRAM) and that the isolated memory is used only for authentication. Figure 9 shows a protocol for this situation that fails because of a subtle attack that (as in the case of F_{alu}) involves Driver creating three protocol instances.

III. Modeling and Validation

In this section, we show how a protocol presented by a flow-diagram as in Section II can be compiled into a transition system, how the transistion system can be modified to include a model of adversarial actions, and finally how the resulting system is model-checked to see if it has a security flaw.

A. *The structure of a flow*

Suppose F is a flow-diagram. It is a directed graph whose vertices we call *tasks* and associate each of them with an agent of the protocol. The graph has two kinds of edges: *message-edges* connect tasks from distinct agents, and *control-edges*

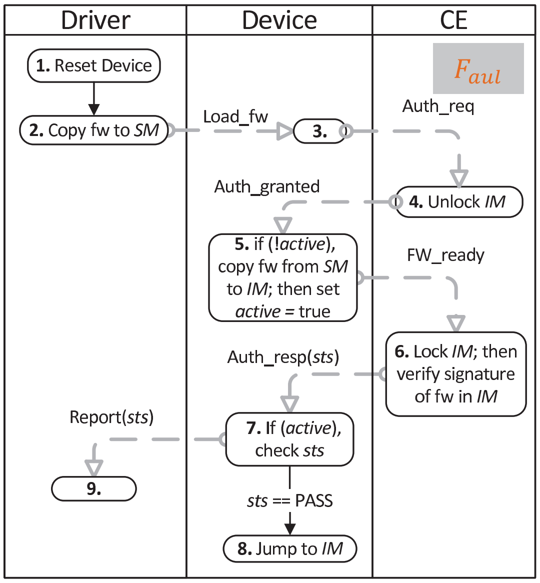

Fig. 8. A combination of F_a and F_{ul} with the counterexample $(1, 2, 3, 4, 5, \underline{2}', \underline{3}, \bar{1}, \underline{4}, 6, \underline{5}, 7, 8)$ that requires CE to execute out-of-order.

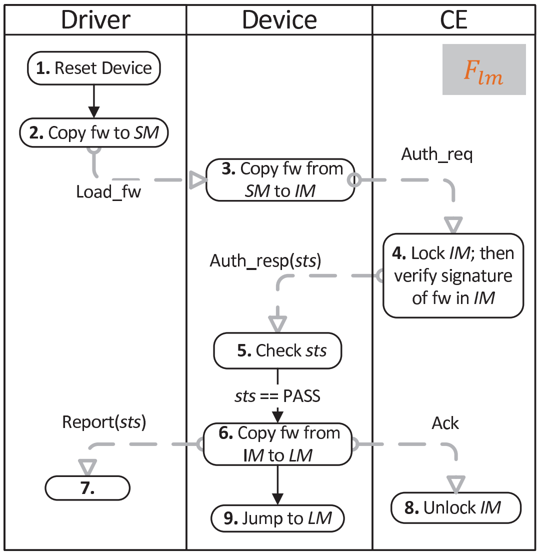

Fig. 9. A protocol for Device that executes from local memory LM. Attack: $(1, 2, 3, \underline{2}, \underline{3}, 4, 5, 6, \underline{4}, 8, \bar{2}', \bar{3}, \underline{6}, 9)$.

connect tasks of the same agent. The English text in tasks is pseudocode; let us assume that it can be readily expressed as code that takes the form of a conditional assignment: if a condition (boolean expression) is true then a sequence of assignments is to be executed. We will refer to variables occuring in task conditions and assignments as *flow variables*. Let us assume that *initial values* of all flow variables are given as part of the definition of F, that every flow variable may *belong* to a unique agent, and that the reset task for any specific agent assigns variables that belong to it to their initial values.

We assume that dynamic access control can be expressed by means of flow variables. Abstracting the exact mechanism, let us just postulate that for every agent A and every flow variable x there exist predicates "A *can read* x" and "A *can write* x" written in terms of flow variables.

Tasks of a flow can be partitioned into *control flow graphs (cfg)*. By definition, two tasks are in the same cfg iff they can be connected by a path of control edges only. Clearly, each cfg belongs to a unique agent. For simplicity, let us assume that (1) every cfg has at most one task with an incoming message (cfg's *start task*); (2) every cfg executes deterministically; and (3) there is exactly one *flow start* task that has no incoming edges.

For example, in the flow F_{aul}, we see nine tasks distributed over the three agents, two control edges, and six messages. The flow variables are SM, IM, $active$, and $lock_{IM}$. Tasks 1 and 2 form one cfg; tasks 7 and 8 form another; all other tasks are one-vertex cfgs.

B. The system described by a flow

Given a flow F, let us now describe how to generate from it a transition system S. We will define S in a standard fashion by (1) a set of *state variables*; (2) *initial conditions*; and (3) *transition rules* in the guard-action format, where the guard is a boolean expression over state variables and the action is an assignment to state variables. The system S executes non-deterministically starting from initial states. A transition $s \to s'$ is possible iff there exists a transition rule whose guard is true in s and whose action when applied to s produces s'.

By definition, the state variables of S are all the flow variables, together with

- a sequence Q of messages, each paired with its *status*, which can take three values: $\mathbf{f}$, $\mathbf{t}$, $\mathbf{e}$
- a subset L of the set of all control edges of F

The sequence ("queue") Q represents messages currently "in-flight" and the status values $\mathbf{f}, \mathbf{t}, \mathbf{e}$ stand for "in fabric", "at target" (received by target agent), and "enabled" (executable by target agent). The set L represents the current set of program locations or control points in the usual sense.

By definition, initial conditions are: L is the empty set; Q is the empty sequence; every flow variable has its initial value.

Stipulating that agents execute tasks atomically, we generate a transition rule τ of S for every task t of F. Recall that t is of the form "`if` c `then` a", where c is boolean expression and a is a sequence of assignments. The *access condition* of t is by definition the conjunction of predicates "A can read x" and "A can write y", where A is the agent that contains t; the conjunction is taken over all variables x that need to be read in order to do the assignemts a, and for all variables y that need to be written in a.

We define the guard of τ to be the condition c conjuncted with the access condition of t and further conjuncted with a disjunction of *trigger conditions*

- $e \in L$
- Q contains message m with status $\mathbf{e}$

for every control edge coming into t and for every message edge m coming into t.[2]

The guard of τ may be additionally strengthened with a conjunct that reflects the constraints of the execution policies implemented in each agent. A typical "single-threadedness" assumption would add to the guard of any message-triggered transition τ the condition that L contains no control edge residing in the agent that contains the task t. (This is saying that the agent cannot process more than one message at a time.)

The action part of τ consists of the assignment sequence a followed by a three-step update of Q and L described next.

Step 1. Pick a trigger condition that is true (typically there will be only one); if it is of the form $e \in L$, then remove e from L; if it is of the form "Q contains m...", then remove m from Q.

Step 2. This applies only if in Step 1 a message m got removed from Q. The step consists of changing the status of some messages in Q from $\mathbf{t}$ to $\mathbf{e}$. Precise details depend on the assumed execution model of the agents participating in S. In a typical ("message queue") case, Q will at any time have at most one message with status $\mathbf{e}$ that targets any particular agent A. When such a message is removed from Q (in Step 1), the oldest message in Q that targets A and has status $\mathbf{t}$ gets new status $\mathbf{e}$.

Step 3. Add to L all control edges coming out of t (if any), and enqueue to Q all the message edges coming out of t (if any). Give the newly added messages status $\mathbf{f}$.

This completes the definition of the transition rule τ of S associated with a task t of F.

The system S has only one additional kind of transitions, which consists of simply changing the status of a message in Q from $\mathbf{f}$ to $\mathbf{t}$. Which of the messages with status $\mathbf{f}$ in a current state can progress to the $\mathbf{t}$ status depends on ordering rules specified for the fabric (interconnect); in fact, this set of transitions provides a *model of the fabric*. In the simplest case, the $\mathbf{f}$-to-$\mathbf{t}$ status change can happen to a message m iff there is no other message with status $\mathbf{f}$ in Q that is older than m and has the same source and target agents as m. This is the model where the fabric is seen as a set of FIFO point-to-point channels.

Finally, the transitions involving reset of an agent include an update of Q that does not fall into the general scheme described above: when an agent A is reset, Q must be rid of all messages whose target is A and whose status is $\mathbf{t}$ or $\mathbf{e}$.

Note that since Q and L are initially empty, the only transition that can happen first is the one associated with the start task. Note also that, unless this transition leads to a state in which the guard of the start task is disabled, the second transition that takes place can be the same start transition. In other words, the system may be allowed to execute multiple instances of the same flow concurrently. This feature is crucial for realistic system modeling; moreover, we often need to model concurrent execution of (instances of) several flows.

C. *The adversary model*

The adversary in the context of SoC communication protocols is one or several untrusted hardware or software components that participate as agents in the protocols. Their adversarial ability in our model consists of *sending arbitrary messages to the fabric* over the interfaces available to them. In particular, the adversary can attempt to read and write any register. Which messages sent by the adversary will be received and acted upon will depend on the access control policies implemented in the fabrics and the receiving agents. We make a non-trivial assumption that *no agent (adversary included) can impersonate another agent*. (For instance, in our Section II flows, Driver cannot send the message Auth_resp(PASS) to Device and force it to execute bad firmware so easily.) Existing hardware mechanisms make this asssumption realistic.

The access control mechanism in our formal model (Section III-A) captures the adversary's reading and writing limitations. As regards sending arbitrary messages, we restrict the power of the adversary agent to *send at any time any message it may normally send in some legitimate protocol*. To extend the flow model defined in Section III-B to include this behavior of the adversary, we simply *relax the guards of all tasks that belong to the adversary by eliminating trigger conditions from them*.

D. *Validation*

Firmware load protocols lend themselves naturally to modeling and analysis with common explicit-state model checkers such as Murphi and Spin [6], [8]. We have created Murphi models of the flows in Section II and all counterexample traces were obtained by the tool.[3] Our Murphi code implements transition system models including adversarial actions along the lines of Sections III-B, III-C, using standard abstractions, such as letting state variables for memory locations that contain firmware take only two values (authentic and inauthentic). The correctness property we checked was an invariant saying that *when the location set L contains the edge into the "Jump to IM" task, then the value in IM is* authentic. All models were small and the tool took negligible time to explore the whole state space.

Modeling of Intel firmware load flows was done in Promela and checked with Spin. Due to the complexity of these flows, modeling took days of work and model-checking runs often took hours. It was beyond the tool's capacity to explore the entire protocol's state space, but we have obtained a number of consequential counterexample scenarios (of lengths up to 150). Some of them were non-bugs showing the unanticipated necessity to add protocol details that we initially abstracted away. Other examples showed concurrency issues that might or might not happen depending on atomicity of hardware operations involved.

IV. Related Work

In order to satisfy their security objectives, modern SoCs implement a multitude of mechanisms and features. Precise formulation of SoC security objectives, their derivation from security objectives of the platforms incorporating the SoC,

[2] Putting access control in the task guard is a modeling decision chosen for simplicity.

[3] The code is available from the authors.

and validating that the security objectives are actually met by the SoC product—all these are important problems that have been lacking a systematic treatment. As a composite of communicating machines (IP blocks), at the high level, the SoC is "programmed" by a large set of protocols, and it concurrently executes multiple instances of these protocols. Identifying essential aspects of SoC communication and adequately formalizing the view of SoC being a set of protocols are foundational questions that do not seem to have received sufficient attention.

Firmware integrity has been a recognized security issue in all contexts involving devices with updateable firmware (for example, see [13], [4]), and so has its vulnerability to time-of-check/time-of-use (TOCTOU) attacks [1]. Our analysis of firmware load flows can be seen as a search for a TOCTOU attack at the level of SoC communication, while in [1] the authors "note that we do not consider TOCTOU on hardware elements or carried out via hardware elements".

Security analysis demands a realistic adversary model. Most of the research in security protocols assumes an Internet-like communication environment and the presence of a strong (Dolev-Yao) adversary who can eavesdrop, impersonate, modify messages, etc.—limited only by unbreakability of crypographic primitives [12]. For SoC communication, this is too strong. The capabilities of the adversary described in Section III-C have been derived as a close approximation of what a malicious hardware or software agent within a concrete Intel SoC actually can and cannot do. It is conceivable that other products will entail significantly different adversary models, though one should not expect a great variety of them. Doubtless, a full SoC adversary model is yet to be worked out; it will need to take into account a curious fact: the SoC contains a family of mutually distrusting processors.

Graphical representation of protocols and scenarios has been widely adopted by SoC architects, but mostly used at an intuitive level. Visual languages like *Message Sequence Charts* [7] and *Live Sequence Charts* [5] with existing semantics and tool support have not found their way into the hardware industry. We refer to [2], [9] for pioneering attemtps to build a specification and validation methodology for SoC communication around Live Sequence Charts. The flow diagrams in this paper are created using BPMN stencil (in Microsoft Visio). In an experimental "flow language" for SoC protocols, we use a restricted set of BPMN shapes (not all of which show in our pictures) and do not strictly follow the BPMN semantics [11], [10], targeting instead a domain-specific semantics as sketched in Section III.

V. Conclusion

Focusing on a single security objective and (versions of) one protocol, we have demonstrated the difficulty of SoC security validation and its dependence on reliable analysis of intricacies of concurrent execution that takes place in an SoC. We laid out a formal SoC model as a system of concurrently executing protocols (flows) and built into it a simple concept of an SoC adversary and the security objective stated as a checkable property of the model.

We excercised this methodology on an Intel SoC product, implementing a model of its most complex firmware upload flow and subjecting it to simulation and model checking with the tool Spin. Even though the analysis did not discover serious bugs, it raised confidence in the protocol's correctness and provided valuable insights into assumptions underpinning the correctness.

Our work has spawned an ongoing research and development effort at Intel aiming at a comprehensive SoC security specification and validation technology that will help eliminate subtle architectural bugs at an early stage of SoC design and thus substantially decrease the validation time. We intend to make the methodology architect-friendly so that the formal model and simulation/verification runs of it can be automatically generated from parseable protocol specifications in flow form and will not require user expertise in formal methods. Identifying crucial SoC security objectives (beyond firmware upload correctness) and formulating them as model properties understood by analysis tools is yet to be done, and so is strengthening of tools to efficiently handle industrial-size input.

Acknowledgment

Yannis Schoinas and Manoj Sastry called for creating crisp specification and validation methodologies for SoC security architecture; we thank them for inspiration and support. We also thank Amit Goel, John O'Leary, Zurab Khasidashvili, Sasha Novakovsky, Eli Singerman, and Mandar Joshi for collaborations and discussions that have influenced the research reported in this paper.

References

[1] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. TOCTOU, traps, and trusted computing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*.

[2] A. Bunker, G. Gopalakrishnan, and K. Slind. Live sequence charts applied to hardware requirements specification and verification. *STTT*, 7(4):341–350, 2005.

[3] L. Chen, J. Franklin, and A. Regenscheid. Guidelines on hardware-rooted security in mobile devices (Draft). Technical Report 800-164, National Institute of Standards and Technology, 2012.

[4] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*. The Internet Society, 2013.

[5] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[6] D. L. Dill. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[7] D. Harel and P. S. Thiagarajan. Message sequence charts. In *In UML for Real: Design of Embedded Real-Time Systems*, pages 77–105. Kluwer Academic Publishers, 2003.

[8] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. 2003.

[9] R. Kumar and E. G. Mercer. Verifying communication protocols using live sequence chart specifications. *Electr. Notes Theor. Comput. Sci.*, 250(2):33–48, 2009.

[10] V. S. Lam. A precise execution semantics for BPMN. *IAENG International Journal of Computer Science*, 39(1):20–33, 2012.

[11] Object Management Group. Business Process Model and Notation (BPMN) version 2.0. Technical report, 2011.

[12] P. Ryan and S. Schneider. *The Modelling and Analysis of Security Protocols: The CSP Approach*. 2000.

[13] Various. UEFI today: Boostrapping the continuum. *Intel Technology Journal*, 15(1):8–21, 2011.