Formal Verification of Taint-propagation Security Properties in a Commercial SoC Design

Pramod Subramanyan
Princeton University
psubrama@princeton.edu

Divya Arora
Intel Corporation
divya.arora@intel.com

***Abstract*—SoCs embedded in mobile phones, tablets and other smart devices come equipped with numerous features that impose specific security requirements on their hardware and firmware. Many security requirements can be formulated as taint-propagation properties that verify information flow between a set of signals in the design. In this work, we take a tablet SoC design, formulate its critical security requirements as taint-propagation properties, and prove them using a formal verification flow. We describe the properties targeted, techniques to help the verifier scale, and security bugs uncovered in the process.**

I. Introduction

We live in a world where a cellphone can unlock itself by "recognizing" an authorized face [1], serve as a virtual wallet [2] when we go shopping, and play a high-definition movie [3] while we wait in the checkout line. And all this while letting our friends know our whereabouts! All these features have tremendous implications on end-user security and privacy, and are achieved through protections distributed across hardware (HW), firmware (FW) and software (SW). To further complicate things, the end-user is not always the "victim" in a security exploit. For example, SoCs may contain high-value assets like the ability to unwrap DRM (Digital Rights Management) keys, where the end-user is the most likely adversary. Another example is the SIM lock capability that binds the end-user to a carrier and helps the phone manufacturer recover the costs of a subsidized device. Add to these, the competing security expectations by the device manufacturer and the chip manufacturer and we get a long list of complex protections that an SoC needs to support.

Security of a product is only as good as its weakest link. Hence, a comprehensive validation of its security requirements is paramount. In this paper, we present a methodology for formal verification of hardware security requirements in a commercial tablet SoC design. The key insight is that many security requirements can be formulated as taint-propagation properties and this is the most natural way of expressing properties related to information flow and access control. Structural path analysis to identify security relevant logic for further manual review has been tried before [4]. In this work, we perform automated formal analysis of taint-propagation properties using a formal verification tool. We translate several high-level security requirements into taint-propagation properties and prove them using the tool. We discuss techniques to overcome the capacity limitations of the verifier and briefly mention the uncovered security vulnerabilities. To limit exposure, details of the SoC design and the vulnerabilities are abstracted out.

II. Background

Fig. 1(a) shows a high-level block diagram of our SoC, which is targeted at tablets for 2015. It comprises several host CPU cores that run the operating system and user applications. A Network-on-Chip (NoC) fabric connects the host CPU to memory and other IP blocks like display, camera, etc. In this work, we focused on the Security Controller IP (SCIP) which is magnified in Fig. 1(b). The SCIP comprises an off-the-shelf microcontroller (μC) which executes authenticated FW (SCIP FW), a memory management unit (MMU) to enable privilege separation on the μC, on-chip ROM and RAM, and several crypto-accelerators. This SCIP HW/FW provides a number of security features like secure boot, DRM, etc.

A **taint-propagation property** has the following elements:

- *src*: RTL signals "seeded' with the taint.
- *dest*: signals to which the taint must *not* propagate for the property to be satisfied.
- *conditions*: temporal logic expressions that must be true at various points in the taint propagation, *e.g.* when the taint starts or when it ends.

Confidentiality requirements can be verified by setting a hardware secret as the *src* and the data bus of an external interface as the *dest*. Similarly, integrity can be verified by setting an untrusted interface as the *src* and a sensitive signal as the *dest*. The formal verification tool we used analyzes taint-propagation properties and either proves each property or finds a counterexample showing a *functional path* from *src* to *dst*.

III. SCIP Security Properties

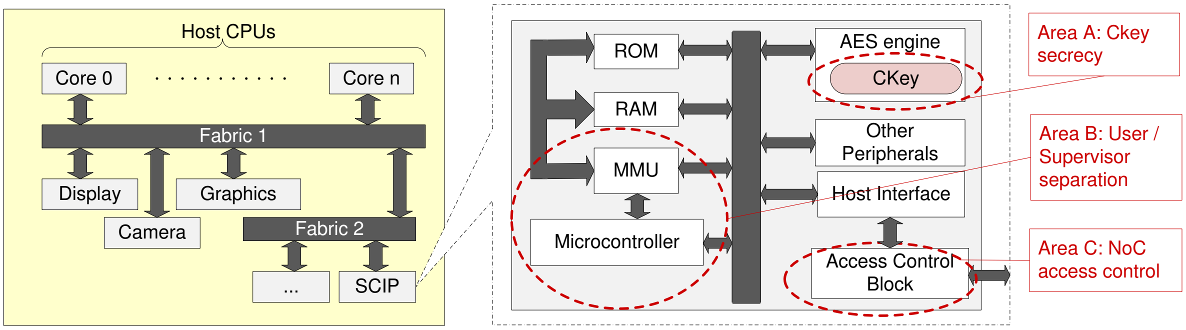
We analyzed the architecture and design of the SCIP and selected three broad areas to verify. After experimentation, we converged on a set of properties which represent the core security requirements for each area. The areas are detailed below and also depicted in Fig.1(b).

A. CKey Secrecy:

Security Requirement: CKey (Chip Key) is a hardware key, used in DRM application flows. SCIP FW can use CKey for encryption or decryption by configuring a crypto-engine to get the key directly from HW. However, to reduce the attack surface, even SCIP FW is not allowed to read the CKey.

Verification Strategy: Phrasing the question "can SCIP FW read the CKey?" as a taint-propagation property was straightforward, but beyond the tool's capacity. Fortunately, taint-propagation properties can be decomposed along the structural path between *src* and *dest*. Instead of asking "Can A reach C?", we can ask "Can A reach B and can B reach C?", where B is some signal on the structural path. We also had to guide the tool to ignore certain paths. For instance, a path that goes through the encryption logic and uses the CKey is a legitimate taint, but uninteresting from the perspective of CKey secrecy.

978-3-9815370-2-4/DATE14/©2014 EDAA



(a) SoC Block Diagram

(b) SCIP Block Diagram

Fig. 1. SoC architecture and block diagram of the Security Controller IP that we analyzed.

B. User/Supervisor (U/SU) Mode Separation:

Security Requirement: SCIP μC can run FW in User or Supervisor mode. The MMU performs logical to physical address translation and also enforces read/write/execute access controls on the physical address range. Once the MMU is configured, User FW should not be able to access privileged memory or I/O directly. Note this security requirement is applicable to almost all modern processors and microcontrollers.

Verification Strategy: Formulation of this property required more creativity and domain knowledge. We identified HW locations which could enable User FW to break mode separation and created distinct properties with each of them as *dest*. These properties also needed a *src-precondition* to ensure the taint was started by a User instruction. To specify the precondition, we had to add some book-keeping logic to the μC pipeline. We also had to initialize the formal model to a state where the MMU was properly configured. Finally, we had to abstract some μC blocks and reduce the number of MMU entries in the design to make the proofs scale.

C. NoC Access Control:

Security Requirement: SCIP is a sensitive IP and transactions from other untrusted IPs, including those initiated by Host SW should not be able to write to SCIP internal registers. NoC access control is a common security property which applies to many sensitive IPs in the SoC and a methodology to verify these properties in one IP can be ported easily to others and added to the SoC regression suite.

Verification Strategy: Property formulation was straightforward but we had to introduce some assumptions to get full proofs. The underlying insight here was that a counter-example could be of 2 types: (a) incoming transaction can write to an SCIP register or (b) incoming transaction can "influence" the value written to SCIP register by the μC. We excluded (b) by disconnecting the μC interface and obtained full proofs.

IV. Results

Since SCIP had been previously validated through simulation-based tests and manual reviews, we were expecting proofs of correctness rather than bugs. However, the formal methodology uncovered two RTL logic bugs and both were also demonstrated through exploits in the pre-si environment.

CKey disclosure bug: This bug uncovered a sequence of transactions whereby SEC FW could "trick" the crypto-engine into thinking that it had deleted the CKey from a temporary register while the key was still available for the FW to read.

U/SU separation bug: This bug involved the interaction between an unaligned store and a prior store instruction and could be exploited by User FW to write to MMU registers.

In our experience, a variety of security requirements can be phrased as taint-propagation properties and be formally verified even in complex, real-world designs. It is also easier to debug counter-examples for these properties, as the flow of tainted information is clearly visible in the counter-example trace. A key advantage of this methodology is the ability to specify an intuitive "high-level" taint-propagation property and then have the tool search through the space of all transaction/instruction sequences to find issues. Both bugs we found involved sequences of seemingly unrelated instructions that would have been quite hard to find through manual review, simulation-based testing or even temporal logic assertions. Even when the formal tool is unable to scale, assumptions can be intelligently added to reduce the search space and yield constrained but useful proofs (e.g., see §III-C). For example, in the NoC Access Control scenario, our assumptions allowed us to prove that a significant subset of attacks were impossible.

V. Conclusions

In this work, we analyzed the security controller of a tablet SoC, formulated its critical security requirements as taint-propagation properties, and proved them using a formal verification tool. We described the properties proved and techniques to help the verifier scale. While the objective of this work was to test a new methodology, it can easily be extended to get coverage on security properties throughout the SoC. We believe our work demonstrates that taint-propagation properties are an intuitive and effective technique for verifying a large class of hardware security properties.

References

[1] Android Documentation: Camera.Face. https://developer.android.com/reference/android/hardware/Camera.Face.html, 2011.

[2] Google Wallet. http://www.google.com/wallet/, 2013.

[3] Widevine DRM. http://www.widevine.com/, 2013.

[4] David W. Palmer and Parbati Kumar Manna. An Efficient Algorithm for Identifying Security Relevant Logic and Vulnerabilities in RTL Designs. In *Proc. of HOST*, 2013.