SoC Security Evaluation: Reflections on Methodology and Tooling

Nassim Corteggiani, Giovanni Camurati, Marius Muench, Sebastian Poeplau, and Aurélien Francillon

***Abstract*—The growing complexity of Systems-on-Chip challenges our ability to ensure their correct operation, on which we rely for more and more sensitive activities. Many security vulnerabilities appear in subtle and unexpected ways in the interaction among blocks and across layers, where current verification tools fail at catching them or do not scale. For this reason, security evaluation still heavily relies on manual review. Inspired by the Hack@DAC19 contest, we present our reflections on this topic from a software and system security perspective. We outline an approach that extends the dynamic analysis of firmware to the hardware.**

***Index Terms*—security evaluation, System-on-Chip, dynamic analysis, HardFails**

I. Introduction

One of the driving factors for the growth of the electronics industry is its pervasiveness in other sectors. Embedded and connected devices are largely present in physical systems, such as cars and industrial plants, where they have a huge impact on safety. In addition, more and more sensitive activities, such as payments and voting, are carried out with digital equipment.

In this context, a major challenge is ensuring the correct operation of an System-on-Chip (SoC) and its software, while satisfying strict requirements in terms of functionality, cost, and time to market. Abstraction and separation of layers, in particular, hardware and software, are effective ways to cope with complexity and make the design and verification of such systems possible. However, many security vulnerabilities originate precisely from unexpected interaction between layers and components.

On the one side, traditional techniques fail at catching these cross-layer problems, or do not scale to real-world designs. On the other side, identifying novel methodologies is hard because researchers often do not have enough access to all the parts of the system. This is particularly true for proprietary hardware micro-architectures. Hack@DAC is a security contest designed to overcome this problem and stimulate research on the automation of security analysis. The contestants have to find software-controllable hardware vulnerabilities in an open-source design, in which the organizers have also injected real-world security bugs.

Based on our experience at Hack@DAC19, in this paper we present our thoughts on SoC security testing from the point of view of software and system security. We first review the typical goals as well as the constraints of a security analysis (Section II). Then we describe our methodology, as applied to the two rounds of the Hack@DAC19 contest (Section III). Finally, we explore research opportunities to reduce or eliminate manual aspects in SoC security analysis and to benefit from synergies between the hardware and the software testing communities (Section IV). We believe that our background in software and system security gives us an interesting perspective on the problem of hardware/software co-design.

II. Background

In this section, we describe the setup and the objectives of the security evaluation before discussing our methodology in the next section.

A. Security evaluation of SoCs and their firmware

The goal of any security evaluation is to establish a system's conformance to a specification of security properties.[1] In the context of SoCs, both hardware and software play an important role.

Software typically abstracts from the low-level details of the hardware it is running on. However, the validation of software against functional and security specifications needs to violate this abstraction for several reasons. First, an increasing number of security features rely on the interaction with complex hardware mechanisms, which cannot be blindly trusted. Even small hardware bugs may undermine the security of the software layers above. Second, embedded software is often intimately connected with hardware components such as the peripherals, and cannot be easily analyzed without taking this relation into account.

On the hardware side, designers need to take software concerns into account. While the traditional verification and testing flow is mature and guarantees a high level of functional correctness at the hardware level, conventional techniques fail to capture the cross-component, cross-layer interactions that may transform small hardware problems into catastrophic security flaws.

Bugs in the blocks that compose the memory interconnect are a typical scenario. For example, unprivileged code may gain access to an encryption key if a secure register is erroneously mapped to unprotected memory. Similar problems can occur when address ranges overlap, or when a peripheral with access to protected memory can be manipulated [1].

In this paper, we focus on a methodology that is targeted at finding precisely those bugs that arise from cross-layer interplay between software and hardware.

[1] In this paper, we use the terms *security analysis* and *security evaluation* to refer to the general process of investigating the level of security of a system, independently of the techniques and specifications used (e.g., formal verification, simulation, FGPA emulation, validation, dynamic analysis).

B. Analysis context

Security analysis may take place in different scenarios. Literature on embedded software security tends to focus on the black-box case, in which only the binary firmware (or the source at best) and the silicon device are available, but the internals of the hardware are unknown. On the other hand, chip manufacturers have access to the RTL code of the hardware design (provided that they do not use black-box IPs) but may not know the software that will run on it. Vendors providing integrated solutions have access to both firmware source code and hardware internals.

Likewise, the security specification of the system under test may or may not be available. Its level of detail can vary greatly, and security properties may be expressed in a formal or informal way. Ideally, system designers have access to models of the system and its required properties at a high level of abstraction. Finally, the goal of the analysis can vary from finding individual violations of security properties (e.g., in an adversarial context where a single vulnerability is sufficient to compromise a system) to the quest of full validation, i.e., proving the absence of violations under all circumstances.

In this paper, we take the point of view of third party security analysts who have access to an RTL description of the hardware, C source code developed for it, and informal specifications of the expected security properties. The goal of the analysis is to find as many security issues as possible in a limited amount of time, but the specification is not precise enough (and the time is not sufficient) to allow for formal verification. Hack@DAC provides an example of this setting for public research, as we explain in the following.

C. SoC and firmware at Hack@DAC19

Hack@DAC is a security contest that simulates the task of security analysts at a chip manufacturer under pressure to ship the product. Participants have to find security vulnerabilities in an SoC, with a focus on bugs that can be exploited from software, called HardFails [1]. The 2019 edition was based on Ariane, a 64-bit RISC-V processor that is able to boot Linux. The design has been extended with additional features, such as a password-protected JTAG debug port, an AES cryptographic engine, a secure ROM with secure registers, a peripheral to select encryption keys, and access control for mapped memory. The contestants had access to the RTL Verilog code, a toolchain and a testbench to simulate the execution of C programs, and a brief natural-language description of the security features of the system. For the finals, the firmware with the APIs to access the peripherals in the intended way was also available. In addition to any vulnerabilities that may already have been present in the system, the organizers injected several real-world vulnerabilities “donated” by hardware vendors.

III. Security evaluation methodology

In the following, we give a detailed description of the methodology we followed to to find bugs on the Hack@DAC platform.

Tool	C&RTL support	Sufficiently expressive	Analysis Complexity	Set-Up Time
FPGA	Y	Y	Low	Long
Verilator	Y	Y	Low	Short
Model Checking	Y	Y	High	Long
Theorem Prover	N	N	High	Very Long
KLEE	N	Y	High	Short

TABLE I
NON-EXHAUSTIVE COMPARISON OF STATE-OF-THE-ART ANALYSIS TECHNIQUES FOR SOC TESTING.

A. Requirements on tooling

The preliminary step of any security analysis consists in choosing the right tools for the investigation. In the given context, we formulated the following requirements:

1) Tools must support RTL, C and hand-written assembly code (e.g., Ariane SoC boot ROM). The system under test is composed of different blocks of hardware and software that interact with each other. These interactions are sometimes complex and may lead to security issues that expose the entire SoC. The trigger conditions associated to these kinds of bugs require putting multiple components of the system in a specific state, potentially involving firmware and various hardware blocks. Therefore, analysis tools need to support both RTL analysis and firmware execution.
2) We need an easy way to express security properties. Time constraints and the vague specification forbid elaborate formulations of expected behavior in a formal language.
3) Time is of the essence, so that we need to find bugs quickly. Given a security property and the platform under test, the tool should determine in a relatively short time and with high confidence whether the property holds. Due to time constraints, we cannot ask for stronger guarantees.
4) For the same reason, we need tools that are fast to set up. It is very difficult to estimate the efficiency of any single tool on a given design and security specification. Therefore, we chose to avoid tools that require a significant set-up time and rather allow for combinations of several tools.

B. Available tools

With these goals in mind, we compared available techniques; see Table I. It is important to remember that, due to time pressure, our goal is to show the *presence* of security violations, not to prove their *absence*. The latter is much more difficult because it needs to exhaustively reason about all possible states of the system. Discovering individual vulnerabilities, in contrast, is less time-consuming as we only need to find a single execution state in which a security property is violated. The intuition is that by iterating the process we approach a state that is indistinguishable from a fully validated system.

We quickly excluded theorem provers and model checking for several reasons. Both techniques generally assume deep knowledge of the system, require significant time to set up and, most importantly, they make it hard to express correctness properties over complex states, such as interactions between multiple hardware blocks. Furthermore, model checking is affected by the state-explosion problem, which we consider a severe obstacle when reasoning about a system as complex as an SoC in limited time.

We like to use software as a means of describing behavior at a high level of abstraction, which makes it easier to test security properties and push the system in a specific state: instead of considering low-level interactions in the electronic circuits, we focus on communication between logic blocks (e.g., AES engine, DMA, access control system) and use custom firmware to find cases where specific security properties fail. We settled on a software-centric workflow built around quick experiments with component interaction, driven by software.

C. Methodology

Our methodology for analyzing the security of an SoC comprises three main steps:

1) *Security properties.* A common challenge for all analysis techniques is the need to describe the desired security properties. Ideally, testers receive a specification containing a precise description of the security properties that the implementation should respect. But in reality, the specification is usually an ambiguous document mixing functionality and security requirements. Therefore, testers typically need to employ intuition, experience and a good amount of skepticism in order to develop hypotheses of potential failures. This step requires studying the literature, documentation, and the implementation (RTL and firmware) to identify suspicious control or data flows that could result in bugs. We complement this manual work with static analysis (Verilator sanity checks and symbolic execution of the firmware with predefined sanity checks). However, while firmware analysis tools can efficiently find memory corruption in firmware, they encounter limitations when searching bugs related to digital hardware components.
2) *Writing a Proof of Concept (PoC).* Once we have an hypothesis on a potential violation of a security property, we test this hypothesis in a two step process. First, a block of C code drives the hardware to reach the specific state that is assumed to be vulnerable. At this level of abstraction, it is easy to drive all hardware components as needed. Then our program checks whether the security property in question has indeed been violated. Again, the higher level of abstraction in software tests lets us conveniently express cross-component constraints.
3) *Running the PoC on a dynamic analysis platform.* To evaluate our PoC against the system under test, we use a cycle-accurate simulation of the RTL code with the goal of executing C code on the Ariane SoC.

Fig. 1 illustrates the methodology. While this methodology is by no means a way of guaranteeing the absence of bugs, we believe that it mirrors the reality in many cases where full verification is too costly to make sense economically.

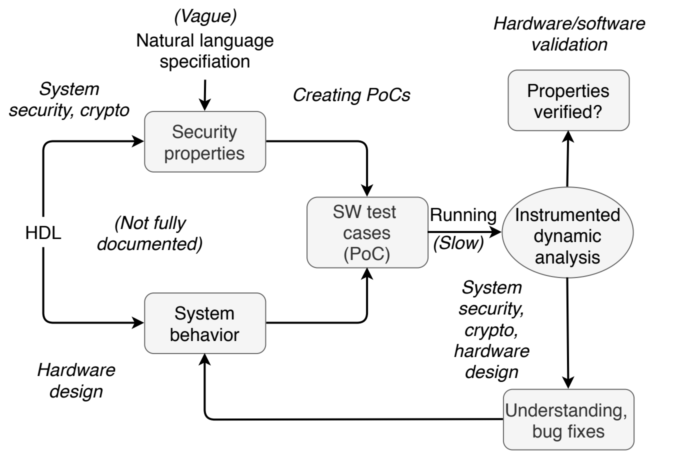

Fig. 1. Overview of the security evaluation methodology based on dynamic analysis. Manual work and human expertise from different fields are marked in italic.

D. Results from Hack@DAC19

We applied the presented methodology at the Hack@DAC19 competition and provide insights about the awarded scores and achieved results in the following. The competition had two different phases: the qualification (10 weeks) and the final (3 days). The design for the qualification has 44975 lines of SystemVerilog code, while the design for the final has 47282 lines of code. We participated as the academic team NOPS and lead the qualification phase for the first 5 weeks, concluding second (240 points) after the team Hackin' Aggies (300 points), and before the other 13 teams (less than 200 points). During the finals, we achieved the first place in the academic bracket with 330 points, just after the academic/industrial team Hackin' Aggies (465 points), and before the other 11 teams (less than 290 points).

The 2018 edition presented similar classes of bugs on a different platform. To the best of our knowledge, we found all those that were based on similar problems. A detailed list of the bugs can be found in [1], together with a description of the techniques used to find them. The authors observe that classic techniques often fail to capture cross-module cross-layer bugs, and that formal properties are often hard if not impossible to write, even when already knowing the bug. On the contrary, our test vectors in software were often straightforward to write, as this is a good layer to stress several blocks at the same time and to bring the system in the desired overall state. In total, we found 29 bugs not listed in [1] out of 32 reported bugs.

Not requiring neither complex tools nor a detailed knowledge of the hardware design, our methodology was well suited for a fast-paced competition, which mimics strict deadlines typically encountered in industrial settings. Our focus on system/software aspects was useful to find a variety of bugs on system level (e.g., wrongly configured access permissions), cryptographic engines (e.g., broken AES mode), and even vulnerabilities in the included firmware (e.g., privilege escalation via system calls). Given the structured approach of our methodology, we could quickly write test-vectors and create bug reports and fixes. Our approach tends to abstract the core (code execution) and focus on the peripherals (accesses to memory mapped registers), therefore we missed most bugs in the core.

IV. Research directions

Our approach to SoC security analysis centers around formulating hypotheses of potential weaknesses and verifying them with software. We believe that the use of software as a means of high-level expression facilitates cross-component and cross-layer evaluation, allowing us to try out different attacks on the system in short iterations. Generating hardware stimulus from from the software abstraction levels is efficient to detect software-exploitable hardware vulnerabilities, at the price of loosing granularity and level of control on specific hardware blocks. Indeed, from software we do not have the freedom to stimulate all the inputs of a block, but only those that are exposed. On the other hand, software can easily drive multiple interconnected blocks at the same time. Visibility is instead preserved if using cycle-accurate simulation of the RTL code. While this approach does not cover all types of hardware bugs, it can be used by security experts without deep knowledge of the design, to overcome the limitations of conventional techniques [1]. However, the process still requires significant amounts of manual work and a high degree of security expertise. Moreover, the cycle-accurate simulation of the full system is very slow and it limits the potential application of more complex software-based approaches. In this section, we discuss how future research could alleviate some of the burden on the analyst and facilitate the execution of the tests, in particular by combining established approaches from the hardware-design community with ideas from software security testing.

A. On abstraction

During a security analysis, especially when searching for cross-component bugs, analysts attempt to regard the system as a whole where components can be verified together. We believe that flaws that affect the security of an SoC via cross-component interaction can best be discovered when the system is viewed from as abstract a perspective as possible, even if the implementation error that introduced the vulnerability is restricted to a single component: a block may function correctly when tested individually while still compromising the entire system's security in the interplay with other components.

We therefore think that software is the right layer for checking the security of the system as a whole. Software-based tests should enhance, not replace, the tests at the hardware layer that are already customarily performed during SoC development. In the following, we illustrate approaches that can potentially simplify software-based testing.

B. Generating tests

One task that currently requires manual effort is the creation of test software. Analysts need to read the security specification (if available), then often interpret it to obtain a more precise formulation of the desired security properties. Only then can they formulate hypotheses where the system may fail to meet the requirements, and finally devise corresponding test software. Note that, in the context of cross-component vulnerabilities, the security specification should remain at the high level of abstraction that includes all components of the system. Refining it to the level of individual components and their implementation is useful for component testing but undesirable for the purpose of assessing the security of the system as a whole.

In general, the more precise the specification of the expected properties of the system, the easier it is to derive meaningful tests from it. At the extreme, a machine-readable specification could automatically be translated to test cases [2]. Less precise descriptions leave room for interpretation and thus require expert knowledge to be used in security testing.

Once an actionable formulation of the security properties is available, it can be used to assess whether the implementation meets the requirements. Manually designed test programs, as used in our methodology, are only one option. In this context, it is worth mentioning symbolic execution and fuzz testing [3], both of which are popular approaches in software testing: they check software by automatically exploring many possible paths through a program. However, while it is relatively easy to express security requirements in a software-only scenario, the same is not true when possibly faulty hardware components enter the picture [4]. We believe that a good specification could be used to drive the analysis carried out by such tools, helping with the difficulty of defining expected behavior. For example, symbolic execution could explore various interactions with the hardware, all the while checking that the security assertions put forth by the specification hold true in each tested case ([5], [6]).

C. Executing tests efficiently and effectively

Software-based tests have to be executed on some representation of the underlying hardware. Recent approaches for hardware simulation face the difficulty of scaling to increasingly complex designs [7]. In general, partitioning the system and describing blocks at different layers of abstraction help to find good trade-offs among execution speed, the ability to catch low-level flaws, and simplicity of introspection and debugging. Partitioning and abstraction of hardware has been especially prominent in recent advances in dynamic binary firmware analysis. In the following, we show how these concepts map to the conventional hardware approach, and we discuss how SoC analysis can benefit from them.

To manage complexity, the traditional design and validation flow of electronic systems follows a top-down approach. Abstract specifications are iteratively refined, gradually introducing partitioning into separate components and implementation details [8]. At each iteration and layer, extensive validation is performed to ensure the correctness of the implementation. Partitioning allows testing a detailed component against blocks described at higher levels of abstraction. At the end, the components are integrated into a final product.

In contrast, many approaches developed for dynamic firmware analysis take the point of view of a security expert who analyses an already finalized commercial product, where part of the system may be unknown to the analyst. As a result, recent methodologies often use a bottom-up approach that reintroduces partitioning and abstraction. In this context, the CPU of an SoC is commonly emulated or completely replaced by a more abstract virtual machine, and the executed code is often translated into an intermediate representation at a desired level of abstraction [9]. However, other parts of the SoC under test (e.g., peripherals) are typically opaque to the analyst in this scenario and thus cannot be emulated. To overcome this issue, modern tooling allows either to specify models for the behavior of unknown parts of the hardware [10], [11], or deploys near real-time forwarding mechanisms between the emulator and the real silicon for hardware accesses [12].

We believe that SoC testing and validation can greatly benefit from these two approaches—abstracting hardware components and selectively forwarding to real hardware while using an emulator—as shown in [6]. Additionally, the capabilities of existing tools (e.g., [11] or [12]) could be easily extended to cooperate with peripherals or other blocks at the RTL level. Such extended tools would then serve as a natural platform for software-based security testing of SoCs. They were already designed with a focus on security and include a number of automated security checks (e.g., checks for memory corruptions). Moreover, they allow for more automated exploration techniques such as symbolic execution and fuzzing.

V. Conclusion

We have outlined how software-based tests are an effective approach for the security evaluation of an SoC. The software abstraction is very convenient, as it bridges the gap between the high-level security properties of the system and the low-level interactions across hardware components, as well as the gap between system and software security and hardware design and validation. Software-based approaches and tools are well known to software experts, and they are generally easy to set up starting from the final product, even without in-depth knowledge of the underlying hardware and without expensive simulation tools. Therefore, they could lower the entry barrier for analyzing the hardware components of an SoC, and facilitate the dissemination of knowledge across research communities.

Future work could address the optimization of software-based methods to the hardware case, in particular regarding test generation and automated design exploration, but also with the goal of efficient execution. Software security tools often take into account that analysts may not have full access to the design, as the hardware platforms running the software to be tested are often proprietary. However, these tools could greatly benefit from the availability of RTL code and models of the hardware components. The security and hardware communities could work together to create more and more accessible SoC platforms with representative vulnerabilities, to lower the barrier for developing new approaches and tools.

References

[1] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *28th USENIX Security Symposium (USENIX Security)*, 2019.

[2] A. Reid, "Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification," *Proceedings of the ACM on Programming Languages (PACMPL)*, 2017.

[3] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, 2018.

[4] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[5] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-End Concolic Testing for Hardware/Software Co-Validation," in *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019.

[6] H. Li, D. Tong, K. Huang, and X. Cheng, "FEMU: A firmware-based emulation framework for SoC verification," in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[7] I. B. Mahapatra, S. Natarajan, Nalesh S, and S. K. Nandy, "SIMAAH: RTL simulation accelerator for complex SoC's," in *IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2015.

[8] C.-Y. R. Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang, "SoC HW/SW Verification and Validation," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2011.

[9] G. Hernandez, F. Fowze, D. J. Tang, T. Yavuz, P. Traynor, and K. R. Butler, "Toward Automated Firmware Analysis in the IoT Era," *IEEE Security & Privacy*, 2019.

[10] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *22nd USENIX Security Symposium (USENIX Security)*, 2013.

[11] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: A Multi-target Orchestration Platform," in *Workshop on Binary Analysis Research (BAR)*, 2018.

[12] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium (USENIX Security)*, 2018.

Nassim Corteggiani is a PhD student at EURECOM, where he is member of the S3 research group. His current research interests include dynamic security testing of firmware and hardware. He holds a MSc in Information Security and Cryptology from Limoges University.
Contact: nassim.corteggiani@eurecom.fr

Giovanni Camurati is a PhD student in system security at EURECOM. His research focuses on the security impact of the interactions between electromagnetic side channels and radio transceivers in modern devices with wireless capabilities. He is also interested in hardware design and dynamic analysis of firmware. He holds a MSc in Electronic Engineering from Politecnico di Torino and from Télécom ParisTech.
Contact: giovanni.camurati@eurecom.fr

Marius Muench is a postdoctoral researcher at VU Amsterdam and lead author and maintainer of the avatar^2 project. He conducted his PhD studies at EURECOM where he systematically tackled challenges for dynamic binary firmware analysis. Additional interests include binary exploitation and software-based defenses.
Contact: m.muench@vu.nl

Sebastian Poeplau is a PhD student at EURECOM. His research interests include symbolic execution and programming languages. He holds an M.Sc. in computer science from the University of Bonn, Germany.
Contact: sebastian.poeplau@eurecom.fr

Aurélien Francillon is an associate professor in the networking and security department at EURECOM. Prior to that, he obtained a PhD from INRIA Grenoble and spent 2 years as a postdoctoral researcher in the System Security Group at ETH Zurich. His research interests are in security of embedded systems as well as software security and telecom fraud.
Contact: aurelien.francillon@eurecom.fr