Hardware Penetration Testing Knocks Your SoCs Off

Mark Fischer, Fabian Langer, Johannes Mono, Clemens Nasenberg, Nils Albartus
Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany
{first.last@rub.de}

Abstract—Today's society depends on interconnected electronic devices, which handle various sensitive information. Due to the knowledge needed to develop these devices and the economic advantage of reusable solutions, most of these systems contain Third-Party Intellectual Property (3PIP) cores that might not be trustworthy. If one of these 3PIP cores is vulnerable, the security of the entire device is potentially affected. As a result, sensitive data that is processed by the device can be leaked to an attacker. Competitions like Hack@DAC serve as a playground to develop and examine novel approaches and computer-aided tools that identify security vulnerabilities in System-on-Chip (SoC) Register-Transfer-Level (RTL) designs. In this paper, we present a successful divide and conquer approach to test SoC security which is illustrated by exemplary RTL vulnerabilities in the competition's SoC design. Additionally, we craft real-world software attacks that exploit these vulnerabilities.

Index Terms—Hack@DAC 2019, Hardware Penetration Testing, RTL Bugs, SoC Design, Hardware Security, RISC-V CPU

1 INTRODUCTION

ELECTRONIC DEVICES are ubiquitous today and build the foundation of our interconnected world. They process sensitive information and are therefore a highly valuable target for attackers. Thus, securing these devices is an important challenge for researchers as well as for other stakeholders such as the electronics industry.

Recently found security flaws such as Spectre and Meltdown highlight the severe consequences security bugs in hardware can have, especially considering the lifetime of hardware. Implementing these protective measures can be a challenging task: Third-Party Intellectual Property (3PIP) cores are used that might not be correctly verified in terms of security as current verification methods mostly focus on the functionality of the design. As a consequence, security verification methods are a very relevant and ongoing research topic. A framework for security verification was proposed by Wang et al. [1] in 2012. They combined formal and functional security verification in order to improve the detection rate of security issues. In a different approach, Zhang et al. [2] extended Verilog with security-focused language constructs to aid developers in creating more secure designs. Ferraiuolo et al. [3] enhanced this approach and were able to verify a complex *ARM Trustzone* implementation.

To encourage further SoC security research Hack@DAC is a hardware Capture The Flag (CTF) competition during which teams are investigating System-on-Chip (SoC) designs to find security vulnerabilities. We participated in the Hack@DAC 2019 competition. In this paper, we present our methods as well as our thoughts on the current state of hardware security.

This paper is structured as follows: Section 2 introduces the reader to hardware security in general, the Hack@DAC 2019 competition and the Ariane SoC. In Section 3, we explain our approach, which divides the design into different domains: system architecture, Intellectual Property (IP) core architecture & implementation, and Central Processing Unit (CPU). For each domain, we provide examples of security vulnerabilities discovered during the competition. Finally, in Section 4, we discuss techniques to prevent bugs for each domain and conclude with a summary of the approach's effectiveness and our learnings and takeaways from the competition.

2 BACKGROUND

This section establishes testing concepts and puts them into context for SoC security testing. Afterward, we introduce the Hack@DAC 2019 competition and utilized hardware setups as well as tools.

2.1 General Approaches for Testing Hardware Security

In general, there are three models to investigate the security of an underlying system: black box, white box and grey box testing.

In black box testing, test cases are derived from the specification. Testing the system is mostly limited to providing a set of inputs and comparing the output of the system to a known good result. As a consequence, it is possible that the system is working correctly in the given test scenario but still has untested functionality in its internals due to insufficient test coverage [4].

In white box testing, the internal functioning of the system is checked. This includes thorough testing and analysis of the internal logic of the system, in particular harder to prove properties. One white box technique used is manual code inspection [4]. It is an effective, reading-based mechanism for detecting flaws in the source code. The main idea is that a trained professional is the most powerful bug detector because determining whether source code contains imperfections or not is in general not decidable by a machine [5]. However, this professional needs in-depth

knowledge of the system which is inspected. Therefore, manual code inspection is not scalable for complex systems.

The combination of white and black box testing is known as grey box testing. One example of grey box testing is formal verification which is a method to prove correctness or falsity of a system by mathematically analyzing the space of possible behaviors. The system's intended behavior is defined by a formal specification but also incorporates implementation details of the internal functioning. Correctness or falsity is proved with the aid of formal mathematical methods. A prevalent tool used for the formal verification of hardware designs is *Coq*. Coupet-Grimal et al. [6] present a use case for the verification of a left-to-right comparator. The authors demonstrate how to define specifications, how to perform the proof of correctness and how to synthesize a certified circuit from its specification. Even though formal verification is a powerful technique the verification of large SoCs is an extensive task and requires significant computational resources, therefore rendering it practically infeasible for many designs.

2.2 Hack@DAC 2019

Within the Hack@DAC 2019 hardware CTF competition[1], teams of researchers identify hardware bugs in real-world open-source SoC Register-Transfer-Level (RTL) designs. The provided SoC designs have been modified by the organizers in a collaboration with industry partners, who injected bugs and added security features for this competition. A variety of these modifications is shown in a case study presenting the results for Hack@DAC 2018 [7].

The competition consisted of two phases, an alpha phase that served as a qualification round, and the final beta phase. Both phases featured a RISC-V Ariane SoC design, which is provided as Verilog code.

Teams score by submitting reports on security bugs, which allow to bypass security features or to compromise protected SoC assets. Points are awarded based on different factors, such as a correct test description and a correct mitigation proposal.

For this purpose, the participants took the role of different adversaries who:

- Execute software with user-level privileges to escalate the privilege-level by exploiting a bug.
- Possess the device and are therefore able to tamper with it.
- Are authorized to debug the production device.

2.2.1 *Ariane Core / SoC*

The Hack@DAC organizers provided two different setups, one for the alpha phase and one for the beta phase.

The SoC of the alpha phase (Fig. 1) consists of a modified Ariane CPU and connected peripherals. The Ariane CPU is a Linux-capable 64-bit core and has 3 different privilege levels: user, supervisor and machine level. It utilizes a 6-stage pipeline and implements the 64-bit RISC-V instruction set. Furthermore, branch prediction with speculative execution is used. The peripherals represent Third-Party Intellectual Property (3PIP) cores and are connected via an Advanced

1. https://hack-dac19.trust-sysec.com

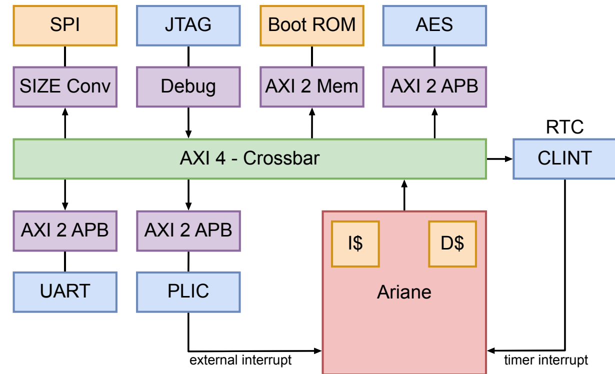

Fig. 1. Alpha phase SoC setup of the Hack@DAC competition.

Extensible Interface (AXI) 4 crossbar. The Advanced Encryption Standard (AES) crypto module is implemented in Counter (CTR)-mode. For each encryption, the initial vector is calculated through a nonce and a counter variable, which increases after every encryption. The Joint Test Action Group (JTAG) interface provides opportunities for debugging and emulation. The Core Local Interrupt Controller (CLINT) and Platform Level Interrupt Controller (PLIC) modules handle the system's interrupts. CLINT deals with local interrupts, while PLIC manages the global interrupts. Finally, there is a Serial Peripheral Interface (SPI) module used for small peripherals and a Universal Asynchronous Receiver/Transmitter (UART) module used for serial communication.

Access to the IP cores over the bus is protected with access control. The access control configuration bits, as well as the JTAG and AES key, are stored in a secure Read-Only Memory (ROM) which itself is connected to the bus.

The SoC of the beta phase is also based on the Ariane core. Additionally to the peripherals of the alpha SoC, a Secure Hash Algorithm (SHA) module performing cryptographic hash functions and a Direct Memory Access (DMA) module offering faster data throughput over the AXI bus system have been added.

2.2.2 *Tooling*

During both phases, the teams are free to use tools and techniques of their choice. However, the teams are advised to use a Hardware Description Language (HDL) simulation environment, because it is more accessible than formal verification.

Within the simulation environment, the participants develop *C* code to test the features of the SoC design and to exploit vulnerabilities.

The SoC was provided in the form of HDL sources and the environment supplied a ready-to-run testbench. This environment was prepared to be executed in the *QuestaSim* cycle-accurate simulator environment from *Mentor Graphics*. This enabled us to directly inspect logic transitions in suspicious regions. By compiling software tests, we could inspect the resulting effects in the digital logic. For every instruction, the complete SoC is simulated, which facilitated the investigation of propagating signals. On the downside, the simulation is computationally expensive and needs a long time to execute even simple software.

For faster execution of exploiting code, the open-source tool *Verilator* can be used. *Verilator* compiles synthesizable *Verilog* code into a *C++\SystemC* model to gain a speed advantage compared to other cycle-accurate simulators. A *C++* wrapper, instantiating the top module of the design and defining the testbench, is linked with the converted *Verilog* files. The resulting executable performs the simulation.

For all developed exploits, the code had first to be compiled in order to be simulated on the SoC. Therefore, the RISC-V *C* and *C++* cross-compilers are provided generating a RISC-V compatible Executable and Linkable Format (ELF) file from *C* or *C++* code. Subsequently, the resulting ELF file can be committed to the simulation environment which runs the code on the provided SoC.

Spike, an Instruction Set Architecture (ISA) simulator, can be used to provide simulation of the compiled *C* code. It uses an underlying model of the RISC-V specification to run assembler directly in the software and thus is faster than cycle-accurate simulation. Although Spike cannot be used directly to find bugs in the SoC but quickly testing software execution greatly reduced exploitation development time.

Besides simulation, other techniques are conceivable. One example is to emulate the SoC design on a supported Field Programmable Gate Array (FPGA) with the advantage of running code much faster than in the simulation environment. For well-defined tests, this helps to drastically reduce the time to run test cases, but it complicates the investigation of the cause as probing many signals of an FPGA with logic analyzers is resource-intensive. However, such an FPGA is not provided during the competition.

3 Real-World Examples

This chapter introduces our bug detection approach by presenting different kinds of real-world vulnerabilities which are placed in the competition's SoC design.

Our general approach is based on the principle of divide and conquer. Hence, we divide the SoC into its system architecture, its IP cores, and its CPU. Subsequently, we analyzed which security features each of these parts offer and which security properties are supposedly provided. Based on these security properties and our experience in software vulnerabilities, we defined hypotheses on vulnerabilities that might be present in each part. Afterward, we performed manual code inspection to find indicators which might confirm the hypotheses. When a potential vulnerability has been identified, we developed an exploit in *C/C++* that leverages the vulnerability to break the security properties of the SoC.

The cycle-accurate simulation with *QuestaSim* was our preferred way of executing software on the SoC. This allowed us to see the effects in the digital logic and to adapt the exploits to target the identified vulnerabilities. It offers an easy way for a quick overview of all internals at every point in time.

3.1 System Architecture

SoCs consist of multiple IP cores, which have to be carefully integrated to avoid security weaknesses. The alpha version of the SoC has — amongst other modules — a secure ROM module. The secure ROM module contains two access control bits, the JTAG key and the AES key and is connected to the AXI bus. The AXI bus has access control to prevent unauthenticated access to the secure ROM.

Due to a misconfiguration of the access control, the secure ROM is readable by an unauthenticated user. Therefore, limited by the AXI bus width, an attacker in the possession of the device can read the lowest 64 bit of the access control bits, the JTAG key and the AES key.

The keys are particularly interesting to an attacker. For the JTAG key, only 32 bits are used and therefore an attacker is able to access the device over JTAG with privileges. For the AES key, the knowledge of the key severely weakens the encryption. Additionally, the complete AES key is readable in the AES module by an unauthenticated user and therefore the AES does not provide any confidentiality, one of the key concepts of information security.

SoCs can also have firmware which has to be carefully designed to avoid security weaknesses. The beta version of the SoC has — unlike the alpha version — a firmware. One of the firmware's tasks is to check (in form of a self-test) whether the cryptography engines (AES and SHA) are working correctly.

Due to an incorrectly implemented self-test for the SHA engine, its function is not verified correctly. Therefore, an attacker who is in the possession of the device can manipulate the SHA engine of the core without being detected by the firmware. A manipulated SHA engine does not provide integrity, another key concept of information security. It can also be disabled and thus does not provide availability, the third key concept of information security.

3.2 IP Core Architecture

Besides security vulnerabilities arising from the integration of multiple IP cores, there might be architectural weaknesses in the IP core architectures itself. Thus, all core architectures which are relevant for the security features have to be tested. These tests include the basic functionality of the core as well as every possible corner case that might occur. One example of this kind of vulnerability is described in the following.

The provided SoC design contains an AES-192 cryptography module which is specified to implement an AES engine. Based on the provided specification list, the AES engine takes a 192-bit key, a 128-bit input data, and a 128-bit initialization vector and produces a 128-bit output. The encryption/decryption itself is implemented in Counter (CTR) block cipher mode.

The CTR-mode uses a block cipher as a stream cipher. The input to the block cipher is an initialization vector IV concatenated with a counter CTR, which is increased after every block. The output of the block cipher is XORed with the plaintext/ciphertext. Figure 2 and the following equations show that principle.

$$\text{Encryption: } C_i = AES_k(IV||CTR_i) \oplus P_i$$
$$\text{Decryption: } P_i = AES_k(IV||CTR_i) \oplus C_i$$

While analyzing the AES engine, we did not find any code which implements a counter as described above. Hence, we suspected that there might be a bug or a missing line of code, which should increase the CTR. To confirm

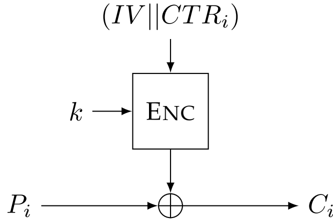

Fig. 2. Block Cipher Counter (CTR) Mode Encryption

our hypothesis, we implemented a short C program, which encrypts an arbitrary block twice in a row, and simulated it on the SoC design. If the CTR had been implemented correctly, the corresponding ciphertexts would differ and the hypothesis would be wrong. However, the corresponding ciphertexts were equal, therefore the outputs of the AES cipher were equal. This concludes that the AES inputs $(IV||CTR_i)$ and especially CTR_i were equal for both encryptions.

This bug results in a downgrade of the encryption to a stream cipher with a fixed key K. The principle is shown in the following equations.

$$\text{Encryption: } C_i = K \oplus P_i$$
$$\text{Decryption: } P_i = K \oplus C_i$$

Such encryption can be easily broken by the following exploit we developed:

The exploit takes a pair of plaintext P_0 and corresponding ciphertext C_0, XORs them, and outputs a fixed key K.

$$C_0 \oplus P_0 = AES_k(IV||CTR_0) = K$$

Afterward, we can use K to decrypt an unknown ciphertext C_1 to get the corresponding plaintext P_1:

$$K \oplus C_1 = AES_k(IV||CTR_0) \oplus AES_k(IV||CTR_1) \oplus P_1$$

As we know from our observations, $CTR_1 = CTR_0$. Therefore, the equation above can be written as:

$$K \oplus C_1 = AES_k(IV||CTR_0) \oplus AES_k(IV||CTR_0) \oplus P_1$$

Which is equal to:

$$K \oplus C_1 = P_1$$

In conclusion, one misconception about the cipher mode resulted in a downgrade to a weak stream cipher, which can be easily broken by one pair of plaintext and ciphertext. As a result, we developed an exploit, which allows an attacker to read all data that is encrypted by the system.

3.3 IP Core Implementation

Apart from architectural errors in IP cores as seen in Section 3.2, implementation errors can impair the functionality and the security of the system. These implementation errors are coding errors, for example, related to syntax, logical flaws or human failure.

A bug, related to the implementation of an IP core, can be found in the SoC's JTAG module. According to the competition's remarks, the JTAG is protected by a key and is only enabled when the correct key is provided. In fact, the JTAG *Read* function is only executed when the correct key is given, while the *Write* function can be utilized regardless of the key.

```
110 if ((dm::dtm_op_t'(dmi.op) ==
    dm::DTM_READ) && (pass_chk == 1'b1))
    begin
111     state_d = Read;
112 end else if ((dm::dtm_op_t'(dmi.op) ==
      dm::DTM_WRITE)) begin
113     state_d = Write;
```

Listing 1. *If-*/ *else if*-construct for *Read* and *Write* function from the JTAG module (*dmi_jtag.sv*).

In the module's code (Listing 1) the *pass_chk* variable is set through the correct key. However, it is only checked in the *if*-construct for the *Read* function (line 110), but not in the following *else if*-condition (line 112), which is responsible for the execution of the *Write* function. So the *Write* function is independent of the key and therefore it is not secured against unauthorized usage.

3.4 CPU

A big part of most SoC designs is a 3PIP CPU core. Often developers license other CPUs like the *ARM* product families to use these as part of their design. CPUs are large projects and therefore hard to understand. Hence, the developer has to trust and use the verification methods of the CPU vendor and thus has to adapt to the vendor's verification ecosystem. Open-source CPU designs with open-source ISAs provide new opportunities and enable the developer to use its own verification framework [8]. It is hard to check for problems that emerge from architectural concepts which is evident by the large number of CPU vendors that were affected by the Meltdown and Spectre vulnerabilities.

Although verification, especially functional verification, already has very high standards, multiple security challenges arise during the design of CPUs. This is also observable for the provided beta SoC. The Ariane RISC-V processor has multiple privilege levels: user level, supervisor level and machine mode. Each privilege level has different access control rights to the memory regions of the peripherals. With machine privilege level, a program would be able to change the access rights to get access to every peripheral and therefore an attacker is interested in escalating their privilege.

During the finals of the competition, we were able to overwrite memory regions of peripherals with an unauthenticated user. The reason for this is, to the best of our knowledge, that the core privilege level and the fabric access control are not properly synchronized. This renders the security mechanism of protected calls useless and enables us to change secure registers like the AES key.

4 COUNTERMEASURES

During the Hack@DAC 2019 competition, we discovered multiple bugs in the provided SoC. For each bug, we submitted possible countermeasures that would have prevented the specific security vulnerability. In the following, we discuss generalized approaches to prevent the types of security vulnerabilities like the ones identified during competition.

4.1 System Architecture Discussion

As shown in Section 3.1, security bugs in the system architecture can have a severe impact on the key concepts of information security: confidentiality, integrity, and availability. Although testing can minimize the amount of security bugs, a multitude of corner cases are created when combining components into a complex system. Therefore, testing for system architecture bugs is a hard task. In general, more research has to be done to provide developers with the tools to combine IP cores and software components such as firmware into a secure system.

One possible solution would be to create standardized communication protocols for security-relevant IP cores that can formally be proven secure. This reduces the problem to test for the correct functioning of the protocol. A disadvantage of this approach is that a protocol has to be developed for each unique scenario.

Another possible solution is to prevent unwanted data-flow at design time with additional language constructs. An example is SecVerilog [2], a HDL in which wires can be assigned security levels and data is only allowed to flow towards an equal or higher security level. Although this might improve the security of data-flow in general, it does not completely eliminate the possibility of security bugs as it is still the responsibility of the developer to use the additional language constructs correctly.

4.2 IP Core Discussion

As shown in Section 3.2 and Section 3.3, the architecture and the implementation of security-relevant IP cores may introduce weaknesses, that either occur due to a lack of knowledge or by mistake. Therefore, SoC vendors have to test the 3PIP core architectures extensively. These tests should be implemented by a team of security experts and SoC designers, which are familiar with all corner cases of the security features and with hardware penetration testing in general.

Customized IP core tests should be complemented by automated testing methods. Tests can be provided by the IP core developers but should be reviewed by another party. Furthermore, testing should be applied before integration in form of stand-alone testing of modules and after integration to verify functionality and security when interacting with the other components of the SoC as discussed in Section 4.1.

Ideally, tests are combined into an independent security verification framework, that provides reliable tests for all established security features. As a result, SoC vendors would have a dependable tool to test 3PIP cores.

Guo et al. [9] developed a similar approach, a Proof-Carrying Hardware (PCH) Framework, to ensure the trustworthiness of IP cores. The framework validates the cores using theorem proving and equivalence checking to provide high-level protection of IP cores. Thus, the PCH framework ensures that the hardware implementation of the 3PIP core is equivalent to its design specification. One year later, Guo et al. [10] extended their approach by combining the method of formal verification with the technique of model checking.

4.3 CPU Discussion

As shown in Section 3.4, the CPU introduced leakage vulnerabilities which are inflicted by a faulty privilege level switch. Bugs that originate from the interaction of the external bus infrastructure are hard to test for the developer. The verification alone requires more effort than designing the entire system. Therefore, to verify the ISA, the designers base their verification efforts on test suites by the CPU supplier. Incorrect interactions with other peripherals increase the simulation effort and require a better testing strategy than random testing. Functional verification can offer an improved approach but is still an ongoing topic in research [11]. The needed test methodologies are closely coupled to the system architecture implications discussed in Section 3.1. The formal verification of information flow as shown by Subramanyan et al. [12] could also prevent leakage of secure data.

5 Conclusion

In this paper, we present our vulnerability detection method and share our experience from participating in the Hack@DAC 2019 competition. During the competition, we developed a divide and conquer approach to detect vulnerabilities in SoC designs. This approach splits the system into different domains and divides it into its components. Consequently, security-relevant components are analyzed separately and vulnerabilities can be detected in a structured manner. To illustrate our approach, we provide examples of different vulnerabilities in diverse components and show how these can be leveraged to break the security properties of the SoC design.

The approach proved itself to be successful. The effectivity of prepared test cases for specific 3PIP cores, such as the AES engine, is shown. However, it is based on manual code inspection and therefore not scalable for complex systems. Nevertheless, in addition to automatic tests, critical components should be tested manually to achieve the best coverage possible. Hence, a security framework offering such tests for all kinds of IP cores could assist SoC vendors, to verify the security of 3PIP cores. Such a framework could be developed using the approach we presented. Additionally, other systematic methods, e.g. formal verification, should be considered to verify the security of 3PIP cores and the system architecture itself.

Our key takeaway is that full test coverage is hardly possible with just manual code analysis, especially with the rising complexity of IP cores. However, a good testing concept with various different test cases can detect a large number of (security) critical bugs.

Acknowledgments

This work was supported through ERC grant 695022.

References

[1] W. Wang, Q. Zeng, and A. P. Mathur, "A Security Assurance Framework Combining Formal Verification and Security Functional Testing," in *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*, A. Tang and H. Muccini, Eds. IEEE, 2012, pp. 136–139. [Online]. Available: https://doi.org/10.1109/QSIC.2012.34

[2] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds. ACM, 2015, pp. 503–516. [Online]. Available: https://doi.org/10.1145/2694344.2694372

[3] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 555–568. [Online]. Available: https://doi.org/10.1145/3037697.3037739

[4] M. Ehmer and F. Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, 06 2012.

[5] T. Nakamura, L. Hochstein, and V. R. Basili, "Identifying Domain-Specific Defect Classes Using Inspections and Change History," in *2006 International Symposium on Empirical Software Engineering (ISESE 2006), September 21-22, 2006, Rio de Janeiro, Brazil*, G. H. Travassos, J. C. Maldonado, and C. Wohlin, Eds. ACM, 2006, pp. 346–355. [Online]. Available: https://doi.org/10.1145/1159733.1159785

[6] S. Coupet-Grimal and L. Jakubiec, "Coq and Hardware Verification: A Case Study," in *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, ser. Lecture Notes in Computer Science, J. von Wright, J. Grundy, and J. Harrison, Eds., vol. 1125. Springer, 1996, pp. 125–139. [Online]. Available: https://doi.org/10.1007/BFb0105401

[7] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. K. Kanuparthi, H. Khattri, J. M. Fung, A. Sadeghi, and J. Rajendran, "When a Patch is Not Enough - HardFails: Software-Exploitable Hardware Bugs," *CoRR*, vol. abs/1812.00197, 2018. [Online]. Available: http://arxiv.org/abs/1812.00197

[8] P. D. Schiavone, E. Sánchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study," in *IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2018, Verona, Italy, October 8-10, 2018*. IEEE, 2018, pp. 43–48. [Online]. Available: https://doi.org/10.1109/VLSI-SoC.2018.8644818

[9] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-Silicon Security Verification and Validation: A Formal Perspective," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015, pp. 145:1–145:6. [Online]. Available: https://doi.org/10.1145/2744769.2747939

[10] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable SoC Trust Verification using Integrated Theorem Proving and Model Checking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*, W. H. Robinson, S. Bhunia, and R. Kastner, Eds. IEEE Computer Society, 2016, pp. 124–129. [Online]. Available: https://doi.org/10.1109/HST.2016.7495569

[11] M. Jenihhin, X. Lai, T. Ghasempouri, and J. Raik, "Towards Multidimensional Verification: Where Functional Meets Non-Functional," in *2018 IEEE Nordic Circuits and Systems Conference, NORCAS 2018: NORCHIP and International Symposium of System-on-Chip (SoC), Tallinn, Estonia, October 30-31, 2018*, J. Nurmi, P. Ellervee, J. Mihhailov, M. Jenihhin, and K. Tammemäe, Eds. IEEE, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NORCHIP.2018.8573495

[12] P. Subramanyan and D. Arora, "Formal Verification of Taint-propagation Security Properties in a Commercial SoC Design," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, G. P. Fettweis and W. Nebel, Eds. European Design and Automation Association, 2014, pp. 1–2. [Online]. Available: https://doi.org/10.7873/DATE.2014.326

Mark Fischer received his B.Sc. degree in Computer Science from RWTH Aachen University, Germany, in 2018. He is currently working towards his M.Sc. degree in IT Security at Ruhr University Bochum, Germany. He participated in the Hack@DAC 2019 competition as a member of *NotATrojan*.

Fabian Langer received the B.Sc. degree in IT Security from Ruhr University Bochum, Germany, in 2015, and the M.Sc. degree in 2019. After participation in the Hack@DAC 2019 competition as a member of *NotATrojan*, he started working as an IT Security expert at TÜV Informationstechnik GmbH, TÜV NORD GROUP. His field of activity includes artificial intelligence and hardware evaluation.

Johannes Mono received his B.Sc. degree in IT Security from Ruhr University Bochum, Germany, in 2019 and is currently working towards his M.Sc. degree in IT Security. In his freetime, he likes to participate in Capture The Flag competitions with the team FluxFingers and participated in the Hack@DAC 2019 competition as a member of *NotATrojan*.

Clemens Nasenberg received his B.Eng. degree in Communications Engineering from Ulm University of Applied Sciences, Germany, in 2015, worked as an FPGA Engineer and is currently working towards his M.Sc. in Electrical Engineering at Ruhr University Bochum, Germany. He participated in the Hack@DAC 2019 competition as a member of *NotATrojan*.

Nils Albartus received the B.Sc. degree in IT Security from Ruhr University Bochum, Germany, in 2016, and the M.Sc. degree in IT Security from Ruhr University Bochum, Germany, in 2018. He is currently working towards the Ph.D. degree in the group for Embedded Security, under the supervision of C. Paar. His research interests include reverse engineering of hardware and embedded software systems. He supervised the *NotATrojan* team during the Hack@DAC 2019 competition.