HardFails: Insights into Software-Exploitable Hardware Bugs

Ghada Dessouky and David Gens, ***Technische Universität Darmstadt;*** **Patrick Haney and Garrett Persyn,** ***Texas A&M University;*** **Arun Kanuparthi, Hareesh Khattri, and Jason M. Fung,** ***Intel Corporation;*** **Ahmad-Reza Sadeghi,** ***Technische Universität Darmstadt;*** **Jeyavijayan Rajendran,** ***Texas A&M University***

https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

HardFails: Insights into Software-Exploitable Hardware Bugs

Ghada Dessouky[†], David Gens[†], Patrick Haney[*], Garrett Persyn[*], Arun Kanuparthi[°], Hareesh Khattri[°], Jason M. Fung[°], Ahmad-Reza Sadeghi[†], Jeyavijayan Rajendran[*]
[†]*Technische Universität Darmstadt, Germany.* [*]*Texas A&M University, College Station, USA.*
[°]*Intel Corporation, Hillsboro, OR, USA.*
ghada.dessouky@trust.tu-darmstadt.de, david.gens@trust.tu-darmstadt.de,
prh537@tamu.edu, gpersyn@tamu.edu, arun.kanuparthi@intel.com,
hareesh.khattri@intel.com, jason.m.fung@intel.com,
ahmad.sadeghi@trust.tu-darmstadt.de, jv.rajendran@tamu.edu

Abstract

Modern computer systems are becoming faster, more efficient, and increasingly interconnected with each generation. Thus, these platforms grow more complex, with new features continually introducing the possibility of new bugs. Although the semiconductor industry employs a combination of different verification techniques to ensure the security of System-on-Chip (SoC) designs, a growing number of increasingly sophisticated attacks are starting to leverage *cross-layer bugs*. These attacks leverage subtle interactions between hardware and software, as recently demonstrated through a series of real-world exploits that affected all major hardware vendors.

In this paper, we take a deep dive into microarchitectural security from a hardware designer's perspective by reviewing state-of-the-art approaches used to detect hardware vulnerabilities at design time. We show that a protection gap currently exists, leaving chip designs vulnerable to software-based attacks that can exploit these hardware vulnerabilities. Inspired by real-world vulnerabilities and insights from our industry collaborator (a leading chip manufacturer), we construct the first representative testbed of real-world software-exploitable RTL bugs based on RISC-V SoCs. Patching these bugs may not always be possible and can potentially result in a product recall. Based on our testbed, we conduct two extensive case studies to analyze the effectiveness of state-of-the-art security verification approaches and identify specific classes of vulnerabilities, which we call *HardFails*, which these approaches fail to detect. Through our work, we focus the spotlight on specific limitations of these approaches to propel future research in these directions. We envision our RISC-V testbed of RTL bugs providing a rich exploratory ground for future research in hardware security verification and contributing to the open-source hardware landscape.

1 Introduction

The divide between hardware and software security research is starting to take its toll, as we witness increasingly sophisticated attacks that combine software and hardware bugs to exploit computing platforms at runtime [20, 23, 36, 43, 45, 64, 69, 72, 74]. These cross-layer attacks disrupt traditional threat models, which assume either hardware-only or software-only adversaries. Such attacks may provoke physical effects to induce hardware faults or trigger unintended microarchitectural states. They can make these effects visible to software adversaries, enabling them to exploit these hardware vulnerabilities remotely. The affected targets range from low-end embedded devices to complex servers, that are hardened with advanced defenses, such as data-execution prevention, supervisor-mode execution prevention, and control-flow integrity.

Hardware vulnerabilities. Cross-layer attacks circumvent many existing security mechanisms [20, 23, 43, 45, 64, 69, 72, 74], that focus on mitigating attacks exploiting software vulnerabilities. Moreover, hardware-security extensions are not designed to tackle hardware vulnerabilities. Their implementation remains vulnerable to potentially undetected hardware bugs committed at design-time. In fact, deployed extensions such as SGX [31] and TrustZone [3] have been targets of successful cross-layer attacks [69, 72]. Research projects such as Sanctum [18], Sanctuary [8], or Keystone [39] are also not designed to ensure security at the hardware implementation level. Hardware vulnerabilities can occur due to: (a) incorrect or ambiguous security specifications, (b) incorrect design, (c) flawed implementation of the design, or (d) a combination thereof. Hardware implementation bugs are introduced either through human error or faulty translation of the design in gate-level synthesis.

SoC designs are typically implemented at register-transfer level (RTL) by engineers using hardware description languages (HDLs), such as Verilog and VHDL, which are *synthesized* into a lower-level representation using automated tools. Just like software programmers introduce bugs to the high-level code, hardware engineers may accidentally introduce bugs to the RTL code. While software errors typically cause a crash which triggers various fallback routines to ensure the safety and security of other programs running on the platform, no such safety net exists for hardware bugs. Thus, even mi-

nor glitches in the implementation of a module within the processor can compromise the SoC security objectives and result in persistent/permanent denial of service, IP leakage, or exposure of assets to untrusted entities.

Detecting hardware security bugs. The semiconductor industry makes extensive use of a variety of techniques, such as simulation, emulation, and formal verification to detect such bugs. Examples of industry-standard tools include Incisive [10], Solidify [5], Questa Simulation and Questa Formal [44], OneSpin 360 [66], and JasperGold [11]. These were originally designed for *functional verification* with security-specific verification incorporated into them later.

While a rich body of knowledge exists within the software community (e.g., regarding software exploitation and techniques to automatically detect software vulnerabilities [38, 46]), security-focused HDL analysis is currently lagging behind [35, 57]. Hence, the industry has recently adopted a *security development lifecycle* (SDL) for hardware [68] — inspired by software practices [26]. This process combines different techniques and tools, such as RTL manual code audits, assertion-based testing, dynamic simulation, and automated security verification. However, the recent outbreak of *cross-layer attacks* [20, 23, 37, 43, 45, 47, 48, 49, 51, 52, 53, 64, 69, 74] poses a spectrum of difficult challenges for these security verification techniques, because they exploit complex and subtle inter-dependencies between hardware and software. Existing verification techniques are fundamentally limited in modeling and verifying these interactions. Moreover, they also do not scale with the size and complexity of real-world SoC designs.

Goals and Contributions. In this paper, we show that current hardware security verification techniques are fundamentally limited. We provide a wide range of results using a comprehensive test harness, encompassing different types of hardware vulnerabilities commonly found in real-world platforms. To that end, we conducted two case studies to systematically and qualitatively assess existing verification techniques with respect to detecting RTL bugs. Together with our industry partners, we compiled a list of 31 RTL bugs based on public Common Vulnerabilities and Exposures (CVEs) [37, 43, 50, 54, 55] and real-world errata [25]. We injected bugs into two open-source RISC-V-based SoC designs, which we will open-source after publication.

We organized an international public hardware security competition, Hack@DAC, where 54 teams of researchers competed for three months to find these bugs. While a number of bugs could not be detected by any of the teams, several participants also reported *new* vulnerabilities of which we had no prior knowledge. The teams used manual RTL inspection and simulation techniques to detect the bugs. In industry, these are usually complemented by automated tool-based and formal verification approaches. Thus, our second case study focused on two state-of-the-art formal verification tools: the first deploys formal verification to perform exhaustive and complete verification of a hardware design, while the second leverages formal verification and path sensitization to check for illegal data flows and fault tolerance.

Our second case study revealed that certain properties of RTL bugs pose challenges for state-of-the-art verification techniques with respect to black-box abstraction, timing flow, and non-register states. This causes security bugs in the RTL of real-world SoCs to slip through the verification process. Our results from the two case studies indicate that particular classes of hardware bugs entirely evade detection—even when complementing systematic tool-based verification approaches with manual inspection. RTL bugs arising from complex and cross-modular interactions in SoCs render these bugs extremely difficult to detect in practice. Furthermore, such bugs are exploitable from software, and thus can compromise the entire platform. We call such bugs **HardFails**.

To the best of our knowledge, this is the first work to provide a systematic and in-depth analysis of state-of-the-art hardware verification approaches for security-relevant RTL bugs. Our findings shed light on the capacity of these tools and demonstrate reproducibly how bugs can slip through current hardware security verification processes. Being also software-exploitable, these bugs pose an immense security threat to SoCs. Through our work, we highlight why further research is required to improve state-of-the-art security verification of hardware. To summarize, our main contributions are:

- **Systematic evaluation and case studies:** We compile a comprehensive test harness of real-world RTL bugs, on which we base our two case studies: (1) Hack@DAC'18, in which 54 independent teams of researchers competed worldwide over three months to find these bugs using manual RTL inspection and simulation techniques, and (2) an investigation of the bugs using industry-leading formal verification tools that are representative of the current state of the art. Our results show that particular classes of bugs entirely evade detection, despite combining both tool-based security verification approaches and manual analysis.
- **Stealthy hardware bugs:** We identify *HardFails* as RTL bugs that are distinctly challenging to detect using industry-standard security verification techniques. We explain the fundamental limitations of these techniques in detail using concrete examples.
- **Open-sourcing:** We will open-source our bugs testbed at publication to the community.

2 SoC Verification Processes and Pitfalls

Similar to the Security Development Lifecycle (SDL) deployed by software companies [26], semiconductor companies [15, 35, 40] have recently adapted SDL for hardware design [57]. We describe next the conventional SDL process for hardware and the challenges thereof.

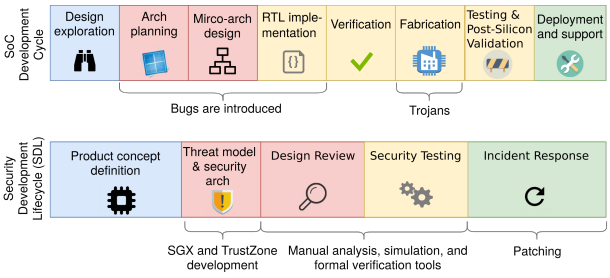

FIGURE 1: Typical Security Development Lifecycle (SDL) process followed by semiconductor companies.

2.1 The Security Development Lifecycle (SDL) for Hardware

SDL is conducted concurrently with the conventional hardware development lifecycle [68], as shown in Figure 1. The top half of Figure 1 shows the hardware development lifecycle. It begins with design exploration followed by defining the specifications of the product architecture. After the architecture specification, the microarchitecture is designed and implemented in RTL. Concurrently, pre-silicon verification efforts are conducted until tape-out to detect and fix all functional bugs that do not meet the *functional specification*. After tape-out and fabrication, iterations of post-silicon validation, functional testing, and tape-out "spins" begin. This cycle is repeated until no defects are found and all quality requirements are met. Only then does the chip enter mass production and is shipped out. Any issues found later in-field are debugged, and the chip is then either patched if possible or recalled.

After architectural features are finalized, a security assessment is performed, shown in the bottom half of Figure 1. The adversary model and the security objectives are compiled in the *security specification*. This typically entails a list of assets, entry points to access these assets, and the adversary capabilities and architectural security objectives to mitigate these threats. These are translated into microarchitectural security specifications, including security test cases (both positive and negative). After implementation, pre-silicon security verification is conducted using dynamic verification (i.e., simulation and emulation), formal verification, and manual RTL reviews. The chip is not signed off for tape-out until all security specifications are met. After tape-out and fabrication, post-silicon security verification commences. The identified security bugs in both pre-silicon and post-silicon phases are rated for severity using the industry-standard scoring systems such as the Common Vulnerability Scoring System (CVSS) [30] and promptly fixed. Incident response teams handle issues in shipped products and provide patches, if possible.

2.2 Challenges with SDL

Despite multiple tools and security validation techniques used by industry to conduct SDL, it remains a highly challenging, tedious, and complex process even for industry experts. Existing techniques largely rely on human expertise to define the security test cases and run the tests. The correct *security specifications* must be exhaustively anticipated, identified, and accurately and adequately expressed using security properties that can be captured and verified by the tools. We discuss these challenges further in Section 7.

Besides the specifications, the techniques and tools themselves are not scalable and are less effective in capturing subtle semantics that are relevant to many vulnerabilities, which is the focus of this work. We elaborate next on the limitations of state-of-the-art hardware security verification tools commonly used by industry. To investigate the capabilities of these tools, we then construct a comprehensive test-harness of real-world RTL vulnerabilities.

3 Assessing Hardware Security Verification

In this section, we focus on why the verification of the security properties of modern hardware is challenging and provide requirements for assessing existing verification techniques under realistic conditions. First, we describe how these verification techniques fall short. Second, we provide a list of common and realistic classes of hardware bugs, which we use to construct a test harness for assessing the effectiveness of these verification techniques. Third, we discuss how these bugs relate to the common security goals of a chip.

3.1 Limitations of Automated Verification

Modern hardware designs are highly complex and incorporate hundreds of in-house and third-party Intellectual Property (IP) components. This creates room for vulnerabilities to be introduced in the inter-modular interactions of the design hierarchy. Multi-core architectures typically have an intricate interconnect fabric between individual cores (utilizing complex communication protocols), multi-level cache controllers with shared un-core and private on-core caches, memory and interrupt controllers, and debug and I/O interfaces.

For each core, these components contain logical modules such as fetch and decode stages, an instruction scheduler, individual execution units, branch prediction, instruction and data caches, the memory subsystem, re-order buffers, and queues. These are implemented and connected using individual RTL modules. The average size of each module is several hundred lines of code (LOC). Thus, real-world SoCs can easily approach 100,000 lines of RTL code, and some designs may even have millions of LOC. Automatically verifying, at the RTL level, the respective interconnections and checking them against security specifications raises a number of fundamental challenges for the state-of-the-art approaches. These are described below.

L-1: Cross-modular effects. Hardware modules are interconnected in a highly hierarchical design with multiple inter-

dependencies. Thus, an RTL bug located in an individual module may trigger a vulnerability in intra- and inter-modular information flows spanning multiple complex modules. Pinpointing the bug requires analyzing these flows across the relevant modules, which is highly cumbersome and unreliable to achieve by manual inspection. It also pushes formal verification techniques to their limits, which work by modeling and analyzing all the RTL modules of the design to verify whether design specifications (expressed using security property assertions, invariants and disallowed information flows) and implementation match.

Detecting such vulnerabilities requires loading the RTL code of all the relevant modules into the tools to model and analyze the entire state space, thus driving them quickly into *state explosion* due to the underlying modeling algorithms [16, 21]. Alleviating this by providing additional computational resources and time is not scalable as the complexity of SoCs continues to increase. Selective "black-box" abstraction of some of the modules, state space constraining, and bounded-model checking are often used. However, they do not eliminate the fundamental problem and rely on interactive human expertise. Erroneously applying them may introduce false negatives, leading to missed vulnerabilities.

L-2: Timing-flow gap. Current industry-standard techniques are limited in capturing and verifying security properties related to timing flow (in terms of clock cycle latency). This leads to vast sources of information leakage due to software-exploitable timing channels (Section 8). A timing flow exists between the circuit's input and output when the number of clock cycles required for the generation of the output depends on input values or the current memory/register state. This can be exploited to leak sensitive information when the timing variation is discernible by an adversary and can be used to infer inputs or memory states. This is especially problematic for information flows and resource sharing across different privilege levels. This timing variation should remain indistinguishable in the RTL, or should not be measurable from the software. However, current industry-standard security verification techniques focus exclusively on the functional information flow of the logic and fail to model the associated timing flow. The complexity of timing-related security issues is aggravated when the timing flow along a logic path spans multiple modules and involves various inter-dependencies.

L-3: Cache-state gap. State-of-the-art verification techniques only model and analyze the *architectural state* of a processor by exclusively focusing on the state of registers. However, they do not support analysis of non-register states, such as caches, thus completely discarding modern processors' highly complex microarchitecture and diverse hierarchy of caches. This can lead to severe security vulnerabilities arising due to state changes that are unaccounted for, e.g., the changing state of shared cache resources across multiple privilege levels. Caches represent a state that is influenced directly or indirectly by many control-path signals and can generate security vulnerabilities in their interactions, such as illegal information leakages across different privilege levels. Identifying RTL bugs that trigger such vulnerabilities is beyond the capabilities of existing techniques.

L-4: Hardware-software interactions. Some RTL bugs remain indiscernible to hardware security verification techniques because they are not explicitly vulnerable unless triggered by the software. For instance, although many SoC access control policies are directly implemented in hardware, some are programmable by the overlying firmware to allow for post-silicon flexibility. Hence, reasoning on whether an RTL bug exists is inconclusive when considering the hardware RTL in isolation. These vulnerabilities would only materialize when the hardware-software interactions are considered, and existing techniques do not handle such interactions.

3.2 Constructing Real-World RTL Bugs

To systematically assess the state of the art in hardware security verification with respect to the limitations described above, we construct a test harness by implementing a large number of RTL bugs in RISC-V SoC designs (cf. Table 1). To the best of our knowledge, we are the first to compile and showcase such a collection of hardware bugs. Together with our co-authors at Intel, we base our selection and construction of bugs on a solid representative spectrum of real-world CVEs [47, 48, 49, 51, 52, 53] as shown in Table 1. For instance, bug #22 was inspired by a recent security vulnerability in the Boot ROM of video gaming mobile processors [56], which allowed an attacker to bring the device into BootROM Recovery Mode (RCM) via USB access. This buffer overflow vulnerability affected many millions of devices and is popularly used to hack a popular video gaming console[1].

We extensively researched CVEs that are based on software-exploitable hardware and firmware bugs and classified them into different categories depending on the weaknesses they represent and the modules they impact. We reproduced them by constructing representative bugs in the RTL and demonstrated their software exploitability and severity by crafting a real-world software exploit based on one of these bugs in Appendix D. Other bugs were constructed with our collaborating hardware security professionals, inspired by bugs that they have previously encountered and patched during the pre-silicon phase, which thus never escalated into CVEs. The chosen bugs were implemented to achieve coverage of different security-relevant modules of the SoC.

Since industry-standard processors are based on proprietary RTL implementations, we mimic the CVEs by reproducing and injecting them into the RTL of widely-used RISC-V SoCs. We also investigate more complex microarchitecture features of another RISC-V SoC and discover vulnerabilities already existing in its RTL (Section 4). These RTL bugs manifest as:

[1] https://github.com/Cease-and-DeSwitch/fusee-launcher

- **Incorrect assignment bugs** due to variables, registers, and parameters being assigned incorrect literal values, incorrectly connected or left floating unintended.
- **Timing bugs** resulting from timing flow issues and incorrect behavior relevant to clock signaling such as information leakage.
- **Incorrect case statement bugs** in the finite state machine (FSM) models such as incorrect or incomplete selection criteria, or incorrect behavior within a case.
- **Incorrect if-else conditional bugs** due to incorrect boolean conditions or incorrect behavior described within either branch.
- **Specification bugs** due to a mismatch between a specified property and its actual implementation or poorly specified / under-specified behavior.

These seemingly minor RTL coding errors may constitute security vulnerabilities, some of which are very difficult to detect during verification. This is because of their interconnection and interaction with the surrounding logic that affects the complexity of the subtle side effects they generate in their manifestation. Some of these RTL bugs may be patched by modifying parts of the software stack that use the hardware (e.g., using firmware/microcode updates) to circumvent them and mitigate specific exploits. However, since RTL is usually compiled into hardwired integrated circuitry logic, the underlying bugs cannot, in principle, be patched after production.

The limited capabilities of current detection approaches in modeling hardware designs and formulating and capturing relevant security assertions raise challenges for detecting some of these vulnerabilities, which we investigate in depth in this work. We describe next the adversary model we assume for our vulnerabilities and our investigation.

3.3 Adversary Model

In our work, we investigate microarchitectural details at the RTL level. However, all hardware vendors keep their proprietary industry designs and implementations closed. Hence, we use an open-source SoC based on the popular open-source RISC-V [73] architecture as our platform. RISC-V supports a wide range of possible configurations with many standard features that are also available in modern processor designs, such as privilege level separation, virtual memory, and multithreading, as well as optimization features such as configurable branch prediction and out-of-order execution.

RISC-V RTL is freely available and open to inspection and modification. While this is not necessarily the case for industry-leading chip designs, an adversary might be able to reverse engineer or disclose/steal parts of the chip using existing tools[2][3]. Hence, we consider a strong adversary that can also inspect the RTL code.

In particular, we make the following assumptions:

- **Hardware Vulnerability:** The attacker has knowledge of a vulnerability in the hardware design of the SoC (i.e., at the RTL level) and can trigger the bug from software.
- **User Access:** The attacker has complete control over a user-space process, and thus can issue unprivileged instructions and system calls in the basic RISC-V architecture.
- **Secure Software:** Software vulnerabilities and resulting attacks, such as code-reuse [65] and data-only attacks [27] against the software stack, are orthogonal to the problem of cross-layer bugs. Thus, we assume all platform software is protected by defenses such as control-flow integrity [1] and data-flow integrity [13], or is formally verified.

The goal of an adversary is to leverage the vulnerability on the chip to provoke unintended functionality, e.g., access to protected memory locations, code execution with elevated privileges, breaking the isolation of other processes running on the platform, or permanently denying services. RTL bugs in certain hardware modules might only be exploitable with physical access to the victim device, for instance, bugs in debug interfaces. However, other bugs are software-exploitable, and thus have a higher impact in practice. Hence, we focus on software-exploitable RTL vulnerabilities, such as the exploit showcased in Appendix D. Persistent denial of service (PDoS) attacks that require exclusive physical access are out of scope. JTAG attacks, though they require physical access, are still in scope as the end user may be the attacker and might attempt to unlock the device to steal manufacturer secrets. Furthermore, exploiting the JTAG interface often requires a combination of both physical access and privilege escalation by means of a software exploit to enable the JTAG interface. We also note that an adversary with unprivileged access is a realistic model for real-world SoCs: Many platforms provide services to other devices over the local network or even over the internet. Thus, the attacker can obtain some limited software access to the platform already, e.g., through a webserver or an RPC interface. Furthermore, we emphasize that this work focuses only on tools and techniques used to detect bugs before tape-out.

4 HardFails: Hardware Security Bugs

In light of the limitations of state-of-the-art verification tools (Section 3.1), we constructed a testbed of real-world RTL bugs (Section 3.2) and conducted two extensive case studies on their detection (described next in Sections 5 and 6). Based on our findings, we have identified particular classes of hardware bugs that exhibit properties that render them more challenging to detect with state-of-the-art techniques. We call these HardFails. We now describe different types of these HardFails encountered during our analysis of two RISC-V SoCs, Ariane [59] and PULPissimo [61]. In Section 5.3, we describe the actual bugs we instantiated for our case studies.

Ariane is a 6-stage in-order RISC-V CPU that implements the RISC-V draft privilege specification and can run Linux OS. It has a memory management unit (MMU) consisting of

[2] https://www.chipworks.com/
[3] http://www.degate.org/

TABLE 1: Detection results for bugs in PULPissimo SoC based on formal verification (**SPV** and **FPV**, i.e., JasperGold Security Path Verification and Formal Property Verification) and our hardware security competition (**M&S**, i.e., manual inspection and simulation). Check and cross marks indicate detected and undetected bugs, respectively. Bugs marked **inserted** were injected by our team and based on the listed CVEs, while bugs marked **native** were already present in the SoC and discovered by the participants during Hack@DAC. **LOC** denotes the number of lines of code, and **states** denotes the total number of logic states for the modules needed to attempt to detect this bug.

#	Bug	Type	SPV	FPV	M&S	Modules	LOC	# States
1	Address range overlap between peripherals SPI Master and SoC	Inserted (CVE-2018-12206 / CVE-2019-6260 / CVE-2018-8933)	✓	✓	✓	91	6685	1.5×10^{20}
2	Addresses for L2 memory is out of the specified range.	Native	✓	✓	✓	43	6746	3.5×10^{13}
3	Processor assigns privilege level of execution incorrectly from CSR.	Native	✗	✓	✓	2	1186	2.1×10^{96}
4	Register that controls GPIO lock can be written to with software.	Inserted (CVE-2017-18293)	✓	✓	✗	2	1186	2.1×10^{96}
5	Reset clears the GPIO lock control register.	Inserted (CVE-2017-18293)	✓	✓	✗	2	408	1
6	Incorrect address range for APB allows memory aliasing.	Inserted (CVE-2018-12206 / CVE-2019-6260)	✓	✓	✗	1	110	2
7	AXI address decoder ignores errors.	Inserted (CVE-2018-4850)	✗	✓	✗	1	227	2
8	Address range overlap between GPIO, SPI, and SoC control peripherals.	Inserted (CVE-2018-12206 / (CVE-2017-5704)	✓	✓	✓	68	14635	9.4×10^{21}
9	Incorrect password checking logic in debug unit.	Inserted (CVE-2018-8870)	✗	✓	✗	4	436	1
10	Advanced debug unit only checks 31 of the 32 bits of the password.	Inserted (CVE-2017-18347 / CVE-2017-7564)	✗	✓	✗	4	436	16
11	Able to access debug register when in halt mode.	Native (CVE-2017-18347 /	✗	✓	✓	2	887	1
12	Password check for the debug unit does not reset after successful check.	Inserted (CVE-2017-7564)	✗	✓	✓	4	436	16
13	Faulty decoder state machine logic in RISC-V core results in a hang.	Native	✗	✓	✓	2	1119	32
14	Incomplete case statement in ALU can cause unpredictable behavior.	Native	✗	✓	✓	2	1152	4
15	Faulty logic in the RTC causing inaccurate time calculation for security-critical flows, e.g., DRM.	Native	✗	✓	✗	1	191	1
16	Reset for the advanced debug unit not operational.	Inserted (CVE-2017-18347)	✗	✗	✓	4	436	16
17	Memory-mapped register file allows code injection.	Native	✗	✗	✓	1	134	1
18	Non-functioning cryptography module causes DOS.	Inserted	✗	✗	✗	24	2651	1
19	Insecure hash function in the cryptography module.	Inserted (CVE-2018-1751)	✗	✗	✗	24	2651	N/A
20	Cryptographic key for AES stored in unprotected memory.	Inserted (CVE-2018-8933 / CVE-2014-0881 / CVE-2017-5704)	✗	✗	✗	57	8955	1
21	Temperature sensor is muxed with the cryptography modules.	Inserted	✗	✗	✓	1	65	1
22	ROM size is too small preventing execution of security code.	Inserted (CVE-2018-6242 /) CVE-2018-15383)	✗	✗	✓	1	751	N/A
23	Disabled the ability to activate the security-enhanced core.	Inserted (CVE-2018-12206)	✗	✗	✗	1	282	N/A
24	GPIO enable always high.	Inserted (CVE-2018-1959)	✗	✗	✗	1	392	1
25	Unprivileged user-space code can write to the privileged CSR.	Inserted (CVE-2018-7522 / CVE-2017-0352)	✗	✗	✓	1	745	1
26	Advanced debug unit password is hard-coded and set on reset.	Inserted (CVE-2018-8870)	✗	✗	✓	1	406	16
27	Secure mode is not required to write to interrupt registers.	Inserted (CVE-2017-0352)	✗	✗	✓	1	303	1
28	JTAG interface is not password protected.	Native	✗	✗	✓	1	441	1
29	Output of MAC is not erased on reset.	Inserted	✗	✗	✓	1	65	1
30	Supervisor mode signal of a core is floating preventing the use of SMAP.	Native	✗	✗	✓	1	282	1
31	GPIO is able to read/write to instruction and data cache.	Native	✗	✗	✓	1	151	4

data and instruction translation lookaside buffers (TLBs), a hardware page table walker, and a branch prediction unit to enable speculative execution. Figure 4 in Appendix A shows its high-level microarchitecture.

PULPissimo is an SoC based on a simpler RISC-V core with both instruction and data RAM as shown in Figure 2. It provides an Advanced Extensible Interface (AXI) for accessing memory from the core. Peripherals are directly connected to an Advanced Peripheral Bus (APB) which connects them to the AXI through a bridge module. It provides support for autonomous I/O, external interrupt controllers and features a debug unit and an SPI slave.

TLB Page Fault Timing Side Channel (L-1 & L-2).
While analyzing the Ariane RTL, we noted a timing side-channel leakage with TLB accesses. TLB page faults due to illegal accesses occur in a different number of clock cycles than page faults that occur due to unmapped memory (we contacted the developers and they acknowledged the vulnerability). This timing disparity in the RTL manifests in the microarchitectural behavior of the processor. Thus, it constitutes a software-visible side channel due to the measurable clock-cycle difference in the two cases. Previous work already demonstrated how this can be exploited by user-space adversaries to probe mapped and unmapped pages and to break randomization-based defenses [24, 29]. Timing flow properties cannot be directly expressed by simple properties or modeled by state-of-the-art verification techniques. Moreover, for this vulnerability, we identify at least seven RTL modules that would need to be modeled, analyzed and verified in combination, namely: *mmu.sv - nbdcache.sv - tlb.sv instantiations - ptw.sv - load_unit.sv - store_unit.sv*. Besides modeling their complex inter- and intra-modular logic flows (L-1), the timing flows need to be modeled to formally prove the absence of this timing channel leakage, which is *not supported* by current industry-standard tools (L-2). Hence, the only alternative is to verify this property by manually inspecting and following the clock cycle transitions across the RTL modules, which is highly cumbersome and error-prone. However, the design must still be analyzed to verify that timing side-channel resilience is implemented correctly and bug-free in the RTL. This only becomes far more complex for real-world industry-standard SoCs. We show the RTL hierarchy of the Ariane core in Figure 5 in Appendix A to illustrate its complexity.

Pre-Fetched Cache State Not Rolled Back (L-1 & L-3).
Another issue in Ariane is with the cache state when a system return instruction is executed, where the privilege level of the core is not changed until this instruction is retired. Before retirement, linear fetching (guided by branch prediction) of data and instructions following the unretired system return instruction continues at the current higher system privilege level. Once the instruction is retired, the execution mode of the core is changed to the unprivileged level, but the entries that

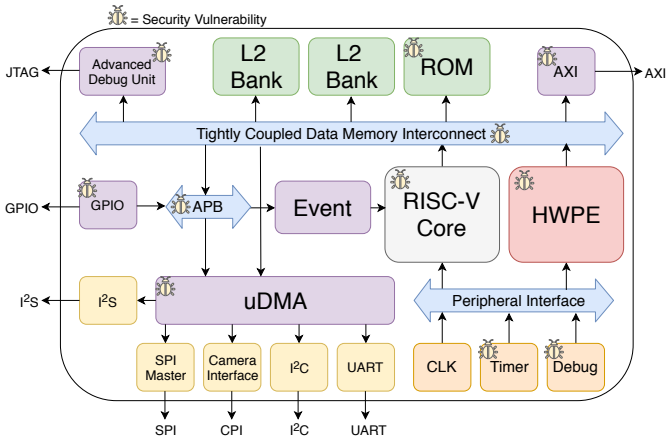

FIGURE 2: Hardware overview of the PULPissimo SoC. Each bug icon indicates the presence of at least one security vulnerability in the module.

were pre-fetched into the cache (at the system privilege level) do not get flushed. These shared cache entries are visible to user-space software, thus enabling timing channels between privileged and unprivileged software.

Verifying the implementation of all the flush control signals and their behavior in all different states of the processor requires examining at least eight modules: *ariane.sv - controller.sv - frontend.sv - id_stage.sv - icache.sv - fetch_fifo - ariane_pkg.sv - csr_regfile.sv* (see Figure 5). This is complex because it requires identifying and defining all the relevant security properties to be checked across these RTL modules. Since current industry-standard approaches do not support expressive capturing and the verification of cache states, this issue in the RTL can only be found by manual inspection.

Firmware-Configured Memory Ranges (L-4).
In PULPissimo, we added peripherals with injected bugs to reproduce bugs from CVEs. We added an AES encryption/decryption engine whose input key is stored and fetched from memory tightly coupled to the processor. The memory address the key is stored in is unknown, and whether it is within the protected memory range or not is inconclusive by observing the RTL alone. In real-world SoCs, the AES key is stored in programmable fuses. During secure boot, the bootloader/firmware senses the fuses and stores the key to memory-mapped registers. The access control filter is then configured to allow only the AES engine access to these registers, thus protecting this memory range. Because the open-source SoC we used did not contain a fuse infrastructure, the key storage was mimicked to be in a register in the Memory-Mapped I/O (MMIO) space.

Although the information flow of the AES key is defined in hardware, its location is controlled by the firmware. Reasoning on whether the information flow is allowed or not using conventional hardware verification approaches is inconclusive when considering the RTL code in isolation.

The vulnerable hardware/firmware interactions cannot be identified unless they are co-verified. Unfortunately, current industry-standard tools do not support this.

Memory Address Range Overlap (L-1 & L-4). PULPissimo provides I/O support to its peripherals by mapping them to different memory address ranges. If an address range overlap bug is committed at design-time or by firmware, this can break access control policies and have critical security consequences, e.g., privilege escalation. We injected an RTL bug where there is address range overlap between the SPI Master Peripheral and the SoC Control Peripheral. This allowed the untrusted SPI Master to access the SoC Control memory address range over the APB bus.

Verifying issues at the SoC interconnect in such complex bus protocols is challenging since too many modules needed to support the interconnect have to be modeled to properly verify their security. This increases the scope and the complexity of potential bugs far beyond just a few modules, as shown in Table 1. Such an effect causes an explosion of the state space since all the possible states have to be modeled accurately to remain sound. Proof kits for accelerated verification of advanced SoC interconnect protocols were introduced to mitigate this for a small number of bus protocols (AMBA3 and AMBA4). However, this requires an add-on to the default software and many protocols are not supported[4].

5 Crowdsourcing Detection

We organized and conducted a capture-the-flag competition, Hack@DAC, in which 54 teams (7 from leading industry vendors and 47 from academia) participated. The objective for the teams was to detect as many RTL bugs as possible from those we injected deliberately in real-world open-source SoC designs (see Table 1). This is designed to mimic real-world bug bounty programs from semiconductor companies [17, 32, 62, 63]. The teams were free to use any techniques: simulation, manual inspection, or formal verification.

5.1 Competition Preparation

RTL of open-source RISC-V SoCs was used as the testbed for Hack@DAC and our investigation. Although these SoCs are less complex than high-end industry proprietary designs, this allows us to feasibly inject (and detect) bugs into less complex RTL. Thus, this represents the best-case results for the verification techniques used during Hack@DAC and our investigation. Moreover, it allows us to open-source and showcase our testbed and bugs to the community.Hack@DAC consisted of two phases: a preliminary Phase 1 and final Phase 2, which featured the RISC-V Pulpino and PULPissimo SoCs, respectively. Phase 1 was conducted remotely over a two-month period. Phase 2 was conducted in an 8-hour time frame co-located with DAC (Design Automation Conference).

For Phase 1, we chose the Pulpino [60] SoC since it was a real-world, yet not an overly complex SoC design for the teams to work with. It features a RISC-V core with instruction and data RAM, an AXI interconnect for accessing memory, with peripherals on an APB accessing the AXI through a bridge module. It also features a boot ROM, a debug unit and a serial peripheral interface (SPI) slave. We inserted security bugs in multiples modules of the SoC, including the AXI, APB, debug unit, GPIO, and bridge.

For Phase 2, we chose the more complex PULPissimo [61] SoC, shown in Figure 2. It additionally supports hardware processing engines, DMA, and more peripherals. This allowed us to extend the SoC with additional security features, making room for additional bugs. Some native security bugs were discovered by the teams and were reported to the SoC designers.

5.2 Competition Objectives

For Hack@DAC, we first implemented additional security features in the SoC, then defined the security objectives and adversary model and accordingly inserted the bugs. Specifying the security goals and the adversary model allows teams to define what constitutes a security bug. Teams had to provide a bug description, location of RTL file, code reference, the security impact, adversary profile, and the proposed mitigation.
Security Features: We added password-based locks on the JTAG modules of both SoCs and access control on certain peripherals. For the Phase-2 SoC, we also added a cryptographic unit implementing multiple cryptographic algorithms. We injected bugs into these features and native features to generate security threats as a result.
Security Goals: We provided the three main security goals for the target SoCs to the teams. Firstly, unprivileged code should not escalate beyond its privilege level. Secondly, the JTAG module should be protected against an adversary with physical access. Finally, the SoCs should thwart software adversaries from launching denial-of-service attacks.

5.3 Overview of Competition Bugs

As described earlier in Section 3.2, the bugs were selected and injected together with our Intel collaborators. They are inspired by their hardware security expertise and real-world CVEs (cf. Table 1) and aim to achieve coverage of different security-relevant components of the SoC. Several participants also reported a number of *native* bugs already present in the SoC that we did not deliberately inject. We describe below some of the most interesting bugs.
UDMA address range overlap: We modified the memory address range of the UDMA so that it overlaps with the master port to the SPI. This bug allows an adversary with access to

[4] http://www.marketwired.com/press-release/jasper-introduces-intelligent-proof-kits-faster-more-accurate-verification-soc-interface-1368721.htm

the UMDA memory to escalate its privileges and modify the SPI memory. This bug is an example of the "Memory Address Range Overlap" HardFail type in Section 4. Other address range configuration bugs (#1, 2, 6 and 8) were also injected in the APB bus for different peripherals.

GPIO errors: The address range of the GPIO memory was erroneously declared. An adversary with GPIO access can escalate its privilege and access the SPI Master and SoC Control. The GPIO enable was rigged to display a fixed erroneous status of '1', which did not give the user a correct display of the actual GPIO status. The GPIO lock control register was made write-accessible by user-space code, and it was flawed to clear at reset. Bugs #4, 5, 24 and 31 are such examples.

Debug/JTAG errors: The password-checking logic in the debug unit was flawed and its state was not being correctly reset after a successful check. We hard-coded the debug unit password, and the JTAG interface was not password protected. Bugs #9, 10, 11, 16, 26, and 28 are such examples.

Untrusted boot ROM: A native bug (bug #22) would allow unprivileged compromise of the boot ROM and potentially the execution of untrusted boot code at a privileged level, thus disclosing sensitive information.

Erroneous AXI finite-state machine: We injected a bug (bug #7) in the AXI address decoder such that, if an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This bug can be exploited to cause computational faults in the execution of security-critical code (we showcase how to exploit this vulnerability—which was not detected by all teams—in Appendix D).

Cryptographic unit bugs: We injected bugs in a cryptographic unit that we inserted to trigger denial-of-service, a broken cryptographic implementation, insecure key storage, and disallowed information leakage. Bugs #18, 19, 20, 21, and 29 are such examples.

5.4 Competition Results

Various insights were drawn from the submitted bug reports and results, which are summarized in Table 1.

Analyzing the bug reports: Bug reports submitted by teams revealed which bug types were harder to detect and analyze using existing approaches. We evaluated the submissions and rated them for accuracy and detail, e.g., bug validity, methodology used, and security impact.

Detected bugs: Most teams easily detected two bugs in PULPissimo. The first one is where debug IPs were used when not intended. The second bug was where we declared a local parameter PULP_SEC, which was always set to '1', instead of the intended PULP_SECURE. The former was detected because debugging interfaces represent security-critical regions of the chip. The latter was detected because it indicated intuitively that exploiting this parameter would lead to privilege escalation attacks. The teams reported that they prioritized inspecting security-relevant modules of the SoC, such as the debug interfaces.

Undetected bugs: Many inserted bugs were not detected. One was in the advanced debug unit, where the password bit index register has an overflow (bug #9). This is an example of a security flaw that would be hard to detect by methods other than verification. Moreover, the presence of many bugs within the advanced debug unit password checker further masked this bug. Another bug was the cryptographic unit key storage in unprotected memory (bug #20). The teams could not detect it as they focused on the RTL code in isolation and did not consider HW/FW interactions.

Techniques used by the teams: The teams were free to use any techniques to detect the bugs but most teams eventually relied on manual inspection and simulation.

- **Formal verification:** One team used an open-source formal verification tool (VeriCoq), but they reported little success because these tools (i) did not scale well with the complete SoC and (ii) required expertise to use and define the security properties. Some teams deployed their in-house verification techniques, albeit with little success. They eventually resorted to manual analysis.
- **Assertion-based simulation:** Some teams prepared RTL testbenches and conducted property-based simulations using SystemVerilog assertion statements.
- **Manual inspection:** All teams relied on manual inspection methods since they are the easiest and most accessible and require less expertise than formal verification, especially when working under time constraints. A couple of teams reported prioritizing the inspection of security-critical modules such as debug interfaces.
- **Software-based testing:** One team detected software-exposure and privilege escalation bugs by running C code on the processor and attempting to make arbitrary reads/writes to privileged memory locations. In doing this, they could detect bugs #4, #8, #15, and #17.

Limitations of manual analysis: While manual inspection can detect the widest array of bugs, our analysis of the Hack@DAC results reveals its limitations. Manual analysis is qualitative and difficult to scale to cross-layer and more complex bugs. In Table 1, out of 16 cross-module bugs (spanning more than one module) only 9 were identified using manual inspection. Three of them (#18, #19, and #20) were also undetected by formal verification methods, which is 10% of the bugs in our case studies.

6 Detection Using State-of-The-Art Tools

Our study reveals two results: (1) a number of bugs could not be detected by means of manual auditing and other ad-hoc methods, and (2) the teams were able to find bugs already existing in the SoC which we did not inject and were not

aware of. This prompted us to conduct a second in-house case study to further investigate whether formal verification techniques can be used to detect these bugs. In practice, hardware-security verification engineers use a combination of techniques such as formal verification, simulation, emulation, and manual inspection. Our first case study covered manual inspection, simulation and emulation techniques. Thus, we focused our second case study on assessing the effectiveness of industry-standard formal verification techniques usually used for verifying pre-silicon hardware security.

In real-world security testing (see Section 2), engineers will not have prior knowledge of the specific vulnerabilities they are trying to find. Our goal, however, is to investigate how an industry-standard tool can detect RTL bugs that we deliberately inject in an open-source SoC and have prior knowledge of (see Table 1). Since there is no regulation or explicitly defined standard for hardware-security verification, we focus our investigation on the most popular and de-facto standard formal verification platform used in industry [11]. This platform encompasses a representative suite of different state-of-the-art formal verification techniques for hardware security assurance. As opposed to simulation and emulation techniques, formal verification guarantees to model the state space of the design and formally prove the desired properties. We emphasize that we deliberately fix all other variables involved in the security testing process, in order to focus in a controlled setting on testing the capacity and limitations of the techniques and tools themselves. Thus, our results reflect the effectiveness of tools in a best case where the bug is known a priori. This eliminates the possibility of writing an incorrect security property assertion which fails to detect the bug.

6.1 Detection Methodology

We examined each of the injected bugs and its nature in order to determine which formal technique would be best suited to detect it. We used two formal techniques: Formal Property Verification (FPV) and JasperGold's Security Path Verification (SPV) [12]. They represent the state of the art in hardware security verification and are used widely by the semiconductor industry [4], including Intel.

FPV checks whether a set of security properties, usually specified as SystemVerilog Assertions (SVA), hold true for the given RTL. To describe the assertions correctly, we examined the location of each bug in the RTL and how its behavior is manifested with the surrounding logic and input/output relationships. Once we specified the security properties using *assert*, *assume* and *cover* statements, we determined which RTL modules we need to model to prove these assertions. If a security property is violated, the tool generates a counterexample; this is examined to ensure whether the intended security property is indeed violated or is a false alarm.

SPV detects bugs which specifically involve unauthorized information flow. Such properties cannot be directly captured using SVA/PSL assertions. SPV uses path sensitization techniques to exhaustively and formally check if unauthorized data propagates (through a functional path) from a source to a destination signal. To specify the SPV properties, we identified source signals where the sensitive information was located and destination signals where it should *not* propagate. We then identified the bounding preconditions to constrain the paths the tool searches to alleviate state and time explosion. Similar to FPV, we identified the modules that are required to capture the information flow of interest. This must include source, destination and intermediate modules, as well as modules that generate control signals which interfere with the information flow.

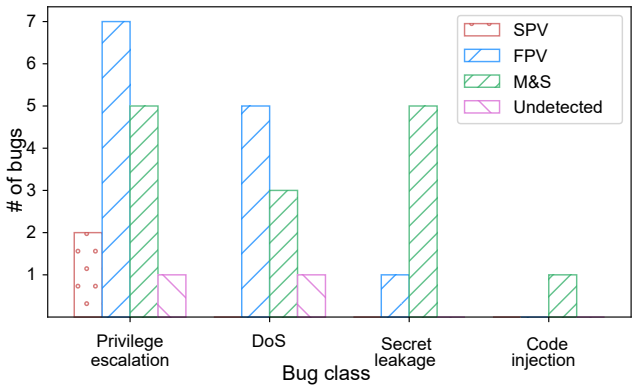

FIGURE 3: Verification results grouped by bug class and the number of bugs in each class detected by Security Path Verification (SPV), Formal Property Verification (FPV) and manual inspection and simulation techniques (M&S).

6.2 Detection Results

Of the 31 bugs we investigated, shown in Table 1, using the formal verification techniques described above, only 15 (48%) were detected. While we attempted to detect all 31 bugs formally, we were able to formulate security properties for only 17 bugs. This indicates that the main challenge with using formal verification tools is identifying and expressing security properties that the tools are capable of capturing and checking. Bugs due to ambiguous specifications of interconnect logic, for instance, are examples of bugs that are difficult to create security properties for.

Our results, shown in Figure 3, indicate that privilege escalation and denial-of-service (DoS) bugs were the most detected at 60% and 67% respectively. Secret leakage only had a 17% detection rate due to incorrect design specification for one bug, state explosion and the inability to express properties that the tool can assert for the remaining bugs. The code injection bug was undetected by formal techniques. Bugs at the interconnect level of the SoC such as bugs #1 and #2 were especially challenging since they involved a large number of highly complex and inter-connected modules that needed to be loaded and modeled by the tool (see L-1 in Section 3.1). Bug #20, which involves hardware/firmware interactions, was also

detected by neither the state-of-the-art FPV nor SPV since they analyze the RTL in isolation (see L-4 in Section 3.1). We describe these bugs in more detail in Appendix C.

6.3 State-Explosion Problem

Formal verification techniques are quickly driven into state space explosion when analyzing large designs with many states. Many large interconnected RTL modules, like those relevant to bugs #1 and #2, can have states in the order of magnitude of 10^{20}. Even smaller ones, like these used for bugs #3 and #4, can have a very large number of states, as shown in Table 1. When combined, the entire SoC will have a total number of states significantly higher than any of the results in Table 1. Attempting to model the entire SoC drove the tool into state explosion, and it ran out of memory and crashed. Formal verification tools, including those specific to security verification are currently incapable of handling so many states, even when computational resources are increased. This is further aggravated for industry-standard complex SoCs.

Because the entire SoC cannot be modeled and analyzed at once, detecting cross-modular bugs becomes very challenging. Engineers work around this (not fundamentally solve it) by adopting a divide-and-conquer approach and selecting which modules are relevant for the properties being tested and which can be black-boxed or abstracted. However, this is time-consuming, non-automated, error-prone, and requires expertise and knowledge of both the tools and design. By relying on the human factor, the tool can no longer guarantee the absence of bugs for the entire design, which is the original advantage of formal verification.

7 Discussion and Future Work

We now describe why microcode patching is insufficient for RTL bugs while emphasizing the need for advancing the hardware security verification process. We discuss the additional challenges of the overall process, besides the limitations of the industry-standard tools, which is the focus of this work.

7.1 Microcode Patching

While existing industry-grade SoCs support hotfixes by *microcode patching* for instance, this approach is limited to a handful of changes to the instruction set architecture, e.g., modifying the interface of individual complex instructions and adding or removing instructions [25]. Some vulnerabilities cannot even be patched by microcode, such as the recent Spoiler attack [33]. Fundamentally mitigating this requires fixing the hardware of the memory subsystem at the hardware design phase. For legacy systems, the application developer is advised to follow best practices for developing side channel-resilient software[5]. For vulnerabilities that can be patched, patches at this higher abstraction level in the firmware only act as a "symptomatic" fix that circumvent the RTL bug. However, they do not fundamentally patch the bug in the RTL, which is already realized as hardwired logic. Thus, microcode patching is a fallback for RTL bugs discovered after production, when you can not patch the RTL. They may also incur performance impact [6] that could be avoided if the underlying problem is discovered and fixed during design.

7.2 Additional Challenges in Practice

Functional vs. Security Specifications. As described in Section 2, pre- and post-silicon validation efforts are conducted to verify that the implementation fully matches both its functional and security specifications. The process becomes increasingly difficult (almost impossible) as the system complexity increases and specification ambiguity arises. Deviations from specification occur due to either functional or security bugs, and it is important to distinguish between them. While functional bugs generate functionally incorrect results, security bugs are not reflected in functionality. They arise due to unconsidered and corner threat cases that are unlikely to get triggered, thus making them more challenging to detect and cover. It is, therefore, important to distinguish between functional and security specifications, since these are often the references for different verification teams working concurrently on the same RTL implementation.

Specification Ambiguity. Another challenge entails anticipating and identifying all the security properties that are required in a real-world scenario. We analyzed the efficacy of industry-standard tools in a controlled setting—where we have prior knowledge of the bugs. However, in practice hardware validation teams do not have prior knowledge of the bugs. Security specifications are often incomplete and ambiguous, only outlining the required security properties under an assumed adversary model. These specifications are invalidated once the adversary model is changed. This is often the case with IP reuse, where the RTL code for one product is re-purposed for another with a different set of security requirements and usage scenarios. Parameters may be declared multiple times and get misinterpreted by the tools, thus causing bugs to slip undetected. Furthermore, specs usually do not specify bugs and information flows that should not exist, and there is no automated approach to determine whether one is proving the intended properties. Thus, a combination of incomplete or incorrect design decisions and implementation errors can easily introduce bugs to the design.

[5] https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00238.html

[6] https://access.redhat.com/articles/3307751

7.3 Future Research Directions

Through our work, we shed light on the limitations of state-of-the-art verification techniques. In doing so, we hope to motivate further research in advancing these techniques to adequately capture and detect these vulnerabilities.

Although manual RTL inspection is generally useful and can potentially cover a wide array of bugs, its efficacy depends exclusively on the expertise of the engineer. This can be inefficient, unreliable and ad hoc in light of rapidly evolving chip designs. Exhaustive testing of specifications through simulation requires amounts of resources exponential in the size of the input (i.e., design state space) while coverage must be intelligently maximized. Hence, current approaches face severe scalability challenges, as diagnosing software-exploitable bugs that reside deep in the design pipeline can require simulation of trillions of cycles [14]. Our results indicate that it is important to first identify high-risk components due to software exposure, such as password checkers, crypto cores, and control registers, and prioritize analyzing them. Scalability due to complex inter-dependencies among modules is one challenge for detection. Vulnerabilities associated with non-register states (such as caches) or clock-cycle dependencies (i.e., timing flows) are another open problem. Initial research is underway [71] to analyze a limited amount of low-level firmware running on top of a simulated RTL design for information and timing flow violations. However, these approaches are still in their infancy and yet to scale for real-world SoC designs.

8 Related Work

We now present related work in hardware security verification while identifying limitations with respect to detecting HardFails. We also provide an overview of recent software attacks exploiting underlying hardware vulnerabilities.

8.1 Current Detection Approaches

Security-aware design of hardware has gained significance only recently as the critical security threat posed by hardware vulnerabilities became acutely established. Confidentiality and integrity are the commonly investigated properties [19] in hardware security. They are usually expressed using information flow properties between entities at different security levels. Besides manual inspection and simulation-based techniques, systematic approaches proposed for verifying hardware security properties include formal verification methods such as proof assistance, model-checking, symbolic execution, and information flow tracking. We exclude the related work in testing mechanisms, e.g., JTAG/scan-chain/built-in self-test, because they are leveraged for hardware testing **after** fabrication. However, the focus of this work is on verifying the security of the hardware **before** fabrication. Interestingly, this includes verifying that the test mechanisms are correctly implemented in the RTL, otherwise they may constitute security vulnerabilities when used after fabrication (see bugs#9,#10,#11,#12,#16, #26 of the JTAG/debug interface).

Proof assistant and theorem-proving methods rely on mathematically modeling the system and the required security properties into logical theorems and formally proving if the model complies with the properties. VeriCoq [7] based on the Coq proof assistant transforms the Verilog code that describes the hardware design into proof-carrying code.VeriCoq supports the automated conversion of only a subset of Verilog code into Coq. However, this assumes accurate labeling of the initial sensitivity labels of each and every signal in order to effectively track the flow of information. This is cumbersome, error-prone, generates many faluse positives, and does not scale well in practice beyond toy examples. Moreover, timing (and other) side-channel information flows are not modeled. Finally, computational scalability to verifying real-world complex SoCs remains an issue given that the proof verification for a single AES core requires $\approx$ 30 minutes to complete [6].

Model checking-based approaches check a given property against the modeled state space and possible state transitions using provided invariants and predefined conditions. They face scalability issues as computation time scales exponentially with the model and state space size. This can be alleviated by using abstraction to simplify the model or constraining the state space to a bounded number of states using assumptions and conditions. However, this introduces false positives, may miss vulnerabilities, and requires expert knowledge. Most industry-leading tools, such as the one we use in this work, rely on model checking algorithms such as boolean satisfiability problem solvers and property specification schemes, e.g., assertion-based verification to verify the required properties of a given hardware design.

Side-channel leakage modeling and detection remain an open problem. Recent work [76] uses the Murφ model checker to verify different hardware cache architectures for side-channel leakage against different adversary models. A formal verification methodology for SGX and Sanctum enclaves under a limited adversary was introduced in [67]. However, such approaches are not directly applicable to hardware implementation. They also rely exclusively on formal verification and remain inherently limited by the underlying algorithms in terms of scalability and state space explosion, besides demanding particular expertise to use.

Information flow analysis (such as SPV) works by assigning a security label (or a *taint*) to a data input and monitoring the taint propagation. In this way, the designer can verify whether the system adheres to the required security policies. Recently, information flow tracking (IFT) has been shown effective in identifying security vulnerabilities, including timing side channels and information-leaking hardware Trojans.

IFT techniques are proposed at different levels of abstraction: gate-, RT, and language-levels. Gate-level information

flow tracking (GLIFT) [2, 58, 70] performs the IFT analysis directly at gate-level by generating GLIFT analysis logic that is derived from the original logic and operates in parallel to it. Although gate-level IFT logic is easy to automatically generate, it does not scale well. Furthermore, when IFT uses strict non-interference, it taints any information flow conservatively as a vulnerability [34] which scales well for more complex hardware, but generates too many false positives.

At the language level, Caisson [42] and Sapper [41] are security-aware HDLs that use a typing system where the designer assigns security "labels" to each variable (wire or register) based on the security policies required. However, they both require redesigning the RTL using a new hardware description language which is not practical. SecVerilog [22, 75] overcomes this by extending the Verilog language with a dynamic security type system. Designers assign a security label to each variable (wire or register) in the RTL to enable a compile-time check of hardware information flow. However, this involves complex analysis during simulation to reason about the run-time behavior of the hardware state and dependencies across data types for precise flow tracking.

Hardware/firmware co-verification to capture and verify hardware/firmware interactions remains an open challenge and is not available in widely used industry-standard tools. A co-verification methodology [28] addresses the semantic gap between hardware and firmware by modeling hardware and firmware using instruction-level abstraction to leverage software verification techniques. However, this requires modeling the hardware that interacts with firmware into an abstraction which is semi-automatic, cumbersome, and lossy.

While research is underway [71] to analyze a limited amount of low-level firmware running on top of a simulated RTL design these approaches are still under development and not scalable. Current verification approaches focus on register-state information-flow analysis, e.g., to monitor whether sensitive locations are accessible from unprivileged signal sources. Further research is required to explicitly model non-register states and timing explicitly alongside the existing capabilities of these tools.

8.2 Recent Attacks

We present and cautiously classify the underlying hardware vulnerabilities of recent cross-layer exploits (see Table 2 in Appendix B), using the categories introduced in 3.1. We do not have access to proprietary processor implementations, so our classification is only based on our deductions from the published technical descriptions. Yarom et al. demonstrate that software-visible side channels can exist even below cache-line granularity in CacheBleed [74]–undermining a core assumption of prior defenses, such as scatter-gather [9]. MemJam [45] exploits false read-after-write dependencies in the CPU to maliciously slow down victim accesses to memory blocks within a cache line. We categorize the underlying vulnerabilities of CacheBleed and MemJam as potentially hard to detect in RTL due to the many cross-module connections involved and the timing-flow leakage. The timing flow leakage is caused by the software triggering clock cycle differences in accesses that map to the same bank below cache line granularity, thus breaking constant-time implementations.

The TLBleed [23] attack shows how current TLB implementations can be exploited to break state-of-the-art cache side-channel protections. As described in Section 4, TLBs are typically highly interconnected with complex processor modules, such as the cache controller and memory management unit, making vulnerabilities therein very hard to detect through automated verification or manual inspection.

BranchScope [20] extracts information through the directional branch predictor, thus bypassing software mitigations that prevent leakage via the BTB. We classify it as a cache-state gap in branch prediction units, which is significantly challenging to detect using existing RTL security verification tools, which cannot capture and verify cache states. Meltdown [43] exploits speculative execution on modern processors to completely bypass all memory access restrictions. Van Bulck et al. [72] also demonstrated how to apply this to Intel SGX. Similarly, Spectre [37] exploits out-of-order execution across different user-space processes as arbitrary instruction executions would continue during speculation. We recognize these vulnerabilities are hard to detect due to scalability challenges in existing tools, since the out-of-order scheduling module is connected to many subsystems in the CPU. Additionally, manually inspecting these interconnected complex RTL modules is very challenging and cumbersome.

CLKScrew [69] abuses low-level power-management functionality that is exposed to software to induce faults and glitches dynamically at runtime in the processor. We categorize CLKScrew to have vulnerable hardware-firmware interactions and timing-flow leakage, since it directly exposes clock-tuning functionality to attacker-controlled software.

9 Conclusion

Software security bugs and their impact have been known for many decades, with a spectrum of established techniques to detect and mitigate them. However, the threat of hardware security bugs has only recently become significant as cross-layer exploits have shown that they can completely undermine software security protections. While some hardware bugs can be patched with microcode updates, many cannot, often leaving millions of affected chips in the wild. In this paper, we presented the first testbed of RTL bugs and systematically analyzed the effectiveness of state-of-the-art formal verification techniques, manual inspection and simulation methods in detecting these bugs. We organized an international hardware security competition and an in-house study. Our results have shown that 54 teams were only able to detect 61% of the total number of bugs, while with industry-leading formal verification techniques, we were only able to detect 48% of

the bugs. We showcase that the grave security impact of many of these undetected bugs is only further exacerbated by being software-exploitable.

Our investigation revealed the limitations of state-of-the-art verification/detection techniques with respect to detecting certain classes of hardware security bugs that exhibit particular properties. These approaches remain limited in the face of detecting vulnerabilities that require capturing and verifying complex cross-module inter-dependencies, timing flows, cache states, and hardware-firmware interactions. While these effects are common in SoC designs, they are difficult to model, capture, and verify using current approaches. Our investigative work highlights the necessity of treating the detection of hardware bugs as significantly as that of software bugs. Through our work, we highlight the pressing call for further research to advance the state of the art in hardware security verification. Particularly, our results indicate the need for increased scalability, efficacy and automation of these tools, making them easily applicable to large-scale commercial SoC designs—without which software protections are futile.

Acknowledgments

We thank our anonymous reviewers and shepherd, Stephen Checkoway, for their valuable feedback. The work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) by CRC 1119 CROSSING P3, and the Office of Naval Research (ONR Award #N00014-18-1-2058). We would also like to acknowledge the co-organizers of Hack@DAC: Dan Holcomb (UMass-Amherst), Siddharth Garg (NYU), and Sourav Sudhir (TAMU), and the sponsors of Hack@DAC: the National Science Foundation (NSF CNS-1749175), NYU CCS, Mentor - a Siemens Business and CROSSING, as well as the participants of Hack@DAC.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. *ACM conference on Computer and communications security*, pages 340–353, 2005.

[2] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design. *Design, Automation & Test in Europe*, pages 1695–1700, 2017.

[3] ARM. Security technology building a secure system using trustzone technology (white paper). http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[4] R. Armstrong, R. Punnoose, M. Wong, and J. Mayo. Survey of Existing Tools for Formal Verification. Sandia National Laboratories https://prod.sandia.gov/techlib-noauth/access-control.cgi/2014/1420533.pdf, 2014.

[5] Averant. Solidify. http://www.averant.com/storage/documents/Solidify.pdf, 2018.

[6] M.-M. Bidmeshki, X. Guo, R. G. Dutta, Y. Jin, and Y. Makris. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part II: Framework Automation. *IEEE Transactions on Information Forensics and Security*, 12(10):2430–2443, 2017.

[7] M.-M. Bidmeshki and Y. Makris. VeriCoq: A Verilog-to-Coq Converter for Proof-Carrying Hardware Automation. *IEEE International Symposium on Circuits and Systems*, pages 29–32, 2015.

[8] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. *Network and Distributed System Security Symposium (NDSS)*, 2019.

[9] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[10] Cadence. Incisive Enterprise Simulator. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html, 2014.

[11] Cadence. JasperGold Formal Verification Platform. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html, 2014.

[12] Cadence. JasperGold Security Path Verification App. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html, 2018. Last accessed on 09/09/18.

[13] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. *USENIX Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[14] D. P. Christopher Celio, Krste Asanovic. The Berkeley Out-of-Order Machine. https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf, 2016.

[15] Cisco. Cisco: Strengthening Cisco Products. https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle.html, 2017.

[16] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. *Tools for Practical Software Verification*, 2012.

[17] K. Conger. Apple announces long-awaited bug bounty program. https://techcrunch.com/2016/08/04/apple-announces-long-awaited-bug-bounty-program/, 2016.

[18] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. *USENIX Security Symposium*, pages 857–874, 2016.

[19] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer. Survey of approaches for security verification of hardware/software systems. https://eprint.iacr.org/2016/846.pdf, 2016.

[20] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[21] F. Farahmandi, Y. Huang, and P. Mishra. Formal Approaches to Hardware Trust Verification. *The Hardware Trojan War*, 2018.

[22] 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. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 555–568, 2017.

[23] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. *USENIX Security Symposium*, 2018.

[24] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379, 2016.

[25] M. Hicks, C. Sturton, S. T. King, and J. M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2015.

[26] M. Howard and S. Lipner. The Security Development Lifecycle. *Microsoft Press Redmond*, 2006.

[27] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. *IEEE Symposium on Security and Privacy*, 2016.

[28] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik. Formal Security Verification of Concurrent Firmware in SoCs Using Instruction-level Abstraction for Hardware. *ACM Annual Design Automation Conference*, pages 91:1–91:6, 2018.

[29] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. *Symposium on Security and Privacy*, 2013.

[30] F. Inc. Common Vulnerability Scoring System v3.0. https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf, 2018.

[31] Intel. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx, 2016. Last accessed on 09/05/18.

[32] Intel. Intel Bug Bounty Program. https://www.intel.com/content/www/us/en/security-center/bug-bounty-program.html, 2018.

[33] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. https://arxiv.org/abs/1903.00446, 2019.

[34] R. Kastner, W. Hu, and A. Althoff. Quantifying Hardware Security Using Joint Information Flow Analysis. *IEEE Design, Automation & Test in Europe*, pages 1523–1528, 2016.

[35] H. Khattri, N. K. V. Mangipudi, and S. Mandujano. Hsdl: A security development lifecycle for hardware technologies. *IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 116–121, 2012.

[36] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[37] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. http://arxiv.org/abs/1801.01203, 2018.

[38] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization*, 2004.

[39] D. Lee. Keystone enclave: An open-source secure enclave for risc-v. https://keystone-enclave.org/, 2018.

[40] Lenovo. Lenovo: Taking Action on Product Security. https://www.lenovo.com/us/en/product-security/about-lenovo-product-security, 2017.

[41] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A Language for Hardware-level Security Policy Enforcement. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–112, 2014.

[42] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 46(6):109–120, 2011.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. https://arxiv.org/abs/1801.01207, 2018.

[44] Mentor. Questa Verification Solution. https://www.mentor.com/products/fv/questa-verification-platform, 2018.

[45] A. Moghimi, T. Eisenbarth, and B. Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

[46] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[47] NIST. HP: Remote update feature in HP LaserJet printers does not require password. https://nvd.nist.gov/vuln/detail/CVE-2004-2439, 2004.

[48] NIST. Microsoft: Hypervisor in Xbox 360 kernel allows attackers with physical access to force execution of the hypervisor syscall with a certain register set, which bypasses intended code protection. https://nvd.nist.gov/vuln/detail/CVE-2007-1221, 2007.

[49] NIST. Apple: Multiple heap-based buffer overflows in the AudioCodecs library in the iPhone allows remote attackers to execute arbitrary code or cause DoS via a crafted AAC/MP3 file. https://nvd.nist.gov/vuln/detail/CVE-2009-2206, 2009.

[50] NIST. Broadcom Wi-Fi chips denial of service. https://nvd.nist.gov/vuln/detail/CVE-2012-2619, 2012.

[51] NIST. Vulnerabilities in Dell BIOS allows local users to bypass intended BIOS signing requirements and install arbitrary BIOS images. https://nvd.nist.gov/vuln/detail/CVE-2013-3582, 2013.

[52] NIST. Google: Escalation of Privilege Vulnerability in MediaTek WiFi driver. https://nvd.nist.gov/vuln/detail/CVE-2016-2453, 2016.

[53] NIST. Samsung: Page table walks conducted by MMU during Virtual to Physical address translation leaves in trace in LLC. https://nvd.nist.gov/vuln/detail/CVE-2017-5927, 2017.

[54] NIST. AMD: Backdoors in security co-processor ASIC. https://nvd.nist.gov/vuln/detail/CVE-2018-8935, 2018.

[55] NIST. AMD: EPYC server processors have insufficient access control for protected memory regions. https://nvd.nist.gov/vuln/detail/CVE-2018-8934, 2018.

[56] NIST. Buffer overflow in bootrom recovery mode of nvidia tegra mobile processors. https://nvd.nist.gov/vuln/detail/CVE-2018-6242, 2018.

[57] J. Oberg. Secure Development Lifecycle for Hardware Becomes an Imperative. https://www.eetimes.com/author.asp?section_id=36&doc_id=1332962, 2018.

[58] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. *IEEE/ACM Design Automation Conference*, pages 244–247, 2010.

[59] PULP Platform. Ariane. https://github.com/pulp-platform/ariane, 2018.

[60] PULP Platform. Pulpino. https://github.com/pulp-platform/pulpino, 2018.

[61] PULP Platform. Pulpissimo. https://github.com/pulp-platform/pulpissimo, 2018.

[62] Qualcomm. Qualcomm Announces Launch of Bounty Program. https://www.qualcomm.com/news/releases/2016/11/17/qualcomm-announces-launch-bounty-program-offering-15000-usd-discovery, 2018.

[63] Samsung. Rewards Program. https://security.samsungmobile.com/rewardsProgram.smsb, 2018.

[64] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[65] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *ACM Symposium on Computer and Communication Security*, pages 552–561, 2007.

[66] O. Solutions. OneSpin 360. https://www.onespin.com/fileadmin/user_upload/pdf/datasheet_dv_web.pdf, 2013.

[67] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. *ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, 2017.

[68] Sunny .L He and Natalie H. Roe and Evan C. L. Wood and Noel Nachtigal and Jovana Helms. Model of the Product Development Lifecycle. https://prod.sandia.gov/techlib-noauth/access-control.cgi/2015/159022.pdf, 2015.

[69] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. *USENIX Security Symposium*, pages 1057–1074, 2017.

[70] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete Information Flow Tracking from the Gates Up. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, 2009.

[71] Tortuga Logic. Verifying Security at the Hardware/Software Boundary. http://www.tortugalogic.com/unison-whitepaper/, 2017.

[72] J. Van Bulck, F. Piessens, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[73] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2014.

[74] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017. 10.1007/s13389-017-0152-y.

[75] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2015.

[76] T. Zhang and R. B. Lee. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. *ACSAC*, pages 96–105, 2014.

Appendix

A Ariane Core and RTL Hierarchy

Figure 4 shows the high-level microarchitecture of the Ariane core to visualize its complexity. This RISC-V core is far less complex than an x86 or ARM processor and their more sophisticated microarchitectural and optimization features.

Figure 5 illustrates the hierarchy of the RTL components of the Ariane core. This focuses only on the core and excludes all uncore components, such as the AXI interconnect, peripherals, the debug module, boot ROM, and RAM.

B Recent Microarchitectural Attacks

We reviewed recent microarchitectural attacks with respect to existing hardware verification approaches and their limitations. We observe that the underlying vulnerabilities would be difficult to detect due to the properties that they exhibit, rendering them as potential HardFails. We do not have access to their proprietary RTL implementation and cannot inspect the underlying vulnerabilities. Thus, we only infer from the published technical descriptions and errata of these attacks the nature of the underlying RTL issues. We classify in Table 2 the properties of these vulnerabilities that represent challenges for state-of-the-art hardware security verification.

C Details on the Pulpissimo Bugs

We present next more detail on some of the RTL bugs used in our investigation.

Bugs in crypto units and incorrect usage: We extended the SoC with a faulty cryptographic unit with a multiplexer to select between AES, SHA1, MD5, and a temperature sensor. The multiplexer was modified such that a race condition occurs if more than one bit in the status register is enabled, causing unreliable behavior in these security critical modules.

Furthermore, both SHA-1 and MD5 are outdated and broken cryptographic hash functions. Such bugs are not detectable by formal verification, since they occur due to a specification/design issue and not an implementation flaw, therefore they are out of the scope of automated approaches and formal verification methods. The cryptographic key is

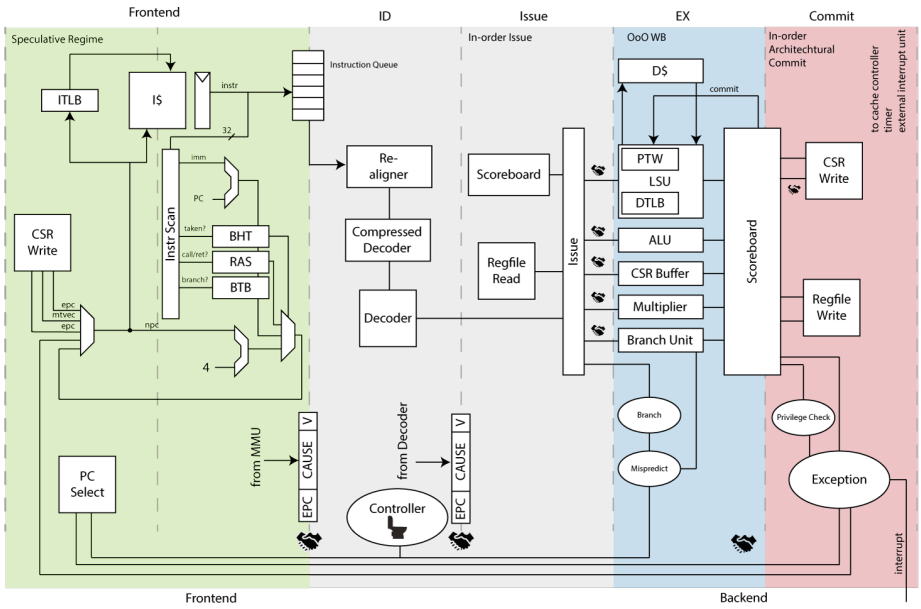

FIGURE 4: High-level architecture of the Ariane core [59].

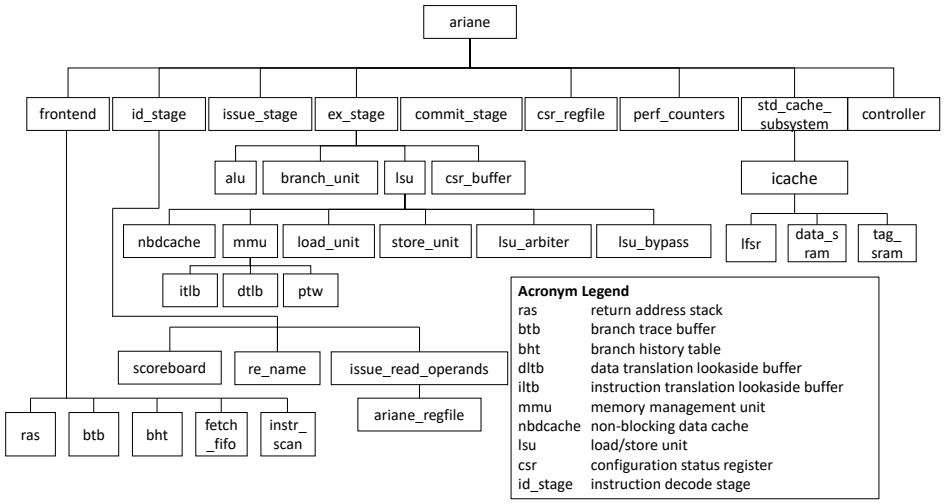

FIGURE 5: Illustration of the RTL module hierarchy of the Ariane core.

Attack	Privilege Level	Memory Corruption	Information Leakage	Cross-modular	HW/FW-Interaction	Cache-State Gap	Timing-Flow Gap	**HardFail**
Cachebleed [74]	unprivileged	✗	✓	✗	✗	✗	✓	✓
TLBleed [23]	unprivileged	✗	✓	✓	✗	✓	✓	✓
BranchScope [20]	unprivileged	✗	✓	✗	✗	✓	✗	✓
Spectre [37]	unprivileged	✗	✓	✓	✗	✓	✗	✓
Meltdown [43]	unprivileged	✗	✓	✓	✗	✓	✗	✓
MemJam [45]	supervisor	✗	✓	✓	✗	✗	✓	✓
CLKScrew [69]	supervisor	✓	✓	✗	✓	✗	✓	✓
Foreshadow [72]	supervisor	✓	✓	✓	✓	✓	✗	✓

TABLE 2: Classification of the underlying vulnerabilities of recent microarchitectural attacks by their HardFail properties.

stored and read from unprotected memory, allowing an attacker access to the key. The temperature sensor register value is incorrectly muxed as output instead of the crypto engine output and vice versa, which are illegal information flows that could compromise the cryptographic operations.

LISTING 1: **Incorrect use of crypto RTL:** The key input for the AES (g_input) is connected to signal b. This signal is then passed through various modules until it connects directly to a tightly coupled memory in the processor.

```
input logic [127:0] b,
...
aes_1cc aes(
  .clk(0),
  .rst(1),
  .g_input(b),
  .e_input(a),
  .o(aes_out)
  );
```

Bugs in security modes: We replaced the standard PULP_SECURE parameter in the riscv_cs_registers and riscv_int_controller modules with another constant parameter to permanently disable the security/privilege checks for these two modules. Another bug we inserted is switching the write and read protections for the AXI bus interface, causing erroneous checks for read and write accesses.

Bugs in the JTAG module: We implemented a JTAG password-checker and injected multiple bugs in it, including the password being hardcoded in the password checking file. The password checker also only checks the first 31 bits, which reduces the computational complexity of brute-forcing the password. The password checker does not reset the state of the correctness of the password when an incorrect bit is detected, allowing for repeated partial checks of passwords to end up unlocking the password checker. This is also facilitated by the fact that the index overflows after the user hits bit 31, allowing for an infinite cycling of bit checks.

D Exploiting Hardware Bugs From Software

We now explain how one of our hardware bugs can be exploited in real-world by software. This RTL vulnerability manifests in the following way. When an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This lets erroneous memory accesses slip through hardware checks at runtime. Armed with the knowledge about this vulnerability, an adversary can force memory access errors to evade the checks. As shown in Figure 6, the memory bus decoder unit (unit of the memory interconnect) is assumed to have the bug. This causes errors to be ignored

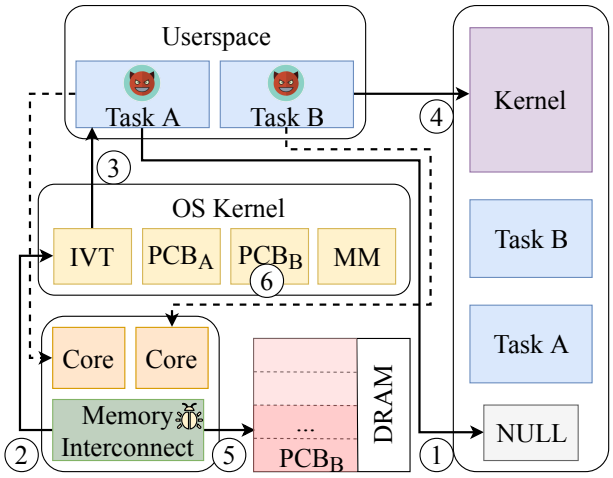

FIGURE 6: Our attack exploits a bug in the implementation of the memory bus of the PULPissimo SoC: by ① *spamming* the bus with invalid transactions an adversary can make ④ malicious write requests be set to `operational`.

under certain conditions (see bug number #7 in Table 1). In the first step ①, the attacker generates a user program (Task A) that registers a dummy signal handler for the segmentation fault (`SIGSEGV`) access violation. Task A then executes a loop with ② a faulting memory access to an invalid memory address (e.g., `LW x5, 0x0`). This will generate an error in the memory subsystem of the processor and issue an invalid memory access interrupt (i.e., `0x0000008C`) to the processor. The processor raises this interrupt to the running software (in this case the OS), using the pre-configured interrupt handler routines in software. The interrupt handler in the OS will then forward this as a signal to the faulting task ③, which keeps looping and continuously generating invalid accesses. Meanwhile, the attacker launches a separate Task B, which will then issue a single memory access ④ to a privileged memory location (e.g., `LW x6, 0xf77c3000`). In this situation, multiple outstanding memory transactions will be generated on the memory bus, all of which but one will be flagged as faulty by the address decoder. An invalid memory access will always proceed the single access of Task B. Due to the bug in the memory bus address decoder, ⑤ the malicious memory access will become `operational` instead of triggering an error. Thus, the attacker can issue read and write instructions to arbitrary privileged (and unprivileged) memory by forcing the malicious illegal access to be preceded with a faulty access. Using this technique the attacker can eventually leverage this read-write primitive, e.g., ⑥ to escalate privileges by writing the process control block (PCB_B) for his task to elevate the corresponding process to root. This bug leaves the attacker with access to a root process, gaining control over the entire platform and potentially compromising all the processes running on the system.