TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities

Rahul Kande, Addison Crump, and Garrett Persyn, *Texas A&M University;*
Patrick Jauernig and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt;*
Aakash Tyagi and Jeyavijayan Rajendran, *Texas A&M University*

https://www.usenix.org/conference/usenixsecurity22/presentation/kande

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities

Rahul Kande[†], Addison Crump[†], Garrett Persyn[†], Patrick Jauernig[*], Ahmad-Reza Sadeghi[*],
Aakash Tyagi[†], and Jeyavijayan Rajendran[†]
[†]*Texas A&M University, USA,* [*]*Technische Universität Darmstadt, Germany*
[†]{rahulkande, addisoncrump, gpersyn, tyagi, jv.rajendran}@tamu.edu,
[*]{patrick.jauernig, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

The increasing complexity of modern processors poses many challenges to existing hardware verification tools and methodologies for detecting security-critical bugs. Recent attacks on processors have shown the fatal consequences of uncovering and exploiting hardware vulnerabilities.

Fuzzing has emerged as a promising technique for detecting software vulnerabilities. Recently, a few hardware fuzzing techniques have been proposed. However, they suffer from several limitations, including non-applicability to commonly-used hardware description languages (HDLs) like Verilog and VHDL, the need for significant human intervention, and inability to capture many intrinsic hardware behaviors, such as signal transitions and floating wires.

In this paper, we present the design and implementation of a novel hardware fuzzer, *TheHuzz*, that overcomes the aforementioned limitations and significantly improves the state of the art. We analyze the intrinsic behaviors of hardware designs in HDLs and then measure the coverage metrics that model such behaviors. *TheHuzz* generates assembly-level instructions to increase the desired coverage values, thereby finding many hardware bugs exploitable from software. We evaluate *TheHuzz* on four popular open-source processors and achieve 1.98× and 3.33× the speed compared to the industry-standard random regression approach and the state-of-the-art hardware fuzzer, DifuzzRTL, respectively. Using *TheHuzz*, we detected 11 bugs in these processors, including 8 new bugs, and we demonstrate exploits using the detected bugs. We also show that *TheHuzz* overcomes the limitations of formal verification tools from the semiconductor industry by comparing its findings to those discovered by the Cadence JasperGold tool.

1 Introduction

Modern processors are becoming increasingly complex with sophisticated functional and security mechanisms and extensions. This development, however, increases the chance of introducing vulnerabilities into the hardware design and implementation which can lead to errors and exploitation attacks with fatal consequences. Hardware vulnerabilities range from functional bugs (e.g., [37]) to emerging security-critical vulnerabilities that have been uncovered and exploited (e.g., [36],[45]), and both affect commodity processors and their dedicated security extensions (e.g., [11], [81]). The hardware common weakness enumeration (CWE) lists numerous hardware vulnerabilities whose impact spans not only the hardware but also software [48]. It is crucial to discover hardware vulnerabilities in the early stages of the design cycle.

Various hardware vulnerability detection techniques and tools have been proposed or developed by both academia and industry, such as formal verification [10, 74, 68, 6, 55, 85, 14, 13, 61], run-time detection [27, 64, 84], information flow tracking [78, 2, 43, 42, 92], and the recent efforts towards fuzzing hardware [51, 79, 39, 30], which is our focus.

While formal verification tools can efficiently find bugs in smaller designs, they are unable to cope with the increasing complexity of modern, large designs and are becoming less efficient in detecting bugs, especially security vulnerabilities [14, 47, 89, 12, 17]. One particular reason is that these tools rely heavily on human expertise to engineer or specify "attack scenarios" for verification. For instance, the popular industrial formal verification tool, Cadence's JasperGold [10], has been evaluated against a crowd-sourced vulnerability detection effort from 54 competing teams participating in a hardware capture-the-flag competition [14]. The results were based on security bugs mimicking real-world common vulnerabilities and exposures (CVEs) [49]. While JasperGold detected 48% of the bugs, manual inspection with simulation detected 61% of the bugs, highlighting issues like state explosion and scalability of the existing techniques, amongst others.

Another approach to find hardware security bugs is runtime detection techniques, which hardcode assertions in hardware to check security violations at runtime [27, 64, 84]. However, these techniques detect bugs only post-fabrication and unlike software, hardware is not easily patchable.

Information-flow tracking (IFT) techniques analyze the hardware to detect security vulnerabilities by labeling all the input signals and propagating this label throughout the design to identify information leakage or tampering of critical data [78, 2, 43, 42, 92]. Although IFT can analyze designs with several thousand lines of code, the labels often get polluted with unwanted signals, resulting in many false positives. The initial labels have to be assigned manually, which can be error-prone, and require expert knowledge of the design.

Hence, there is an increasing need for methodologies and tools to detect hardware vulnerabilities that are scalable to large and complex designs, highly automatic, effective and efficient in detecting security-critical vulnerabilities that are exploitable (and not just only functional bugs), compatible with existing chip design and verification flows, applicable to different hardware models (register-transfer level, gate-level, transistor-level, taped-out chip), and account for different hardware behaviors (signal transitions, finite-state machines (FSMs), and floating wires).

A promising technique extensively used for software vulnerability detection is fuzzing. Fuzzing uses random generation of test cases to detect invalid states in the target[46]. While it seems natural to apply or extend a software fuzzer to detect security bugs in hardware [79, 51], such approaches do not capture hardware-intrinsic behaviors, for instance, signal transitions of wires, FSMs, and floating wires, defined in hardware description languages (HDLs) like Verilog and VHDL. We will discuss these challenges in Section 3.

So far, there have been a few proposals towards fuzzing hardware [39, 51, 79, 30]. However, as we elaborate in Section 7, they suffer from various limitations: lack of support for commonly-used HDLs such as VHDL and Verilog [39] or only partially supporting their constructs [79], strong reliance on human intervention [51], and the inherent inability of capturing many hardware behaviors, including transitioning of logical values in wires and of floating wires [30].

Our goals and contributions. We present the design and implementation of a novel hardware fuzzer, *TheHuzz*. It tackles the challenges of building a hardware fuzzer (cf. Section 3) and addresses the aforementioned shortcomings of the current hardware fuzzing proposals (cf. Section 4). We analyze the intrinsic behaviors of hardware designs and describe appropriate coverage metrics of the HDL to capture such behaviors. Given the importance of software-exploitable hardware vulnerabilities [14, 47, 89, 12, 17], *TheHuzz* fuzzes the target hardware by testing instruction sequences, thereby discovering security bugs that are exploitable by the software code which executes such instruction sequences. Through a built-in optimizer, *TheHuzz* can select the best instructions and mutation techniques to achieve the best coverage.

TheHuzz (i) supports commonly-used HDLs like Verilog and VHDL, (ii) is compatible with conventional industry-standard IC design and verification flow, (iii) detects software-exploitable hardware vulnerabilities, (iv) accounts for different hardware behaviors, (v) does not require knowledge of the design, (vi) is scalable to large-scale designs, and (vii) does not need human intervention.

In summary, our main contributions are:

- We present a novel hardware fuzzer, *TheHuzz*, (Section 4), which uses coverage metrics that capture a wide variety of hardware behaviors, including signal transitions, floating wires, multiplexers, along with combinational and sequential logic. *TheHuzz* optimizes the selection of the best instructions and mutation techniques and can achieve high coverage rates (cf. Section 4.4). Our fuzzer achieves 1.98× and 3.33× the speed compared to the industry-standard random regression approach and the state-of-the-art hardware fuzzer, DifuzzRTL, respectively (cf. Section 6.4).
- We extensively evaluate our fuzzer, *TheHuzz*, on four well-known and complex real-world open-source processor designs from two different open-source instruction set architectures (ISAs): (i) or1200 processor (OpenRISC ISA), (ii) mor1kx processor (OpenRISC ISA), (iii) Ariane processor (RISC-V ISA), and (iv) Rocket Core (RISC-V ISA). All these processors can run Linux-based operating systems and are used in multiple hardware verification research studies [14, 27, 94, 93].
- *TheHuzz* found 11 bugs that are software exploitable in four different processors; eight of them are new bugs. We also showcase two attacks from unprivileged software exploiting vulnerabilities found by *TheHuzz* (cf. Section 6.2).
- We perform an investigation of the bugs detected by *TheHuzz* using a leading formal verification tool, Cadence's JasperGold [10] (cf. Section 6.5). *TheHuzz* overcomes the limitations of JasperGold: state explosion, intensive resource consumption, reliance upon error-prone human expertise, and a requirement of prior knowledge of hardware vulnerabilities or security properties.
- To foster research in the area of hardware fuzzing, we plan to open-source the code of *TheHuzz* to provide the community a framework to build upon.

2 Background

The growing number of attacks that exploit hardware vulnerabilities from software [37, 36, 45, 59, 52, 82, 76, 60, 34, 11, 81] call for new and effective hardware vulnerability detection techniques that address the limitations of existing methods and tools, such as state-space explosion, modeling hardware-software interactions, and the need for manual analysis.

2.1 Fuzzing

Fuzzing techniques are shown to be highly effective in detecting software vulnerabilities [75, 40, 46, 67, 23, 16, 22, 87]. Fuzzing generates test inputs and simulates the target design to detect vulnerabilities in it. The inputs are generated by *mutating* the previous inputs, which are generated from seeds. Mutation techniques modify the input by performing predefined operations, including bit-flip, clone, and swap. The mutation process also generates invalid inputs, testing the design outside the specification. In the past, fuzzers were created specifically to target different kinds of software: binary targets [40], JIT compilers [23], web applications [16], and operating systems [22]. Thus, specialized fuzzers conform to the needs of each target type. Fuzzers have seen use from both independent researchers and organizations as an additional verification step, most notably that of Google's OSS Fuzz [67], which actively fuzzes a plethora of software on their ClusterFuzz platform [20]. Fuzzers are highly successful in detecting software vulnerabilities as they are automated, are scalable to large codebases, do not require the knowledge of the underlying system, and are highly efficient in detecting many security vulnerabilities.

Unfortunately, comparable approaches for hardware fuzzing are still in their infancy. Hardware-specific behaviors pose several challenges to the design of hardware fuzzers, which we present in this section. However, before we consider the natural question of why one cannot trivially adopt the advances of software fuzzers for hardware, we briefly explain the typical hardware (security) development life cycle.

2.2 Hardware Development Lifecycle

The hardware development lifecycle [26, 7, 83, 50] typically begins with a design exploration driven by the market segment served by the product. Architects then engineer the optimal architecture while trading off among performance, area, and power, and the associated microarchitectural features. Designers implement all the microarchitectural modules using hardware description languages (HDLs), which are usually written at the register-transfer level (RTL). To this end, popular HDLs like Verilog and VHDL are used to describe complex hardware structures such as buses, controllers as finite state machines (FSMs), queues, and datapath units like adders, multipliers, etc. Electronic design automation (EDA) tools synthesize the RTL models into gate-level designs, which realize the hardware using Boolean gates, multiplexers, wires, and state elements like flip-flops. EDA tools then synthesize the gate-level design into transistor-level and eventually to layout, which is then sent to the foundry for manufacturing.

Most of the design effort and time spent by designers goes into manually writing HDLs at the RTL as the rest of the steps are highly automated. Unfortunately, writing HDL at the RTL is error-prone [7, 83, 50] . Thus, the verification team checks if the design at its various stages meets the required specification or not using functional, formal, and simulation-based tools; if the design does not meet the specification, the designers patch the bugs, and the process is repeated until the design passes the verification tests. To this end, companies typically develop a golden reference model (GRM)[*] for industry designs to be used with the conventional verification flows. GRMs are often written at a higher abstraction level (e.g., for RTL, the GRM is a software model of the hardware). Verification techniques usually compare the outputs of RTL and the GRM to find any mismatches, which will reveal the bugs. The accuracy of these techniques is further increased by comparing not only the final outputs but also the values of intermediate registers and by performing comparisons after every clock cycle. They perform similar tests on the gate-level design and the fabricated chip; for these models, the adjacent abstraction level acts as the GRM. Similarly, post-manufacturing, testing of the fabricated chips is performed to weed out the faulty chips.

When the architecture of the chip is designed, the security team concurrently identifies the threat model, security features, and assets. During the design phase, the security team performs security testing, starting with the RTL model via simulation and emulation, formal verification, and manual review of RTL code. Post-deployment, the security engineers provide support and patch any bugs, if possible.

[*]The GRM for hardware is similar to a test oracle in software which helps verify the result of a program's execution [28].

3 Challenges of Hardware Fuzzing

In this section, we outline the challenges that arise when analyzing hardware using fuzzing. We first elaborate on the problems that one encounters when deploying existing software fuzzers to analyze hardware. Then we discuss challenges that need to be tackled when designing and implementing a dedicated hardware fuzzer.

3.1 Fuzzing Hardware with Software Fuzzers

There are two ways to fuzz hardware with software fuzzers: (i) using software fuzzers directly on the hardware, and (ii) fuzzing hardware as software. However, both approaches face several limitations.

Problems with using software fuzzers directly on hardware. First, software fuzzers rely on a different behavior in vulnerability detection. They rely on software abstractions to find a bug by using the operating system or instrumenting software to monitor failure detection [54, 66]. Most software fuzzers use crashes to detect bugs, but hardware does not *crash* [79]. Thus, hardware fuzzers need to find their equivalent of crashes and memory leaks. Second, hardware simulations are slow. Typically, given a function, executing its software equivalent is faster than simulating its hardware model. Parallelization of hardware simulation is difficult due to the complex interdependencies in the hardware design [12]. Third, many software fuzzers rely on *instrumentation* of the software program to obtain feedback (e.g., AFL [40]) and use custom compilers (e.g., *afl-gcc*) to instrument the code [29, 19, 44, 40], but these compilers will not be able to instrument the hardware designs since they do not support HDLs such as Verilog and VHDL. One of the prior works, RFUZZ [39] made the first attempt towards solving this challenge: it uses hardware simulators to compile the hardware and applies a modified version of software fuzzer, AFL [40] to fuzz the hardware. However, this fuzzer is limited in terms of the scalability [30] and coverage (cf. Appendix B).

Problems with fuzzing hardware as software. Another strategy of fuzzing hardware using software fuzzers is to convert a HDL model into an equivalent software model using tools like Verilator [70], and then apply software fuzzers to the resultant software code [79]. Unfortunately, converting hardware into software models poses its own set of challenges.

First, applying existing software fuzzers on software models of hardware designs is, in general, inefficient. The software models of hardware designs need to account for properties unique to the working of the hardware, like computing all the register values for every clock cycle and bit manipulation operations, and components such as controllers, system bus, and queues—which makes the model computationally expensive. Moreover, software fuzzers use program crashes and instrumented memory safety checks to detect bugs in an application; these concepts cannot be trivially applied to hardware [79]. Instead, a well-defined specification to compare against is needed to detect incorrect logic implementations, timing violations, and unintended data flow or control flow.

Second, inferring actual hardware coverage from the generated software model is difficult. While software and hardware line and edge/block coverage are comparable in some instances [79], other forms of coverage may not be. A relatively simple operation in a HDL, like bit manipulation, may be significantly more complex in software. Conversely, a more complex component in HDL, such as a multiplexer, could be represented by a simple switch statement in software. Thus, one has to account for the effects of conversion.

Third, the hardware community has developed its own standards, processes, and flows for using verification methodologies and tools over several decades of research [50, 7, 83]. Any new approach has to be compatible with the hardware verification flow, as these methodologies have specialized data structures and algorithms geared towards hardware models and behaviors.

An open-source approach to solve the many challenges of fuzzing the software model of hardware is performed in [79]. This technique derives equivalences between the coverage metrics (e.g., line and FSM) used in hardware to that of software (e.g., line and edge). While this approach is promising, it does not scale to complex designs such as processors, which is the focus of this work (cf. Section 7).

3.2 Creating a Hardware Fuzzer

A hardware fuzzer needs to take into account the nature and requirements of hardware to improve efficiency. For example, Syzkaller [22], which specializes in kernel-fuzzing, incorporates system call signatures to generate better test cases. A hardware design fundamentally differs from any software program in terms of inputs, language used, feedback information available, and design complexity. Also, designing a hardware fuzzer has its own set of unique challenges, which are presented below. Multiple attempts have been made in the recent past towards building hardware fuzzers [39, 30, 51, 79] where each of these challenges are approached differently.

Input generation. For a hardware fuzzer to be efficient and effective, it should generate inputs in the format expected by the target processor. Directly applying the input-generation techniques used in software fuzzing is impossible as the input formats differ: while many software fuzzers take input files or a set of values assigned to a variable, the input to hardware is mostly continuous without a defined length [79]. Further, inputs to hardware can be generated at various hardware abstraction levels: architecture level, register-transfer level (RTL), gate level, and transistor level. Each level also has its own input representation, ranging from transaction packets, over continuous-time digital signals, to continuous-time analog signals. Hence, the major challenges in input generation are to *determine the suitable abstraction level to fuzz and the input representation* that maximizes the efficiency in finding vulnerabilities [12, 50, 51, 39, 30, 79] .

Another important aspect is the continuous nature of the hardware since it changes its state with every input (and/or time). Also, multiple FSMs can run in parallel, and one or more of them could enter in deadlock states, preventing the hardware from receiving inputs from the fuzzer [12]. For instance, a password checking module could be designed to lock itself forever after one incorrect password entry unless the system is reset. Hence, another crucial challenge is to *identify situations where the hardware simulation should be stopped or reset before applying new inputs.*

Finally, similar to how software fuzzers like syzkaller [22] encode functional dependencies (e.g., of system calls), hardware modules often need to be initialized to enable the fuzzer to test further functionality, e.g., an AES encryption module needs to be initialized with the key size and encryption mode before testing the actual encryption with plaintext and key. *Inferring these functional dependencies is highly challenging,* as such information is usually only available with a well-defined formal specification [51, 79].

Feedback mechanism. Exploring complex targets, especially hardware, often forces fuzzers to generate tremendous amounts of inputs, while making decisions like which mutation technique to use, when to stop mutating an input, and how to generate the seed inputs repeatedly. Rather than relying on randomly-generated inputs alone, a more efficient way is to analyze the impact of these parameters on the target processor and adapt input generation accordingly as done in feedback-guided fuzzing [65, 41]. Prior works [39, 30] addressed this challenge using hardware-friendly coverage metrics but fail to capture many hardware behaviors (cf. Appendix B).

Adapting software feedback mechanisms to hardware is

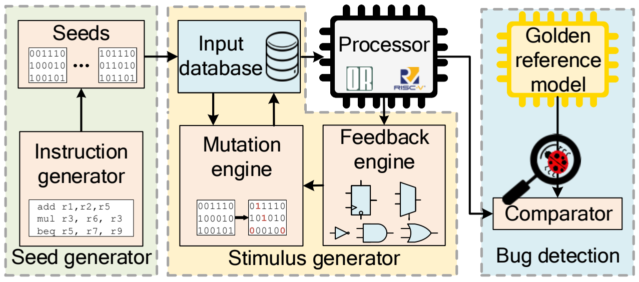

Figure 1: Framework of *TheHuzz*.

difficult due to the differences in execution/simulation for software and hardware [39, 30, 51, 79]. Instrumentation needs to be added to the hardware design such that the activities of different combinational and sequential structures, which are critical to the functionality of the hardware, can be traced. Although feedback-guided fuzzers have more potential to explore complex targets, capturing, analyzing, and processing the feedback data is challenging [65, 41]. This issue will become more profound in hardware since hardware designs are slower to simulate. One way to speedup hardware fuzzing is to use FPGA emulation, but instrumenting a design on an FPGA is challenging [39, 30]. Hence, the *feedback mechanism needs to capture the complex characteristics of hardware.*

Lastly, the performance of a fuzzer needs to be evaluated on hardware designs comparable to what is used in practice. However, unlike with software, commercial hardware designs like Intel's x86 processors do not have their source code available. Hence, a key challenge is to find *openly-available designs that are reasonably modern and complex.*

4 Design of Our Fuzzer, *TheHuzz*

TheHuzz is a novel hardware fuzzer that overcomes the challenges identified in Section 3.2. We directly fuzz the hardware design instead of the software model, thereby eliminating the need for hardware-to-software conversions and the associated equivalency checks. To overcome the slowness of hardware simulation, *TheHuzz* selects the optimal instructions and mutation techniques to use. *TheHuzz* is easily integratable with existing hardware design and verification methodologies—thereby, easily adaptable by companies—as our approach does not require any modification to the target processor and utilizes existing hardware simulation tools and techniques. We refer to the target processor as the design under test (DUT). Our fuzzer generates instructions as inputs to the DUT since we focus on software-exploitable processor vulnerabilities.

TheHuzz comprises three modules, as shown in Figure 1. First, the *seed generator* starts the fuzzing process by generating an initial sequence of instructions (*seeds* or *seed inputs*). Then, the *stimulus generator* generates new instruction sequences by mutating them, beginning with the *seeds*. These inputs are passed to the simulated RTL design of the DUT,

Listing 1: Chisel code of the case study.

```
36    // combinational logic for vld register
37    vld :=debug_en |(flush |en) // Bug b2
38
39    // select signal for mux
40    val sel1 =Wire(Bool())
41    sel1 :=((pass ===ipass) |debug_en) // Bug b1
42
43    // flush logic
44    val state_f =Wire(UInt(3.W))
45    when (flush &en){
46       state_f :=FLUSH
47    } otherwise {
48       state_f :=state
49    }
50
51    state :=(!sel1 &state_f) |(sel1 &D_READ)
```

which returns coverage feedback to the *stimulus generator* and trace information for *bug detection*. Finally, the *bug detection* mechanism compares the RTL simulation trace with that of a golden reference model (GRM) to find differences in execution, and hence, find bugs.

In the following, before we explain the modules of *TheHuzz*, we first analyze the intrinsic behaviors of designs at the RTL, as *TheHuzz* targets such behaviors, and describe the coverage metrics that capture those behaviors. Then, we describe the seed generator and stimulus generator of our fuzzer in detail and how they interact. Finally, we detail how we optimize the mutation engine and how the bugs are detected.

4.1 Hardware Design and Coverage Metrics

Hardware designs at RTL consist of combinational and sequential logic. Combinational logic is a time-independent circuit with boolean logic gates (e.g., AND, OR, XOR) and wires connecting them. Apart from building datapath units like adders and multipliers, these logic gates are used to build basic combinational structures like multiplexers (MUXes), demultiplexers, encoders, and decoders, which are in turn used in building complex blocks. Apart from combinational gates, sequential logic also uses registers, which are usually implemented using D flip-flops (DFFs). In the following, we explain the effectiveness of our fuzzer in capturing hardware behaviors over existing hardware fuzzers using a case study.

Case study. We now present a case study using a design with two bugs inspired by CVEs. First, we explain the intended behavior and then the bugs. Then, we detail *TheHuzz*'s coverage metrics and describe how they detect these bugs.

Consider a cache controller module—similar to the instruction cache controller of the Ariane processor [91]—shown in Listing 1. As shown in Figure 3, the D_READ and the FLUSH states determine the read operation during the debug mode and the flush operation during the normal mode, respectively, as listed in Lines 39–51[†]. The controller enters the FLUSH state when there is a flush command and if the cache is enabled. The intended behavior of the FSM is that the read operations in the debug mode are permitted only if the user

[†]For succinctness, we ignore the other states of the cache controller.

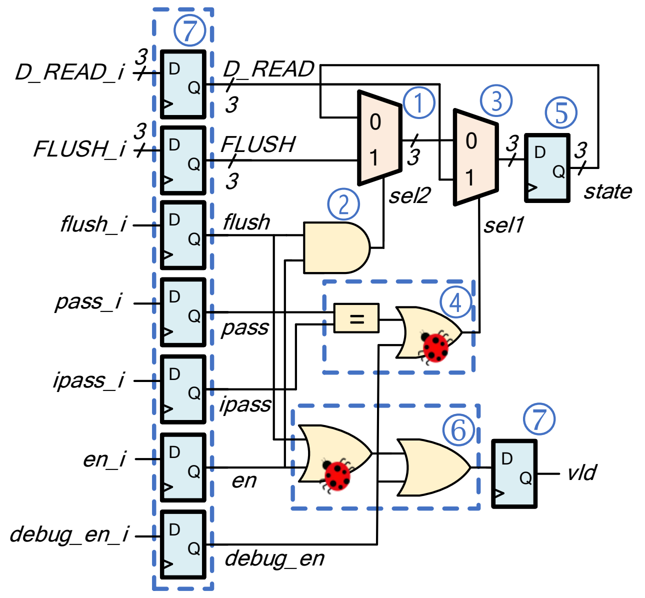

Figure 2: Hardware design for Listing 1.

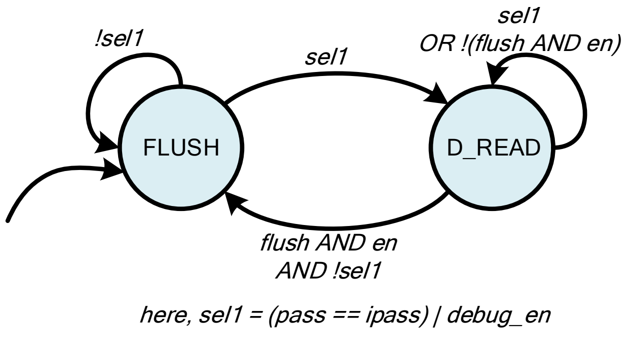

Figure 3: Finite state machine (FSM) of the design in Figure 2.

has inputted the correct password (Line 41). This protection mechanism allows only authorized users to read the cache in the debug mode. The cache controller sets the valid signal (`vld`) based on the flush and debug requests issued to the controller (Line 37 of Listing 1).

The electronic design automation (EDA) tools synthesize this RTL code into an equivalent gate-level design shown in Figure 2. The MUXes ① and ③ select the next state. The combinational logic ② and ④ controls the state transitions. The DFFs in ⑤ hold the current state. The EDA tools implement Line 37 as combinational logic ⑥. The DFFs in ⑦ register the inputs and outputs.

This design has two bugs: *b1* and *b2*. Bug *b1* (Line 41 in Listing 1) is from HardFails [14], which has been used for the Hack@DAC competitions, and is similar to CVE-2017-18293. This bug is in the combinational logic ④, where the debug read operation is access-protected but the bug allows one to perform the debug read operation illegally. This compromises the security of the read operations as it allows users without the correct password to read the cache. Bug *b2* is similar to CVE-2019-19602 and is in the combinational logic ⑥ that drives the `vld` register (Line 37), allowing one to flush the cache even when it is not enabled.

In ①, all the inputs of the MUX and their corresponding values on the select lines must be tested for correctness. For this purpose, we use **branch coverage**, which tests each branching construct (the `when` block of Line 45) for both "when" and "otherwise" conditions.

In ②, one should check that every input combination produces the correct output value. To this end, we use **condition coverage**, which requires the condition block (i.e., the condition `(flush & en)` in the `when` block of Line 45) to be tested for all possible input values and not only a subset of values.

In ⑤, the value of the 3-bit register can be one of the eight possible values. We use **FSM coverage** of the `state` register to check for all the eight values. This coverage captures the different FSM states and also their transitions.

In ⑥, all the input signals generating `vld` should be tested for all possible values, similar to ②. We use **expression coverage** for this purpose which requires the combinational block (i.e., the expression `debug_en|(flush|en)` in Line 37) to be tested for all possible input values. Furthermore, expression coverage covers the select line of MUX ③ and the combinational logic ④ that drives it as they are defined using an expression in Line 51, unlike MUX ① which is defined as branch in Lines 45–49.

In ⑤ and ⑦, the value of each DFF can be `1`, `0`, or floating[‡]. We use **toggle coverage** of these DFFs to check for toggling of their values among these three possibilities. Unlike FSM coverage, toggle coverage covers all the DFFs in the design. In addition, we also use **statement coverage** to ensure every line of the RTL code is executed during simulation.

TheHuzz uses commercial industrial-standard tools—Synopsys [74], ModelSim [68], Cadence [10]—to compile the hardware and extract these coverage values. The semiconductor industry has been using these tools for the last few decades, and its verification flow is built on these tools, thus providing a promising way to obtain coverage [50].

TheHuzz detects both *b1* and *b2* using the expression coverage of ④ and ⑥, respectively. The expression coverage verifies that all the signals involved in the combinational logics ④ and ⑥ cover all possible values. One such combination will trigger the bugs *b1* and *b2*, resulting in an incorrect output, which will be flagged as a mismatch. Thus, *TheHuzz*'s coverage metrics aid in detecting bugs *b1* and *b2*.

In contrast to *TheHuzz*, existing hardware fuzzers lose hardware intrinsic behaviors (e.g., floating wires, signal transitions) while converting the target hardware into a software model [79], operate only on the select signals of the MUXes [39], operate only on the DFFs that determine the

[‡]Referred to as a high-impedance state or tristate and denoted as z. Such floating wire-related bugs (CWE-1189) have compromised systems [14].

select signals of the MUXes [30], or operate at the protocol level [51]. Hence, the coverage used by existing fuzzers will not be able to cover the bugs in ④, ⑥, and some DFFs in ⑦ including the bugs we inserted, *b1* and *b2* (cf. Appendix B).

4.2 Seed Generator

Given that we have discussed the various coverage metrics to capture hardware behaviors, we now describe the seed generation in more detail. The seed generator generates *seed inputs* that run on the DUT and are used to generate further inputs through mutation.

Seed inputs. *TheHuzz*'s goal is to detect software-exploitable vulnerabilities in the RTL model of the processors. Processors execute instructions using the data from the instruction memory. Hence, our fuzzer provides inputs at the instruction set architecture (ISA) abstraction level by generating processor instructions. The *seed inputs* are data files containing a sequence of instructions, which are loaded onto the memory.

Instruction generator generates the instructions for the *seed inputs* from a set of valid instructions of the processor.

Input format. Each input consists of two types of instructions: *configuration instructions* (CIs) and *test instructions* (TIs). The CIs are needed to setup the baremetal environment, e.g., setting up the stack, exception handler table, and clearing the general-purpose registers. This baremetal environment allows *TheHuzz* to run instructions directly on the processor without the need for an operating system. The TIs are generated by the *instruction generator*, which are the actual instructions used to fuzz the processor.

4.3 Stimulus Generator

The stimulus generator is responsible for mutating the current inputs, generating new inputs, and discarding the underperforming inputs. Seed inputs are used to generate the first set of new inputs. We mutate the instructions directly as binary data instead of at a higher abstraction level such as assembly. This allows us to mutate all the bits of the instruction based on the mutation technique used. Thereby, we can test the processor with out-of-spec inputs like *illegal* instructions (i.e., instructions not specified in the ISA) generated through mutation of the *opcode bits* of the instruction. This allows us to detect issues that other verification techniques may not have detected, like the bug B3 in the Ariane and B8 in the or1200 processors, which cannot be detected with legal instructions.

Mutation engine performs the mutation operations on the instructions. We mutate only the TIs since these are the instructions used to fuzz the processor. The CIs are not mutated to ensure the correct initialization of the processor for fuzzing. The mutation techniques used by our fuzzer can be classified into two types. The first type only mutates the data bits keeping the opcode unchanged. These mutations increase the coverage on different data paths that are close to each other.

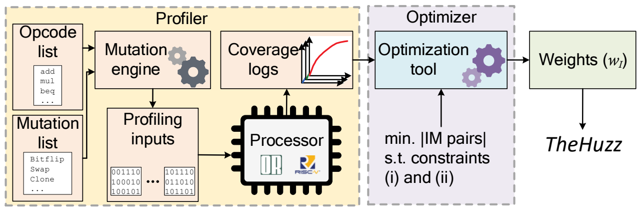

Figure 4: Optimization process used for *TheHuzz*.

To generate bug-triggering out-of-spec inputs, the second type of mutation techniques mutates both the data and the opcode bits. Mutating the opcode bits will create inputs with new instruction sequences and help uncover different control paths in the DUT. This will help generate illegal instructions to test the processor with out-of-spec inputs. We employ AFL-style mutation techniques as listed in Appendix A.

Every time new inputs are generated by the stimulus generator, the code coverage data of these inputs is used to discard the underperforming inputs, thereby only retaining the inputs that trigger new coverage points. This helps steer the fuzzer towards discovering new coverage points quickly.

4.4 Optimization

We now propose an optimization for improving the efficiency of a processor fuzzer, as shown in Figure 4. Instead of using all the instructions and mutations, we optimally select the ones that achieve the best coverage. To this end, we first profile the individual instructions and mutations and formulate an optimization problem, which returns the optimal weights for each instruction-mutation (IM) pair.

Profiler characterizes the control and data flow paths explored by each IM pair. *TheHuzz* generates the coverage values specific to each IM pair via hardware simulation.

Optimizer aims to minimize the number of IM pairs while achieving the same amount of coverage as using all the IM pairs. Let I and M be the sets of instructions and mutations, respectively. Let $\mathcal{P} = I \times M$. $\mathcal{C}$ denotes the union of coverage metrics such as statement, branch, expression, toggle, FSM, and condition. The coverage from the profiling phase for each IM pair is denoted by the indicator function $\mathcal{D} : \mathcal{P} \times \mathcal{C} \mapsto \{0,1\}$. $\mathcal{C}_d \subseteq \mathcal{C}$ denotes the coverage points hit by an IM pair during the profiling phase. The optimization problem is to find the smallest subset of $\mathcal{P}$, denoted as Q, that covers all the coverage points identified during the profiling stage, $\mathcal{C}_d$. The optimizer returns the set Q that contains the optimal IM pairs. *TheHuzz* uses this information to generate the weights for each instruction-mutation pair $w_{(I,M)}(i,m) = \mathbb{I}_{\{i,m\} \in Q}, \;\; \forall\ (i,m) \in P$, where $\mathbb{I}$ is an indicator function. The seed generator uses the weights, w_I, to select instructions, and the stimulus generator uses the weights, w_M to select the mutation techniques for each instruction and thereby, eliminating underperforming instructions and mutations.

4.5 Bug Detection

Software programs indicate bug triggers through crashes, memory leaks, and exit status codes. However, hardware intrinsically cannot provide such feedback because it does not crash or have memory leaks. Thus, as performed in traditional hardware verification, we compare the outputs of GRMs and the DUT for the inputs generated by the fuzzer. Any *mismatch* event indicates the presence of a bug, which is then manually analyzed to identify its cause.

5 Implementation

We implemented *TheHuzz* such that it is compatible with traditional IC design and verification flow, while effectively detecting security vulnerabilities. All the components are implemented in Python unless specified otherwise. We used CPLEX [31] for optimization.

Register-Transfer-Level simulation. We simulate the target hardware using a leading industry tool, Synopsys VCS [74]. This tool supports a wide variety of hardware description languages (HDLs) and different hardware models: RTL, gate level, and transistor level. We wrote custom Python scripts to process the logs of VCS to extract the coverage metrics—statement, branch, toggle, expression, and condition. It also generates instruction traces, which contain the sequence of instructions executed along with the register or memory locations modified by each instruction and their updated values. Thus, *TheHuzz* leverages existing hardware simulation tools to instrument the HDLs.

Seed generator generates C programs that consist of configuration instructions (CIs) and test instructions (TIs). The CIs configure a baremetal C environment on the processors; we extract these CIs from the baremetal libraries of the corresponding ISAs, e.g., the RISC-V tests repository [62]. The TIs are the actual instructions used to fuzz the processor from the initial state. Each seed input has 20 TIs; this number is selected based on empirical observations before a random TI leads to a deadlock. Events like exceptions or instructions like branch, jump, system calls, and atomic instructions can cause the control flow of the processor to jump to a different location or even freeze for a large number of clock cycles, waiting for resources (in the case of atomic instructions). The first half of the TIs are generated uniformly from the instructions that are less likely to trigger such events (e.g., arithmetic and logical instructions). This maximizes the number of TIs executed by the *TheHuzz* in each simulation. The other half of the TIs are generated uniformly from all the instructions returned by the optimizer. Thus, the processor is reset after the execution of every 20 instructions and is simulated with new input. This results in periodical initialization of the processor control flow back to the location of the TIs. The GCC toolchain compiles these C programs to generate the executable files which are loaded onto the processor RAM and used as seeds.

Stimulus generator consists of the mutation and the feedback engines. The mutation engine mutates the TIs using the AFL-like mutations listed in Appendix A. The feedback engine uses coverage logs for each mutated TI from the RTL simulation. It retains the best performing instruction-mutation pairs and discards the ones that do not improve the coverage.

Golden Reference Models (GRMs). We used *spike* ISA emulator [62] as the GRM for Ariane and Rocket Core, and *or1ksim* [57] as the GRM for mor1kx and or1200 processors.

6 Evaluation

We now describe the four open-source processors—Ariane, mor1kx, or1200, and Rocket Core—used to evaluate our fuzzer *TheHuzz* and present the evaluation results, along with bugs detected (cf. Table 1) and the coverage. We compare *TheHuzz* with another fuzzer DifuzzRTL [30] and two traditional hardware verification techniques: random regression testing and formal verification. The experiments are conducted on a 32-core Intel Xeon processor running at 2.6Ghz with 512GB of RAM with CentOS Linux release 7.3.1611.

6.1 Evaluation Setup

With rich hardware-software interactions and complex hardware components, processor designs provide a challenging target for evaluating the potential of hardware fuzzers. While testing commercial processors is appealing, their closed-source nature makes register-transfer level (RTL) analysis impossible. This is a challenge hardware researchers face, and hence, most papers which evaluate their tool's effectiveness on processors use open-source designs. We have selected four processors from two widely used open-source ISAs, OpenRISC [57] and RISC-V [63]. All these processors can run a modern Linux-based operating system.

Ariane (a.k.a. *cva6* core) is a RISC-V based, 64-bit, 6-stage, in-order processor, and supports a Unix-like operating system [91]. mor1kx is a 32-bit OpenRISC based processor. From the three possible configurations, we selected the 6-stage *Cappuccino* configuration, as it is the most complex design. Developers and the open-source community have evaluated this design for more than seven years. or1200 is a 32-bit OpenRISC based processor. It is one of the first open-source processors and is used for more than two decades [57]. Rocket Core is a RISC-V based, 64-bit, 5-stage, in-order scalar processor, and supports a Unix-like operating system [5]. RISC-V open-source processors are widely used in prior work in hardware verification and security, as shown in Table 1, and have proven to be effective replacements for commercial designs.

6.2 Bugs Detected

We now detail the vulnerabilities detected by *TheHuzz*. We found eight new bugs. We map each bug to the relevant hardware common weakness enumerations (CWEs), as listed in

Table 1: Bugs detected by *TheHuzz*.

Processor	Prior work using the design	Design size		Bug description	Location	Coverage types	CWE	New bug?	# instructions to detect the bug
		LOC	Coverage points						
Ariane [91] ISA: RISC-V [63] Design year: 2018 64-bit, 6-stage pipeline	[86, 69], [18, 14], [58]	2.07×10^4	3.42×10^5	**B1**: Incorrect implementation of logic to detect the FENCE.I instruction.	Decoder	Branch	CWE-440	✓	1.36×10^4
				B2: Incorrect propagation of exception type in instruction queue	Frontend	Toggle	CWE-1202	✗	4.02×10^4
				B3: Some *illegal* instructions can be executed	Decode	Condition	CWE-1242	✓	1.81×10^6
				B4: Failure to detect cache coherency violation	Cache controller	FSM	CWE-1202	✓	1.72×10^5
mor1kx [57] ISA: OpenRISC [57] Design year: 2013 32-bit, 6-stage pipeline	[15, 93], [38, 30]	2.21×10^4	4×10^4	**B5**: Incorrect implementation of the logic to generate the *carry* flag	ALU	Expression	CWE-1201	✓	20
				B6: Read/write access checking not implemented for privileged register	Register file	Condition	CWE-1262	✓	4.46×10^5
				B7: Incomplete implementation of EEAR register write logic	Register file	Condition	CWE-1199	✓	1.12×10^5
or1200 [57] ISA: OpenRISC [57] Design year: 2000 32-bit, 5-stage pipeline	[94, 24], [27, 25], [35, 88, 8]	3.16×10^4	3.90×10^4	**B8**: Incorrect forwarding logic for the GPR0	Register forwarding	Condition and expression	CWE-1281	✗	174
				B9: Incomplete update logic of overflow bit for MSB & MAC instructions	ALU	Toggle	CWE-1201	✓	3.35×10^3
				B10: Incorrect implementation of the logic to generate the *overflow* flag	ALU	Expression	CWE-1201	✓	2.21×10^4
Rocket Core [5] ISA: RISC-V [63] Design year:2016 32-bit, 5-stage pipeline	[30]	1.06×10^4	6.65×10^5	**B11**: Instruction retired count not increased when `EBREAK`	Register file	Condition	CWE-1201	✗	776

Listing 2: Verilog code snippet for **B1** in Ariane.

```verilog
 1 // Memory ordering instructions
 2 riscv::OpcodeMiscMem: begin
 3   instruction_o.fu =CSR;
 4   instruction_o.rs1 ='0;
 5   instruction_o.rs2 ='0;
 6   instruction_o.rd ='0;
 7   case (instr.stype.funct3)
 8     // FENCE: Currently implemented as a whole DCache flush
           ↪ boldly ignoring other things
 9     3'b000: instruction_o.op =ariane_pkg::FENCE;
10     // FENCE.I
11     3'b001: begin
12       if (instr.instr[31:20] !='0)
13         illegal_instr =1'b1;
14       instruction_o.op =ariane_pkg::FENCE_I;
15     end
16     default: illegal_instr =1'b1;
17   endcase
18   if (instr.stype.rs1 !='0 ||instr.stype.imm0 !='0 ||instr.
         ↪ instr[31:28] !='0)
19     illegal_instr =1'b1;
20 end
```

Table 1. We present bugs **B1**, **B4**, and **B6** in detail as we exploit them in Section 6.3 and briefly describe the other bugs; arXiv version [80] details the other bugs.

6.2.1 Bugs in Ariane Processor

Bug B1 is located in the decode stage of Ariane. According to the RISC-V specification [63], the decoder should ignore certain fields in a *FENCE.I* instruction, which enforces cache coherence in the processor (e.g., by flushing the instruction cache and instruction pipeline). It also ensures that the correct instruction memory is used for execution when performing memory sensitive operations (e.g., updating the instruction memory). The bug is that the decoder does not ignore the *imm* and *rs1* fields and expects a value of *0* in these fields, as seen in Lines 12 and 18 of Listing 2. This Ariane implementation declares valid instructions as illegal (Lines 13 and 19) due to this additional constraint on the *imm* and *rs1* fields, thus violating the specification. We detected this bug when the fuzzer generated a *FENCE.I* instruction with a non-zero value in the *imm* field. Ariane raised an exception saying that the instruction is *illegal*, whereas *spike* successfully executed the instruction, resulting in a mismatch[§]. Due to this bug, *failing-FENCE.I* will not be executed, resulting in a potential violation of cache coherence. This bug is similar to the expected behavior violation vulnerability, CWE-440 [48].

Bug B2 is in the instruction queue of the frontend stage of Ariane. The bug is that a fixed exception is forwarded instead of the actual exception. We detected this bug as a mismatch in the value of a register that loads the exception type when an exception occurs. Operating systems that assume that instruction access-faults are raised correctly will not behave as expected, and triggering this bug may lead to undefined (and possibly exploitable) behavior. Also, an incorrect exception handling might be executed, resulting in a memory and storage vulnerability, CWE-1202 [48].

Bug B3 is that the decode stage does not correctly check for certain illegal instructions. It was detected as a mismatch when the fuzzer generated one such illegal instruction. Due to this, any undocumented instruction of a certain value can be executed on Ariane, resulting in an undocumented feature vulnerability, CWE-1242 [48].

Bug B4 As per the RISC-V specification [63], when the instruction memory is modified, the software should handle cache coherency using *FENCE.I* instruction. Failure to handle cache coherency results in undefined behavior, wherein processors may use stale data and incorrect execution of instructions [71]. When the fuzzer generated an input program that modified the instruction memory but did not use a *FENCE.I* instruction, *TheHuzz* detected a mismatch in the trace logs of Ariane and *spike*. This mismatch could have been avoided

[§]We refer to these *FENCE.I* instructions that Ariane fails to detect as *failing-FENCE.I* and the rest as the *working-FENCE.I*.

if the RISC-V specification or the Ariane processor detected violations of cache coherency in hardware. Due to this bug, software running on Ariane could run into cache coherency issues and remain undetected if the *FENCE.I* instruction is used incorrectly, resulting in a memory and storage vulnerability, CWE-1202 [48]. In Section 6.3.1 we use this bug and bug **B1** to successfully exploit a theoretically safe program.

6.2.2 Bugs in mor1kx Processor

Bug B5 is the inaccurate implementation of the *carry* flag logic for subtract operations. The fuzzer generated inputs that triggered this bug by mutating the data bits of subtract instructions. This caused a mismatch in the value of the *carry* flag between the RTL and golden reference model (GRM). This bug can cause incorrect computations, including those used in cryptographic functions, resulting in corruption and compromise of the processor security (CWE-1201 [48]).

Bug B6 The register file stores, updates, and shares the value of all the architectural registers. These registers include the general- and special-purpose registers (GPRs and SPRs, respectively). *Read* and *write* operations to the SPRs are restricted based on the privilege mode of the processor, as per the OpenRISC specification [57]. The Exception Program Counter Register (`EPCR`) is an SPR that stores the address to which the processor should return after handling an exception. A user-level program should not be able to access this register. The bug in mor1kx is that the register file does not check for privilege mode access permissions when performing *read* and *write* operations on `EPCR`. This bug was detected when our fuzzer generated an instruction that tried to write into `EPCR` from user privilege mode. Due to this bug, an attacker can write into `EPCR` from user privilege mode and control the return address of the processor after handling an exception (CWE-1262 [48]). This bug can have severe security consequences like privilege escalation, as demonstrated in our mor1kx exploit in Section 6.3.2.

Bug B7. The register file in mor1kx does not allow one to write into the Exception Effective Address Register (`EEAR`), even for supervisor privilege mode. This bug is detected when our fuzzer generated an instruction that tried to write into `EEAR` from the supervisor privilege mode. This bug prevents programs from updating `EEAR`, resulting in incorrect executions. Thus, it prevents software from correctly performing exception handling. This bug is similar to CWE-1199 [48].

6.2.3 Bugs in or1200 Processor

Bug B8 is that the register forwarding logic forwards a non-zero value for `GPR0` if a previous instruction in the pipeline writes to `GPR0`. We found this bug as a mismatch when the fuzzer applied an ADD instruction to create a data hazard for `GPR0`. This bug can result in incorrect computations since `GPR0` is frequently used by software to check for conditions. An attacker can cause data hazards to obfuscate the behavior of malware, e.g., by jumping to an offset computed by an instruction that uses `GPR0`. This bug is similar to CWE-1281 [48], where a sequence of processor instructions result in unexpected behavior.

Bug B9 is that the *overflow* flag is not correctly calculated for the multiply and subtract (MSB) and the multiply and accumulate (MAC) instructions. This bug results in the failure of the software programs to detect the *overflow* events. Thus this bug is a core and compute issue vulnerability, CWE-1201 [48], resulting in more software vulnerabilities.

Bug B10 is the incorrect overflow logic for the subtract instruction. The bug was detected when the fuzzer was mutating data bits of subtract instruction. This bug also compromises the security mechanisms relying on the *overflow* flag and is a core and compute issue vulnerability, CWE-1201 [48].

6.2.4 Bugs in Rocket Core Processor

Bug B11 is that the instruction retired count does not increase on an `EBREAK` instruction. It was detected when the fuzzer executed the `EBREAK` instruction. *TheHuzz* was able to detect the only bug, **B11** reported by DifuzzRTL using only 776 instructions and is 6.7× faster than DifuzzRTL.

All the bugs except for **B2**, **B8**, and **B11** are new bugs detected by *TheHuzz*. **B2** is fixed in the latest version of Ariane. **B8** is first reported in [93].

6.3 Case Study: Exploitability

We now present the two exploits we crafted to demonstrate the security implications of the bugs found by *TheHuzz*. Both attacks can be mounted from unprivileged software.

6.3.1 Ariane FENCE.I Exploit

The Ariane exploit leverages **B1** and **B4** to cause incoherence in the instruction cache. As a result, in the contrived "safe" just-in-time (JIT) compiler we developed to demonstrate this bug, an attacker can generate inputs that selectively invalidate cache lines containing old instructions. This program uses an extension of the *FENCE.I* instruction (from the *failing-FENCE.I* instructions) which should fall back to standard fence behavior and flush the entire instruction cache as the extension is not understood by *spike* or Ariane. For our threat model, we assume that the attacker is aware of the use of an extended FENCE.I instruction present in a target and is capable of loading and executing "safe" programs in the target's JIT compiler. An attacker first loads a region of executable code (which does not contain a vulnerability) into the cache by executing it. The attacker then overwrites the same region of executable code with new instructions (which also does not contain a vulnerability), then executes separate code which jumps to instructions which align to cache lines the attacker

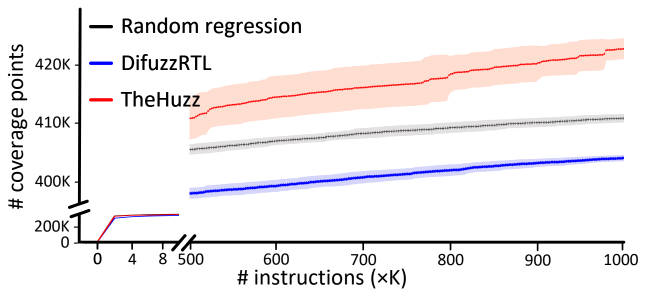

Figure 5: Coverage analysis of random regression testing, DifuzzRTL [30], and *TheHuzz* for the Rocket Core processor.

wishes to invalidate. After, they execute the original region of executable code, at which point the behavior of *spike* and Ariane diverge. In *spike*, the new instructions will be present and will execute as expected with no vulnerabilities present. This is because *spike* successfully identified the *FENCE.I* instruction, but did not recognise its extension, and fell back to flushing the entire cache. In Ariane, the old instructions will be present; Ariane fails to recognise the *FENCE.I* instruction as it instead marks it as an illegal instruction, an implementation which is non-compliant with the RISC-V ISA. Because the cache lines were only invalidated in regions selected by the attacker, the attacker is able to successfully replace bounds checks in the original program with effectively nops, leading to a vulnerability which was neither present in the old or the new JIT code. As a result, the attacker is able to inject a stack overflow vulnerability and gain arbitrary code execution. A more detailed description of the vulnerability, exploit, ramifications, and threat model are presented in the arXiv version [80].

6.3.2 mor1kx EPCR Register Exploit

The mor1kx exploit leverages the **B6** to set the `EPCR` to point to an attacker-controlled exploit function. An exception return instruction is executed to mimic the return from an exception event, causing the processor to update the program counter (`PC`) and status register (`SR`) values with `EPCR` and exception status register (`ESR`) values, respectively. The `SR` stores the privilege level. By performing the exploit when the `ESR` stores a higher-privilege level, execution jumps to the exploit function while overwriting the privilege level stored in `SR`. For our threat model, we assume that the attacker already has "foothold" access to a target machine and has the ability to execute arbitrary instructions as a low-privilege user. In this scenario, an attacker can perform privilege escalation in the mor1kx processor. The arXiv version [80] explains this exploit in detail.

6.4 Coverage Analysis

Figure 5 shows the coverage achieved by random regression testing, DifuzzRTL, and *TheHuzz* for the Rocket Core processor. Each experiment is repeated 10 times. Even after 1M instructions, both random regression testing and DifuzzRTL did not improve their coverage beyond 2.5% than what they collected after applying 300K instructions; on the other hand *TheHuzz*'s coverage kept increasing. *TheHuzz* is slower in the beginning than random regression testing as the fuzzer uses a set of instructions until it cannot reach new coverage points; in that case, it discards and selects new a set of instructions. *TheHuzz* achieved the 404.1K coverage points achieved by DifuzzRTL at 3.33× the speed of DifuzzRTL. *TheHuzz* and random regression testing outperformed DifuzzRTL because DifuzzRTL is guided by the control-register coverage, which does not capture many hardware behaviors (cf. Appendix B). The p-value from the Mann-Whitney U test [53] shows that the result is statistically significant ($p < 0.05$) with a p-value of 1.4e-4 for both random regression testing and DifuzzRTL. The Vargha-Delaney A12 measure returned *TheHuzz* as the best performing technique when compared with random regression testing and DifuzzRTL.

The instrumentation overhead of DifuzzRTL is 18% in terms of lines of Verilog code. *TheHuzz* does not instrument Verilog code explicitly and instead relies on the commercial tools which do not produce the overhead information. Hence, the instrumentation overheads of these two fuzzers are not comparable. The runtime overhead for *TheHuzz* (71%) is greater than DifuzzRTL (6.9%) since *TheHuzz* requires accessing multiple files to collect all the coverage, whereas DifuzzRTL only needs to collect control-register coverage.

6.5 Comparison with Formal Verification

We also compared our fuzzer with another standard approach used by the semiconductor industry—formal verification. For this purpose, we used the industry-leading formal verification tool, Cadence JasperGold [10]. However, there are two challenges in performing this comparison. First, there is no industry-standard formal tool that can produce a set of instructions that can trigger a hardware bug in RTL, even if the bug is known apriori. Second, these industry tools require one to write assertions targeting each vulnerability manually. Thus, the usage of formal tools in this scenario requires one to know of these vulnerabilities apriori—unlike *TheHuzz*, which does not make any such assumptions.

To manually write these assertions, one has to know the entire design, identify the signals and specific conditions that trigger the security vulnerability. This step is highly cumbersome given the vast number of modules, signals, and states in processors, as shown in Table 2. Many bugs are cross-modular, and hence, they require one to load multiple modules, which only makes writing assertions difficult as they now need to consider signals across modules and their interactions. These tools only produce Boolean assignments to the inputs of these modules and not a set of instructions that violate these assertions. As shown Table 2, the number of inputs range in few

Table 2: Hardware complexity encountered while using industry-standard JasperGold [10] to detect the bugs.

Processor	Ariane				mor1kx			or1200			Rocket Core
Bug / Statistics	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**	**B11**
No. of modules	1	9	1	655	1	4	4	6	1	1	1
No. of inputs	518	627	518	3	298	752	752	703	123	123	284
No. of states	2.51e+58	2.16e+68	2.16e+68	2.01e+59	4.72e+10	1.55e+11	1.55e+11	3.83e+11	1.29e+10	1.29e+10	2.23e+20

hundreds, thereby increasing the number of states that need to be checked, leading to state-space explosion. Some bugs like **B4** require one to load the entire system-on-chip into the formal tool, which is not always feasible due to state-space explosion. Thus, in contrast to *TheHuzz*, existing formal tools are resource intense, error-prone, and not scalable to complex bugs and larger designs, apart from relying heavily on human expertise and prior knowledge of hardware vulnerabilities.

7 Related Work

We now describe the limitations of the existing attempts to fuzz hardware and how *TheHuzz* is different from them, as summarized in Table 3.

RFUZZ is a mux-coverage-guided fuzzer for hardware designs [39]. Although this technique can fuzz designs on FPGAs, it is computationally intensive and does not scale to large designs [30]. Additionally, its coverage metric does not capture many hardware behaviors (cf. Appendix B). It is also ineffective in finding any bugs.

HyperFuzzing proposes a new grammar to represent the security specification rules for hardware, converts the hardware design into equivalent software models, and fuzzes them using AFL fuzzer [51]. It is inapplicable to general hardware designs like finite state machines (FSMs) or combinational logic and requires a lot of human intervention, including writing security specifications manually. It did not report any bugs.

Fuzzing hardware like software translates the hardware design to software models and fuzzes them using a software fuzzer [79]. While this is a promising approach, it is limited by the strength of existing open-source tools (i.e., Verilator [70]): they currently do not support many constructs of HDLs such as latches, floating wires, etc. It did not report any bugs. The largest benchmark used by this technique has 4,585 lines of code (LOC). It also does not scale to real-world designs like processors. For instance, while fuzzing Google's OpenTitan SoC [21], this work could only fuzz the peripheral modules but not the *iBex* processor in it.

DifuzzRTL, a recent work, uses a custom-developed control-register coverage as feedback for the fuzzer by instrumenting the HDL [30]. The technique only focuses on the coverage of registers generating the select signals of MUXes and does not check for toggle, expression, and FSM coverage points, thereby missing the bugs in ③, ④, and floating wires in ⑤ in Figure 2 (cf. Appendix B for more details). None of the bugs found by this fuzzer are shown to be exploitable, as most bugs are triggered by physically controlling the interrupt signals with precise timing; such interrupt signals are not usually exposed to unprivileged software [56]. The fuzzer is also slower in detecting the bugs as it compares the processor state after the entire program is executed, while our fuzzer performs comparison after each instruction is executed.

In contrast, ***TheHuzz***: (i) is compatible with traditional IC design verification flow allowing for seamless integration by using coverage metrics already widely used in the semiconductor industry; (ii) is scalable to large, complicated, industrial-designs with several tens of thousands of code, and not just small FSM designs; (iii) captures many intrinsic hardware behaviors, such as signal transitions and floating wires, using multiple coverage metrics: statement, toggle, branch, expression, condition, and FSM; (iv) does not require the designer to specify security rules; and (v) detects several bugs that lead to severe security exploits. Instead, we compare how the software views the hardware (i.e., ISA emulator) and how the hardware actually behaves (i.e., Verilog), leading to an effective hardware fuzzer.

8 Discussion and Limitations

Requirement of Golden Reference Models (GRMs). *TheHuzz* and other hardware fuzzers [30, 51] depend on GRMs to find vulnerabilities. Such GRMs are widely available in the semiconductor industry. Verification of many commercial (proprietary and open-source) CPUs critically depend on the availability of GRMs, including many industrial, large-scale designs, e.g. Intel x86 Archsim [33], AMD x86 Simnow [1], ARM Cortex Neoverse [3], and ARM Fast Models [4]. Thus, the reliance on GRM is not a limiting factor for *TheHuzz*. Sometimes, the GRM itself can be buggy, thereby causing false positives. This situation is highly unlikely because GRMs are carefully curated and versioned with legacy code, and rigorously tested. Verifying a GRM is easier as it is written at a higher abstraction level and is thus less complex than a RTL model.

Requirement of Register-Transfer Level (RTL) source code. *TheHuzz* depends on RTL access, similar to previous works such as DifuzzRTL [30], RFUZZ [39], and Hyperfuzzing [51]. As mentioned in Section 2.2, verification teams already have access to RTL. An attacker can also buy RTL models of the target design, as many companies like Imagi-

Table 3: Comparison with the prior work on hardware fuzzers.

Methodology	Fuzzer used	HDL	Simulator	Target design	Design knowledge	Largest design (Lines of code)	Metrics used	Comparison against random regression testing	Bugs reported	Exploitable from software	Exploits presented
RFUZZ [39]	H/W fuzzer	FIRRTL	Any	RTL designs	Not required	5-stage Sodor core (4,088)	mux-coverage	~5% increase in mux coverage	0	N/A	0
Hyperfuzzing [51]	S/W AFL fuzzer	Any	Verilator	SoC designs	Need security rules	SHA crypto engine (1,196)	None	N/A	0	N/A	0
Trippel et al. [79]	S/W AFL fuzzer	Any	Verilator	RTL designs	Not required	KMAC (4,585)	FSM , line, edge, toggle, and functional coverage	Two orders magnitude faster for datapath FSMs	0	N/A	0
DifuzzRTL [30]	H/W fuzzer	Any	Any	CPU designs	Not required	Boom (12,956 in Scala)	Control-register coverage	~10% increase in control-register coverage	16	Not reported	0
TheHuzz	H/W fuzzer	Any	Commercial, industry-standard HDL simulator	CPU designs	Not required	Ariane (20,698)	statement, toggle, branch, expression, condition, and FSM coverage	2.86% increase in coverage metrics	10	Yes	2*

*In theory, the bugs discovered can be used to build more than two exploits, but we show only two due to page limitations.

nation Tech. Limited [32], Cadence [9], and Synopsys [73] sell proprietary hardware designs, and run *TheHuzz* on them as these designs are compatible with industry-standard tools. While companies like Intel and ARM do not reveal the RTL model of their processors, attackers can use reverse engineering services from companies like TechInsights [77] on the target chip and use gate-level to RTL reverse engineering techniques [72] to obtain the RTL model.

FPGA emulations. DifuzzRTL and RFUZZ can fuzz processors faster through FPGA emulation than RTL simulations [30, 39]. *TheHuzz* uses the coverage metrics implemented by EDA simulation tools like Modelsim [68] and Synopsys VCS [74]. These coverage metrics are not readily available for FPGA emulations, thereby limiting *TheHuzz*'s applicability to fuzz FPGA-emulated designs.

Fuzzing non-processor designs. Currently, *TheHuzz*, similar to DifuzzRTL [30], is limited to fuzzing processor designs since it generates processor specific inputs. These fuzzers cannot fuzz standalone hardware components like SoC peripherals, memory modules, and other hardware accelerators, which are targeted by RFUZZ and Tripple et al. [79]. *TheHuzz* could be extended to fuzz non-processor designs by fuzzing the individual input signals of the design. The seeds would be assignments to individual input signal values rather than instructions. The coverage metrics and the bug detection mechanism used by *TheHuzz* will still be applicable.

Fuzzing parametric properties of hardware. *TheHuzz* currently fuzzes only processors for functional behavior but not for parametric behavior (e.g., cache timing behavior) and thereby cannot detect side-channel vulnerabilities. One can extend *TheHuzz* to cover such vulnerabilities by developing timing-related coverage properties and targeting them.

9 Conclusion

Bugs in hardware are increasingly exposed and exploited. Current techniques fall short of detecting bugs, as our results demonstrated by finding bugs in a 20-year old processor and others. This calls for a revamp of security evaluation methodologies for hardware designs.

We presented an instruction fuzzer, *TheHuzz*, for processor-based hardware designs. The effectiveness of *TheHuzz* is shown by fuzzing three popular open-sourced processor designs. *TheHuzz* has detected eight new bugs in the three designs tested and three previously detected bugs. These bugs, when used individually or in tandem, resulted in ROP and privilege escalation exploits that could compromise both hardware and software, as shown in the two exploits we presented. Our fuzzer achieved 1.98× and 3.33× the speed compared to the industry-standard random regression approach and the state-of-the-art hardware fuzzer, DifuzzRTL, respectively. Finally, compared to the industry-standard formal verification tool, JasperGold, *TheHuzz* does not need human intervention and overcomes its other limitations.

Responsible disclosure. The bugs have been responsibly disclosed through the legal department of our institution(s).

Acknowledgement

Our research work was partially funded by the US Office of Naval Research (ONR Award #N00014-18-1-2058), by Intel's Scalable Assurance Program, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—SFB 1119—236615297 within project S2, and by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within ATHENE. We thank Kevin Laeufer (UC Berkeley), Jaewon Hur (Seoul National University), and TAMU HRPC for their support. And, we thank anonymous reviewers for their comments. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] AMD. AMD SimNow. https://developer.amd.com/simnow-simulator, 2021. Last accessed on 10/09/2021.

[2] A. Ardeshiricham, W. Hu, et al. Clepsydra: Modeling Timing Flows in Hardware Designs. *IEEE/ACM ICCAD*, pages 147–154, 2017.

[3] ARM. ARM Cortex Neoverse. https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-n1, 2021. Last accessed on 10/09/2021.

[4] ARM. ARM Fast Models. https://developer.arm.com/tools-and-software/simulation-models/fast-models, 2021. Last accessed on 10/09/2021.

[5] K. Asanovic, R. Avizienis, et al. The Rocket Chip Generator. *EECS Department, UCB, Tech. Rep.*, 2016.

[6] Averant. Averant Solidify. http://www.averant.com/storage/documents/Solidify.pdf, 2015. Last accessed on 04/08/2021.

[7] W. Badawy and G. A. Julien. *System-on-chip for Real-time Applications*, volume 711. Springer Science & Business Media, 2012.

[8] J. Bai, L. Wu, et al. A 10Gbps In-line Network Security Processor With a 32-bit Embedded CPU. *IEEE WOCC*, pages 616–619, 2013.

[9] Cadence. Cadence Design IP Portfolio. https://ip.cadence.com/ipportfolio/ip-portfolio-overview, 2021. Last accessed on 10/09/2021.

[10] Cadence. Cadence Webpage. https://www.cadence.com/en_US/home.html, 2021. Last accessed on 04/08/2021.

[11] G. Chen, S. Chen, et al. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. *IEEE S&P*, pages 142–157, 2019.

[12] W. Chen, S. Ray, et al. Challenges and Trends in Modern SoC Design Verification. *IEEE D&T*, 34(5):7–22, 2017.

[13] E. M. Clarke, W. Klieber, et al. Model Checking and the State Explosion Problem. *LASER Summer School on Software Engineering*, pages 1–30, 2011.

[14] G. Dessouky, D. Gens, et al. Hardfails: Insights into Software-Exploitable Hardware Bugs. *USENIX Security Symposium*, pages 213–230, 2019.

[15] C. Deutschbein and C. Sturton. Mining Security Critical Linear Temporal Logic Specifications for Processors. *IEEE MTV*, pages 18–23, 2018.

[16] L. Dukes, X. Yuan, et al. A Case Study on Web Application Security Testing with Tools and Manual Testing. *IEEE Southeastcon*, pages 1–6, 2013.

[17] F. Farahmandi, Y. Huang, et al. *System-on-Chip Security: Validation and Verification*. Springer Nature, 2019.

[18] M. Fischer, F. Langer, et al. Hardware Penetration Testing Knocks Your SoCs Off. *IEEE D&T*, 2020.

[19] S. Gan, C. Zhang, et al. CollAFL: Path Sensitive Fuzzing. *IEEE S&P*, pages 679–696, 2018.

[20] Google. ClusterFuzz. https://google.github.io/clusterfuzz/, 2021. Last accessed on 04/08/2021.

[21] Google. Opentitan SoC. https://opentitan.org/, 2021. Last accessed on 04/08/2021.

[22] Google. Syzkaller. https://github.com/google/syzkaller, 2021. Last accessed on 04/08/2021.

[23] S. Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. https://saelo.github.io/papers/thesis.pdf. Last accessed on 04/08/2021.

[24] S. Gurumurthy, S. Vasudevan, et al. Automatic Generation of Instruction Sequences Targeting Hard-to-detect Structural Faults in a Processor. *IEEE ITC*, pages 1–9, 2006.

[25] S. Gurumurthy, R. Vemu, et al. Automatic Generation of Instructions to Robustly Test Delay Defects in Processors. *IEEE ETS*, pages 173–178, 2007.

[26] S. L. He, N. H. Roe, et al. Model of the Product Development Lifecycle. *Sandia Report (2015)*, pages 1–49, 2015.

[27] M. Hicks, C. Sturton, et al. Specs: A Lightweight Runtime Mechanism for Protecting Software From Security-critical Processor Bugs. *ACM ASPLOS*, pages 517–529, 2015.

[28] W. E. Howden. Theoretical and Empirical Studies of Program Testing. *IEEE TSE*, SE-4(4):293–298, 1978.

[29] C.-C. Hsu, C.-Y. Wu, et al. Instrim: Lightweight Instrumentation for Coverage-guided Fuzzing. *NDSS, Workshop on Binary Analysis Research*, 2018.

[30] J. Hur, S. Song, et al. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. *IEEE S&P*, pages 1286–1303, 2021.

[31] IBM. CPLEX. https://pypi.org/project/cplex/, 2021. Last accessed on 04/08/2021.

[32] Imagination. Imagination Technologies. https://www.imaginationtech.com/products, 2021. Last accessed on 10/08/2021.

[33] Intel. Intel Archsim. https://course.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=marr_hyperthread02.pdf, 2021. Last accessed on 10/09/2021.

[34] Z. Kenjar, T. Frassetto, et al. V0LTpwn: Attacking x86 Processor Integrity from Software. *USENIX Security Symposium*, pages 1445–1461, 2020.

[35] A. R. Khatri. Implementation, Verification and Validation of an OpenRISC-1200 Soft-core Processor on FPGA. *IJACSA*, 10(1):480–487, 2019.

[36] P. Kocher, J. Horn, et al. Spectre Attacks: Exploiting Speculative Execution. *IEEE S&P*, pages 1–19, 2019.

[37] D. Koncaliev. Pentium FDIV bug. https://www.cs.earlham.edu/~dusko/cs63/fdiv.html, 2001. Last accessed on 04/08/2021.

[38] G. Krishnakumar and C. Rebeiro. MSMPX: Microarchitectural Extensions for Meltdown Safe Memory Protection. *IEEE SOCC*, pages 432–437, 2019.

[39] K. Laeufer, J. Koenig, et al. RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs. *IEEE/ACM ICCAD*, pages 1–8, 2018.

[40] lcamtuf. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Last accessed on 04/08/2021.

[41] J. Li, B. Zhao, et al. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[42] X. Li, V. Kashyap, et al. Sapper: A Language for Hardware-level Security Policy Enforcement. *ACM ASPLOS*, pages 97–112, 2014.

[43] X. Li, M. Tiwari, et al. Caisson: A Hardware Description Language for Secure Information Flow. *ACM PLDI*, 46(6):109–120, 2011.

[44] Y. Li, B. Chen, et al. Steelix: Program-State Based Binary Fuzzing. *ESEC/FSE*, pages 627–637, 2017.

[45] M. Lipp, M. Schwarz, et al. Meltdown: Reading Kernel Memory from User Space. *USENIX Security Symposium*, pages 973–990, 2018.

[46] V. J. M. Manès, H. Han, et al. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE TSE*, pages 1–1, 2019.

[47] B. Marshall. Hardware Verification in an Open Source Context. *ODSA*, 2019.

[48] MITRE. Hardware Design CWEs. https://cwe.mitre.org/data/definitions/1194.html, 2019. Last accessed on 04/08/2021.

[49] MITRE. CVE Database. https://cveform.mitre.org/, 2021. Last accessed on 04/08/2021.

[50] A. Molina and O. Cadenas. Functional Verification: Approaches and Challenges. *Latin American applied research*, 37(1):65–69, 2007.

[51] S. K. Muduli, G. Takhar, et al. Hyperfuzzing for SoC Security Validation. *IEEE/ACM ICCAD*, pages 1–9, 2020.

[52] O. Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. *IEEE DATE*, pages 1116–1121, 2017.

[53] W. C. Navidi. *Statistics for engineers and scientists*. McGraw-Hill Higher Education New York, NY, USA, 2008. Last accessed on 04/08/2021.

[54] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007.

[55] Onespin. Onespin Website. https://www.onespin.com/, 2021. Last accessed on 04/08/2021.

[56] OpenHW Group. Ariane Source Code. https://github.com/lowRISC/ariane, 2020. Last accessed on 04/08/2021.

[57] OpenRISC. OpenRISC Homepage. https://openrisc.io/, 2020. Last accessed on 04/08/2021.

[58] Princeton. OpenPiton. https://parallel.princeton.edu/openpiton/index.html, 2018. Last accessed on 04/08/2021.

[59] R. Qiao and M. Seaborn. A New Approach for Rowhammer Attacks. *IEEE HOST*, pages 161–166, 2016.

[60] P. Qiu, D. Wang, et al. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. *ACM CCS*, pages 195–209, 2019.

[61] J. Rajendran, V. Vedula, et al. Detecting Malicious Modifications of Data in Third-Party Intellectual Property cores. *IEEE/ACM DAC*, pages 1–6, 2015.

[62] RISC-V. RISC-V Github Repositories. https://github.com/riscv, 2021. Last accessed on 04/08/2021.

[63] RISC-V. RISC-V Webpage. https://riscv.org/, 2021. Last accessed on 04/08/2021.

[64] S. R. Sarangi, A. Tiwari, et al. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. *IEEE/ACM MICRO*, pages 26–37, 2006.

[65] S. Schumilo, C. Aschermann, et al. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. *USENIX Security Symposium*, pages 167–182, August 2017.

[66] K. Serebryany, D. Bruening, et al. AddressSanitizer: A Fast Address Sanity Checker. *USENIX ATC*, page 28, 2012.

[67] K. Serebryany. OSS-Fuzz - Google's Continuous Fuzzing Service for Open Source Software. *USENIX Association*, August 2017.

[68] Siemens. Modelsim. https://eda.sw.siemens.com/en-US/ic/modelsim/, 2021. Last accessed on 04/08/2021.

[69] D. Šišejković, F. Merchant, et al. A Secure Hardware-software Solution Based on RISC-V, Logic Locking and Microkernel. *SCOPES*, pages 62–65, 2020.

[70] W. Snyder. Verilator. https://www.veripool.org/wiki/verilator, 2021. Last accessed on 04/08/2021.

[71] D. J. Sorin, M. D. Hill, et al. A Primer on Memory Consistency and Cache Coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

[72] P. Subramanyan, N. Tsiskaridze, et al. Reverse Engineering Digital Circuits Using Structural and Functional Analyses. *IEEE Transactions on Emerging Topics in Computing*, 2(1):63–80, 2013.

[73] Synopsys. Synopsys DesignWare IP. https://www.synopsys.com/designware-ip.html, 2021. Last accessed on 10/08/2021.

[74] Synopsys. Synopsys Webpage. https://www.synopsys.com/, 2021. Last accessed on 04/08/2021.

[75] A. Takanen, J. D. Demott, et al. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[76] A. Tang, S. Sethumadhavan, et al. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. *USENIX Security Symposium*, pages 1057–1074, 2017.

[77] Techinsights. TechInsights. https://www.techinsights.com/, 2021. Last accessed on 10/08/2021.

[78] M. Tiwari, J. K. Oberg, et al. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. *ACM/IEEE ISCA*, 39(3):189–200, 2011.

[79] T. Trippel, K. G. Shin, et al. Fuzzing Hardware Like Software. *arXiv preprint arXiv:2102.02308*, 2021.

[80] A. Tyagi, A. Crump, et al. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *arXiv preprint arXiv:2201.09941*, 2022.

[81] J. Van Bulck, F. Piessens, et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, pages 991–1008, 2018.

[82] V. Van Der Veen, Y. Fratantonio, et al. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. *ACM CCS*, pages 1675–1689, 2016.

[83] S. Vasudevan. An Introduction to IC Verification. *Effective Functional Verification: Principles and Processes*, pages 3–12, 2006.

[84] I. Wagner and V. Bertacco. Engineering Trust with Semantic Guardians. *IEEE DATE*, pages 1–6, 2007.

[85] B. Wile, J. Goss, et al. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.

[86] N. Wistoff, M. Schneider, et al. Prevention of Microarchitectural Covert Channels on an Open-source 64-bit RISC-V Core. *arXiv preprint arXiv:2005.02193*, 2020.

[87] W. Xu, S. Park, et al. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. *ACM CCS*, page 971–986, 2020.

[88] S. Xuan, J. Han, et al. A Configurable SoC Design for Information Security. *IEEE ASICON*, pages 1–4, 2015.

[89] W. Yang, M.-K. Chung, et al. Current Status and Challenges of SoC Verification for Embedded Systems Market. *IEEE SOCC*, pages 213–216, 2003.

[90] M. Zalewski. Technical Whitepaper for AFL Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2015. Last accessed on 04/08/2021.

[91] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[92] D. Zhang, Y. Wang, et al. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *ACM ASPLOS*, pages 503–516, 2015.

[93] R. Zhang, C. Deutschbein, et al. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. *IEEE/ACM MICRO*, pages 815–827, 2018.

[94] R. Zhang, N. Stanley, et al. Identifying Security Critical Properties for the Dynamic Verification of a Processor. *ACM SIGARCH Computer Architecture News*, 45(1):541–554, 2017.

Appendix

A Mutation Techniques

We use 12 distinct mutation techniques inspired by the popular binary manipulation fuzzer, AFL [90], as indicated in Table 4 and also detailed in the arXiv version [80].

Table 4: Mutation techniques used by *TheHuzz*.

#	Name	Description
M0	Bitflip 1/1	Flip single bit
M1	Bitflip 2/1	Flip two adjacent bits
M2	Bitflip 4/1	Flip four adjacent bits
M3	Bitflip 8/8	Flip single byte
M4	Bitflip 16/8	Flip two adjacent bytes
M5	Arith 8/8	Treat single byte as 8-bit integer, +/- value from 0 to 35
M6	Arith 16/8	Treat 2 adjacent bytes as 16-bit integer, +/- value from 0 to 35
M7	Arith 32/8	Treat 4 adjacent bytes as 32-bit integer, +/- value from 0 to 35
M8	Random 8	Overwrite random byte with random value
M9	Delete	Delete an instruction
M10	Clone	Clone an instruction
M11	Opcode	Overwrite opcode bits

B Coverage Metrics Of Prior Work

We now demonstrate why the coverage metrics of DifuzzRTL and RFUZZ cannot cover the bugs in Figure 2.

B.1 DifuzzRTL's coverage metric: control-register coverage

The control-register coverage metric of DifuzzRTL defines all the registers that drive the select signals of the MUXes as control registers. These registers in each module are concatenated into a single *module_state* register; all possible values of these *module_state* registers are defined as coverage points.

When applied to the example in Figure 2, DifuzzRTL should concatenate all the registers that drive the select signals of the two MUXes ① and ③: `flush`, `en`, `pass`, `ipass`, and `debug_en`. Since there are five 1-bit registers, there are $2^5 = 32$ possible values; DifuzzRTL considers each of them as coverage points, resulting in 32 coverage points. We now discuss in detail why the control-register coverage metric does not cover the two bugs in Figure 2.

Limitation 1. DifuzzRTL detects only certain implementations of MUXes in the RTL code. When a MUX is implemented differently (e.g., as a combination of NOT, AND, or OR gates), DifuzzRTL fails to detect the MUX and ignores the corresponding control registers. Therefore, it fails to account for certain control registers driving the select signals of such MUX implementations. Consequently, it does not produce coverage points for these control registers.

Listing 3: Verilog code of the hardware design in Figure 2 instrumented by DifuzzRTL.

```
61   assign _T =flush |en; // @[cmd3.sc 37:30]
62   assign _T_2 =pass ==ipass; // @[cmd3.sc 41:20]
63   assign sel1 = _T_2 |debug_en; // @[cmd3.sc 41:31]
64   assign _T_4 =flush &en; // @[cmd3.sc 46:17]
65   assign state_f = _T_4 ?FLUSH :state; // @[cmd3.sc 46:22]
66   assign _GEN_1 ={{2'd0}, ~sel1}; // @[cmd3.sc 52:21]
67   assign _T_6 = _GEN_1 &state_f; // @[cmd3.sc 52:21]
68   assign _GEN_2 ={{2'd0}, sel1}; // @[cmd3.sc 52:40]
69   assign _T_7 = _GEN_2 &D_READ; // @[cmd3.sc 52:40]
...  ...
78   assign en_shl =en;
79   assign en_pad ={1'h0,en_shl};
80   assign flush_shl ={flush, 1'h0};
81   assign flush_pad =flush_shl;
82   assign cache_controller_xor0 =en_pad \xor flush_pad;
...  ...
168  always @(posedge clock) begin
...    ...
207     state <= _T_6 | _T_7;
...    ...
212     vld <=debug_en | _T;
213    end
214    cache_controller_state <=cache_controller_xor0;
...    ...
218  end
```

In the controller example in Figure 2, the combinational logic ④ generates the select signal `sel1` of MUX ③. DifuzzRTL cannot detect this MUX because its RTL code is described using combinational logic (Line 51 of Listing 1: `state := ((!sel1 & state_f) | (sel1 & D_READ)))` instead of control flow constructs (like `when` block at Lines 45–49 of Listing 1), thereby failing to detect the bug *b1* in ④.

To demonstrate this limitation, we compiled the Chisel code (Listing 1) of the controller, generated the corresponding FIRRTL code, and ran DifuzzRTL on it. The instrumented Verilog code and output of DifuzzRTL instrumentation are shown in Listing 3 and Listing 4, respectively. It can be seen from the Lines 28 and 32 of DifuzzRTL's report (Listing 4) that DifuzzRTL detected only one MUX and two control registers; Lines 78–82 of the instrumented Verilog code (Listing 3) show that these control registers are `flush` and `en`. The control registers (`pass, ipass, debug_en`) generating the signal `sel1` of MUX ③ are not included. Consequently, DifuzzRTL does not have any coverage point in ④, thereby failing to detect *b1*.

Limitation 2. DifuzzRTL focuses only on the control-registers that drive the select signals of MUXes. Thus, DifuzzRTL will not cover any combinational logic that does not drive the select signals of the MUXes. In the controller example in Figure 2, the bug *b2* is in the combinational logic ⑥. DifuzzRTL cannot detect this bug since it does not cover the registers, `flush` and `en`, generating `vld` in ⑥ as these registers are not generating the select signals of any MUX.

We demonstrate this limitation of DifuzzRTL using the same instrumented Verilog code (Listing 3) and the output of DifuzzRTL instrumentation (Listing 4) . DifuzzRTL only

Listing 4: DifuzzRTL's output of the hardware design in Figure 2. MUX2 is undetected.

```
 1 ============Finding Control Registers =========
 5 numRegs: 9, numCtrlRegs: 2, numMuxes: 1
 8 ============Instrumenting Coverage ============
12 regStateSize: 2, totBitWidth: 2, numRegs: 2
13 numOptRegs: 2
25 ============Instrumentation Summary ===========
26 Total number of registers: 9
27 Total number of control registers: 2
28 Total number of muxes: 1
29 Total number of optimized registers: 2
30 Total bit width of registers: 15
31 Total bit width of control registers: 2
32 Optimized total bit width of control registers: 2
33 Total bit width of muxes: 1
```

reports the two control registers: `flush` and `en` generating the select signal `sel2` of MUX ① (Lines 78–82 of the instrumented Verilog code in Listing 3). However, DifuzzRTL does not have any coverage points for the signals in the combinational logic ⑥, where the bug resides. Combinational logic constitutes a significant portion of the hardware design, and thus these bugs cannot be overlooked as rare corner cases.

B.2 RFUZZ's coverage metric: Mux-coverage

RFUZZ uses a coverage metric called mux-coverage. It treats the select signal of each 2:1 MUX as a coverage point. When applied to the controller design in Figure 2, `sel1` and `sel2` signals are selected as the mux-coverage points. Since both are 1-bit wide, the total number of mux-coverage points is $2^1 + 2^1 = 4$ coverage points. We now discuss in detail why the mux-coverage metric does not cover the two bugs in Figure 2.
Limitation 1. RFUZZ detects only certain implementations of MUXes in the RTL code. When a MUX is implemented differently (e.g., as a combination of NOT, AND, or OR gates), RFUZZ fails to detect the MUX and ignores the corresponding select signals. Therefore, it fails to account for select signals of such MUX implementations. Consequently, it does not produce coverage point for these MUXes.

In the controller example in Figure 2, the combinational logic ④ generates the select signal `sel1` of MUX ③. RFUZZ cannot detect this MUX because its RTL code is described using combinational logic (Line 51 of Listing 1: `state := ((!sel1 & state_f) | (sel1 & D_READ)))` instead of control flow constructs (like `when` block at Lines 45–49 of Listing 1), thereby failing to detect the bug *b1* in ④.

To demonstrate this limitation, we compiled the Chisel code (Listing 1) of the controller, generated the corresponding FIRRTL code, and ran RFUZZ on it. The instrumented

Listing 5: Verilog code of the hardware design in Figure 2 instrumented by RFUZZ.

```
37  wire  _T =flush |en; // @[cmd3.sc 37:30]
38  wire  _T_2 =pass ==ipass; // @[cmd3.sc 41:20]
39  wire sel1 = _T_2 |debug_en; // @[cmd3.sc 41:31]
40  wire [2:0] state_f =profilePin ?FLUSH :state; // @[cmd3.sc
        ↪ 46:22]
41  wire  _T_5 =~sel1; // @[cmd3.sc 52:15]
42  wire [2:0] _GEN_1 ={{2'd0}, _T_5}; // @[cmd3.sc 52:21]
43  wire [2:0] _T_6 = _GEN_1 &state_f; // @[cmd3.sc 52:21]
44  wire [2:0] _GEN_2 ={{2'd0}, sel1}; // @[cmd3.sc 52:40]
45  wire [2:0] _T_7 = _GEN_2 &D_READ; // @[cmd3.sc 52:40]
... ...
48  assign auto_cover_out =flush &en;
... ...
109 always @(posedge clock) begin
...   ...
148     state <= _T_6 | _T_7;
...   ...
153     vld <=debug_en | _T;
154   end
155 end
```

Listing 6: RFUZZ's output for the hardware design in Figure 2. MUX2 is undetected.

```
51  [[coverage]]
52  port ="auto_cover_out"
... ...
58  human ="(flush and en)"
```

Verilog code and output of RFUZZ instrumentation are shown in Listing 5 and Listing 6, respectively. It can be seen from the Lines 49 and 58 of RFUZZ's report (Listing 6) that RFUZZ detected only one select signal of the MUX ①; Line 48 of the instrumented Verilog code (Listing 5) shows the same. The select signal `sel1` of MUX ③ is not included. Consequently, RFUZZ does not have any coverage point in ④, thereby failing to detect *b1*.

Limitation 2. RFUZZ focuses only on the select signals of the MUXes. Thus, RFUZZ will not cover any combinational logic that does not drive the select signals of the MUXes. In the controller example in Figure 2, the second bug *b2* is in the combinational logic ⑥. RFUZZ cannot detect this bug since it does not cover the registers, `flush` and `en`, generating `vld` in ⑥ as these registers are not the select signals of any MUX.

We demonstrate this limitation of RFUZZ using the same instrumented Verilog code (Listing 5) and the output of RFUZZ instrumentation (Listing 6) . RFUZZ only reports the one signal: the select signal `sel2` of the MUX ① (Line 58 of the RFUZZ's output). However, RFUZZ does not have any coverage points for the signals in the combinational logic ⑥, where the bug resides. Combinational logic constitutes a significant portion of the hardware design, and thus these bugs cannot be overlooked as rare corner cases.