RISCVuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing

Fabian Thomas*, Lorenz Hetterich*, Ruiyi Zhang*, Daniel Weber*, Lukas Gerlach*, Michael Schwarz*
*CISPA Helmholtz Center for Information Security
{fabian.thomas, lorenz.hetterich, ruiyi.zhang, daniel.weber, lukas.gerlach, michael.schwarz}@cispa.de

***Abstract*—The open and extensible RISC-V instruction set architecture marks a significant advancement in the CPU industry by enabling new vendors to enter the CPU market. RISC-V is quickly gaining popularity, as demonstrated by its support in the Linux kernel and its presence in consumer devices and even cloud platforms. However, the flexibility of RISC-V has resulted in a diverse range of hardware implementations, which differ in features and security measures. Additionally, no automated approach exists currently to assess the security of these implementations.**

In this paper, we introduce a novel framework, RISCVuzz, that leverages this diversity of RISC-V implementations to automatically detect vulnerabilities in hardware CPUs without the need for source code or emulators. RISCVuzz uses a differential CPU fuzzing approach to compare architectural behaviors across different vendors and CPU models. We evaluate RISCVuzz using all 5 currently available consumer-grade RISC-V CPUs and identify 3 severe security vulnerabilities along with numerous bugs. Notably, RISCVuzz identifies GhostWrite, an unprivileged instruction sequence to write attacker-controlled bytes to attacker-chosen physical memory locations, including attached devices. In 3 end-to-end attacks, we demonstrate how GhostWrite can be transformed to read physical memory and lead to arbitrary machine-mode code execution, even in cloud environments. Additionally, RISCVuzz exposes 2 unprivileged "halt-and-catch-fire" instruction sequences that result in an irrecoverable CPU halt.

I. Introduction

RISC-V is still a young instruction set architecture (ISA). Nevertheless, there is considerable support for this ISA: The Linux kernel supports RISC-V CPUs in the upstream code, and major compilers, such as GCC and Clang, support RISC-V [39]. Furthermore, mainstream Linux distributions support the RISC-V architecture, e.g., Ubuntu and Debian [41]. Besides software support, there is a steadily rising number of RISC-V CPUs. While the first CPUs were mainly softcores designed for emulators and FPGAs [1], [8], there are already a small number of hardware cores available on the market [47], [54], [55]. These cores are used in single-board computers (SBCs) similar to the Raspberry Pi, as well as laptops [50], [63], mobile phones [49], servers [42], [46], and gaming consoles [50].

The available RISC-V CPUs implement the base ISA and typically an additional selection of finalized ISA extensions, such as compressed instructions or instructions for handling floating-point numbers [16]–[18], [61]. Ongoing work also focuses on thoroughly testing the implementation of the finalized ISA extensions to ensure that CPUs implement them correctly [20]. However, RISC-V also supports vendor-specific custom ISA extensions that are already used, e.g., to implement cache-maintenance instructions in the T-Head XuanTie C906 [22]. Worse, for most available high-end cores, no source code is available. Thus, previous approaches that focus on finding vulnerabilities in RISC-V cores on an RTL level [23], [26], [51] cannot be applied by researchers or other third parties. Consequently, while crucial, it remains challenging to analyze the security of high-end RISC-V cores due to missing documentation and source code.

In this paper, we ask the following research question:

Can we leverage the inhomogeneity of RISC-V implementations to automatically find architectural CPU vulnerabilities without requiring the source code of the CPU?

To answer this question, we build RISCVuzz, a differential CPU fuzzing framework for analyzing RISC-V CPUs. RISCVuzz relies on the assumption that the *architectural* result of *every* instruction has to be the *same across different CPUs* if the instruction is supported. Additionally, ISA extensions can be detected using the same approach, independent of whether they are standardized or vendor-specific. An instruction from an extension manifests itself by showing different architectural behavior across different CPUs. RISCVuzz executes instruction sequences with different parameters on different RISC-V CPUs, including emulators. These instruction sequences can be as simple as single instructions to complex instruction chains consisting of multiple thousands instructions as shown in prior work [51]. Any deviation from the majority vote concerning the output or side effect, i.e., system crash, is reported as a potentially misbehaving instruction sequence. These instruction sequences require further manual inspection.

We evaluate RISCVuzz on 5 RISC-V hardware CPUs: T-Head XuanTie C906/C908/C910, and SiFive U54/U74. These are all currently consumer-available hardware RISC-V CPUs running a 64-bit Linux operating system, and are used in various devices. For the evaluation, we use 10 different devices featuring these CPUs. Additionally, we evaluate RISCVuzz on 4 emulators. In total, RISCVuzz discovers 3 architectural CPU vulnerabilities and numerous bugs. On the T-Head XuanTie C910 CPU, RISCVuzz discovers a successfully-executing instruction that leads to a segmentation fault on other CPUs.

Further manual analysis reveals a CPU vulnerability that fully breaks integrity by providing a write-anything-anywhere primitive for *physical memory* to unprivileged users. We dub this vulnerability GhostWrite for easy reference, as it fully circumvents virtual memory and caches, making it also invisible in performance counters. GhostWrite is in the non-compliant implementation of the high-order strided vector-store instructions (`vse128.v` to `vse1024.v`). The other 2 vulnerabilities belong to the class of "halt-and-catch-fire" CPU vulnerabilities [10] that can be used for unprivileged denial-of-service attacks on the CPU, one on each of T-Head XuanTie C906 and C908. One of these vulnerabilities is in the vendor-specific `XTheadMemIdx` extension, which provides additional memory operations such as increment-address-before-load (`th.lbib`). The other is a range of broken vector instructions that halt the CPU core. RISCVuzz discovers these vulnerabilities fully automated by observing hangs of these CPUs while the instruction sequence succeeds on other CPUs. Additionally, we discover numerous architectural bugs in CPUs and emulators, most of them within seconds of fuzzing. These include address-handling bugs, decoder bugs, ISA incompatibilities, and fault-reporting issues, as well as segmentation faults in the two latest major versions of QEMU.

We demonstrate the security impact of our findings in 4 case studies. On the T-Head XuanTie C910, we build an end-to-end attack with GhostWrite that allows unprivileged users to read and write arbitrary memory, including machine-mode code and devices mapped via MMIO. Additionally, we build two end-to-end attacks escalating privileges to root and machine mode by using GhostWrite to inject and execute code in supervisor- and machine mode. As a third end-to-end attack, we show how GhostWrite can be used on cryptographic keys to mount an ineffective fault attack [9], [11], [65], fully recovering a 2048-bit RSA key within $30\,\mathrm{min}$. We demonstrate that this vulnerability can also be exploited by unprivileged users in the cloud by successfully testing it on the Scaleway TH1520 instances. For the instructions that halt the CPU, we demonstrate that they can be used by any unprivileged application and also work from inside Docker containers.

The only mitigation we identify for the bugs in the vector extension, e.g., GhostWrite, is disabling the extension, which breaks applications using it. In a benchmark using `rvv-bench`, we measure an overhead of up to $77\,\%$ when this mitigation is active. Thus, in spite of preventing exploitation, it is not a practical solution for entities relying on the vector extension. For the C906 CPU-halting bug, we find no mitigation since the responsible vendor extension cannot be disabled.

Our results provide interesting insights into the current state of hardware RISC-V CPUs. Vendor extensions and "rushed" implementations of non-finalized extensions do not only lead to bugs but also to exploitable security vulnerabilities that are difficult to mitigate. Our results indicate that the base instruction set is tested significantly better than complex extensions, such as vector extensions. An additional insight is that even for open-source cores, such as the C910, the hardware implementation differs from the released source. RISCVuzz discovers most bugs and vulnerabilities within seconds, showing the efficacy of our approach. This is even true for rather simple instruction-sequence generation, which aligns with previous work on software fuzzing [33]. We emphasize that our approach is orthogonal to RTL fuzzing, covering scenarios RTL fuzzing cannot. However, even if the source code is available, RISCVuzz might find bugs introduced by the synthesis that are invisible in the RTL. Hence, we argue that post-silicon fuzzing is a valuable extension of existing pre-silicon fuzzers [23], [26], [43], [51]. Another insight is that the simplicity of RISC-V does not prevent bugs, but prevents mitigating them as done in x86 CPUs using microcode updates [5], [34], [35], [38], [66].

Contributions. We summarize our contributions as follows.

- We present RISCVuzz, a differential CPU-fuzzing framework for finding CPU vulnerabilities on hardware RISC-V CPUs that does not require access to the CPU source or an emulator, enabling vulnerability discovery on high-end cores.
- We automatically test 5 different off-the-shelf RISC-V CPUs and 4 emulators and discover 3 severe CPU vulnerabilities on real-world-deployed CPUs: an unprivileged arbitrary physical write primitive we dub GhostWrite and 2 unprivileged "halt-and-catch-fire instructions".
- In 4 end-to-end attacks, we demonstrate the impact of the vulnerabilities by reading and writing arbitrary memory and executing code with kernel- and machine-mode privileges, fully breaking the confidentiality and integrity of these systems, including cloud setups.
- We discover numerous additional architectural bugs in the tested CPUs as well as in emulators within seconds of fuzzing.

Responsible Disclosure. We reported all the security-critical vulnerabilities on the C906, C908, and C910 to T-Head. They acknowledged and reproduced GhostWrite and the C906 CPU-halting instruction sequence. We have no answer yet for the C908 CPU-halting instructions. Further, we also reported GhostWrite to Scaleway since they offer C910-based bare-metal machines in the cloud. Scaleway reproduced our findings and is currently in the process of giving out instructions to customers for manually rolling out kernel patches that disable the vector extension, mitigating GhostWrite. Additionally, we reported a segmentation fault that is present in the latest version of QEMU.

Availability. We will open source the RISCVuzz framework and our reproducers with acceptance of the paper. We provide a preliminary version of these artifacts at the following address: https://anonymous.4open.science/r/riscvuzz-artifacts-116D

II. Background

This section covers the relevant background required for the remainder of the paper.

A. RISC-V

RISC-V is an open instruction set architecture (ISA) developed by the RISC-V foundation. The RISC-V ISA consists of a core instruction set that must be implemented by all RISC-V CPUs and extensions that can be implemented as needed [59], [60]. An example of such an extension is the vector extension [17], which is implemented in the XuanTie C908. In addition, multiple vendor-specific extensions, some

only vaguely documented, are added to RISC-V cores to equip them with additional functionality [16], [22], [56]. ISA variants for 32 bit, 64 bit, and plans for 128 bit addressing [59] are available, making RISC-V suitable for a wide range of devices. Nowadays, RISC-V CPUs are not only available as soft cores for FPGAs but are already used in embedded devices [40], SBCs [29], laptops [50], [63], and cloud computing [42]. RISC-V devices are supported by upstream Linux [15] and can run a variety of distributions, e.g., Ubuntu or Debian.

a) Privilege Levels: RISC-V has three privilege levels: User (U) for unprivileged applications, Supervisor (S) for operating systems, and Machine (M) to manage trusted execution environments [60]. While only M is required to be implemented, most non-embedded RISC-V CPUs implement all three privilege levels. Access to Control and Status Registers (CSRs) and privileged instructions is limited depending on the current privilege level.

b) Paging: RISC-V supports virtual memory through paging. With virtual memory, the single physical address space for DRAM and memory-mapped I/O is isolated through virtual address spaces. The mapping between virtual and physical addresses is defined in per-process page tables. A page table is a sparse tree-like data structure mapping virtual memory blocks of fixed size called *memory pages* to physical memory pages. The size of such memory pages is typically 4 kB [60]. In addition to the physical address, page tables can store metadata for each memory page, for instance, whether a memory page is present as opposed to swapped to disk, whether it can be accessed from userspace, or whether it is writable.

B. Fuzzing

Fuzzing is a soft- and hardware testing technique that provides randomly generated inputs to hardware or software targets and checks that they behave in an expected way. While fuzzing cannot prove the absence of bugs, it has been shown to be effective in finding bugs in soft- and hardware [23], [26], [43], [51]. A special case is differential fuzzing [32], where multiple targets implementing the same specification are tested against each other. Each difference resulting from the execution of identical inputs by different targets is considered a violation of the specification by one of the targets. Differential fuzzing has the benefit that no golden reference model, which correctly implements the specification, is required, as bugs manifest through differing behavior.

C. CPU Vulnerabilities

While software-based side-channel attacks have been known for decades [28], recent years have proven that more critical CPU vulnerabilities exist [27], [31], [35], [36]. A prominent class of attacks are transient-execution attacks [7], [27], [31], which abuse performance optimizations of modern CPUs, such as out-of-order and speculative execution. While transient-execution attacks allow for leaking data from various security boundaries [27], [58], they are restricted to read primitives. Transient-execution attacks do not violate the specification of a CPU but allow an attacker to execute security-boundary-crossing instructions that are never committed to the architectural state of a CPU. However, those instructions leave microarchitectural traces that attackers use to recover confidential information. In contrast to transient-execution attacks, architectural bugs are mismatches between the CPU specification and implementation. For example, the Pentium F00F bug allows an unprivileged attacker to lock up an affected system by executing a specific instruction even though the instruction encoding is invalid and should raise an exception according to the specification [10]. More recently, these architectural bugs gained traction, with multiple CPU bugs causing architectural data leakage [6], [35], [36], [66].

III. Methodology

This section describes the methodology of our work. We introduce the main idea in Section III-A and outline challenges in Section III-B. In Section III-C, we introduce the fuzzing targets used in the remainder of the paper. We provide implementation details in Section IV.

A. Idea

We rely on the basic assumption that the *architectural* result of *every* instruction has to be the *same across different CPUs*. This assumption ensures that different CPUs have to adhere to the ISA specification. However, this assumption is only valid for instructions supported on all the tested CPUs. Thus, we consider any architectural effect that differs between CPUs an *instruction anomaly* that has to be investigated further, excluding instructions only supported on one CPU. These instruction anomalies are likely due to a bug, or worse, a security vulnerability in the CPU.

The main advantage of this approach compared to previous CPU fuzzers searching for architectural bugs [23], [26], [51] is that we require neither source code nor golden models. Thus, this is the first approach that can automatically find vulnerabilities on closed-source CPUs, such as the T-Head C908. Moreover, running code on hardware CPUs is significantly faster than emulating them.

In addition to instruction differences, ISA extensions can be detected using our approach. This is true for both documented and undocumented extensions if two CPUs implement differing extensions. An instruction from an extension manifests itself by showing different architectural behavior across different CPUs, e.g., by throwing an illegal-instruction exception on a subset of CPUs.

Finally, this approach can be extended from single instructions to instruction sequences. Although many previous CPU bugs do not require the interaction between multiple instructions [6], [35], [66], some do [36], [51]. Thus, we also compare the architectural effects of instruction sequences. As our differential approach is generic, this addition does not result in conceptual changes but only in additional engineering effort. However, unlike single instructions, we cannot exhaustively test all instruction sequences. Thus we have to select instructions pseudo-randomly, i.e., we rely on fuzzing techniques for generating instruction sequences.

B. Challenges

While the idea of differential CPU testing (or fuzzing) is quite intuitive, we identify multiple challenges in both design and implementation. Conceptually, there are the following 3 challenges:

a) $\mathcal{C}1$: Sequence Generation: Executing every possible instruction encoding in the 32-bit encoding space already results in a large search space. This search space grows exponentially with the length of the instruction sequence. Hence, randomly executing bitstreams is inefficient for exploring "interesting" effects. Additionally, if there is a difference between two CPUs in one instruction, such an instruction is reported multiple times if it encodes an immediate. As the difference is likely independent of the immediate, testing all possible immediates wastes resources and inflates the result set, which has to be checked manually. Hence, the challenge is finding a trade-off between coverage of the encoding space and the number of tested instructions. In Section IV-C1, we describe how our proof-of-concept implementation RISCVuzz solves that challenge by using a bottom-up approach to gradually increase the search space using instruction types inferred from instruction encodings.

b) $\mathcal{C}2$: Non-deterministic Effects: Comparing the architectural effects of instructions requires that these effects only depend on factors we can control, e.g., memory and register content. Unfortunately, this is not the case for all instructions. Some instructions provide internal values of the CPU, such as performance counters. These values often depend on the CPU state and previous instructions executed on the core and can thus not be controlled. Finally, memory reads from certain addresses, such as Linux vDSO [21] return values that are out of the control of the testing framework. All these cases have to be considered to avoid false positives, i.e., reporting different behavior across CPUs even though the instructions have the same behavior. In Section IV-B1, we describe how we prevent the reporting of non-deterministic effects by minimizing the sources of non-determinism and ignoring the results of the remaining non-deterministic instructions.

c) $\mathcal{C}3$: Test-framework Integrity: The test framework has to record the architectural effects of instruction sequences. Thus, from a high-level perspective, architectural states, such as register and memory content, must be saved before and after executing the sequence. While this is relatively easy for many instructions, some instructions need special care to ensure the integrity of the test framework. These instructions include those that change the control flow, e.g., calls and (conditional) jumps, those that change the CPU behavior, e.g., CSR writes, and those that change the stack pointer or stack content. Hence, the test framework implementation has to handle all corner cases that would change the saved architectural states or the internal state of the test framework. We describe the implementation details of this approach in Section IV-B2.

C. Fuzzing Targets

While the number of silicon RISC-V cores is still limited, we test all widespread commercially available 64-bit RISC-V cores that support booting a Linux distribution. At the time of writing, there are 2 manufacturers of silicon RISC-V CPUs—SiFive and T-Head Semiconductors. We test on 2 SiFive and 3 T-Head CPU models as listed in Table I. All tested CPUs are mounted on single-board computers. We also use the C910 in the Scaleway cloud [42]. The targets run various operating systems such as Ubuntu or Debian (cf. Table III in Section A).

All tested cores support at least the base ISA, the standard extensions, and compressed instructions, i.e., RV64GC. The C908 supports the ratified RISC-V vector extension (`v`), while C906 and C910 use a pre-ratified draft version 0.7.1 (`v0p7`) supported by vendor-provided kernels and toolchains. Further, we test on different QEMU versions from version 6 (default on Ubuntu 22.04) to 9 (newest) (cf. Table I).

TABLE I: Overview of tested RISC-V boards and emulators. We test 5 CPUs and 4 emulators. The CPUs come from 2 different vendors.

Board	CPU	CPU Vendor	Relevant Extensions
A	U54	SiFive	-
B	U74		-
C, D, E	C906	T-Head	`v0p7`, `zfh`, `XTheadMemIdx`
F	C908		`v`, `zfh`, `XTheadMemIdx`
G, H, I	C910		`v0p7`, `zfh`, `XTheadMemIdx`
Emulator	**Version**		**Relevant Extensions**
A	QEMU 6.2.0		-
B	QEMU 7.2.0		v
C	QEMU 8.2.2		v
D	QEMU 9.0.0		v

IV. RISCVuzz Framework

In this section, we describe RISCVuzz, our proof-of-concept implementation of the methodology described in Section III. Specifically, Section IV-A describes the design of RISCVuzz. Section IV-B and Section IV-C discuss relevant design and implementation details of the client and server components of RISCVuzz, respectively.

A. Design Overview

RISCVuzz uses a centralized design: A server orchestrating the testing, and the clients, i.e., RISC-V CPUs, are connected to the server. We reduce the task of the clients to a minimum due to resource constraints on the clients. A client receives test cases, i.e., instruction sequences plus input, over the network, runs them, and reports back the resulting state e.g., register values and changed memory contents. The server is responsible for generating the test cases, distributing the cases to the clients, collecting the results, and analyzing the results.

The server-based approach has different advantages over a decentralized approach. First, the clients typically have limited storage, making storing the architectural effects of all tested instructions difficult. With a 32-bit search space for single instructions, the resulting states require multiple gigabytes of storage. Second, the CPU processing power of the clients is also constrained, leading to non-negligible overhead for generating the test cases in addition to executing them. As we aim to test instructions as fast as possible, we delegate all resource-intensive tasks to more powerful CPUs. Finally, with a central server-based solution, we can compare the results of the tests in parallel and do not have to wait for the completion of the test runs on all machines.

B. Client

The client is a runner of server-provided test cases. It is implemented in a mixture of C and RISC-V assembly. To execute test cases, the client sets the registers as specified,

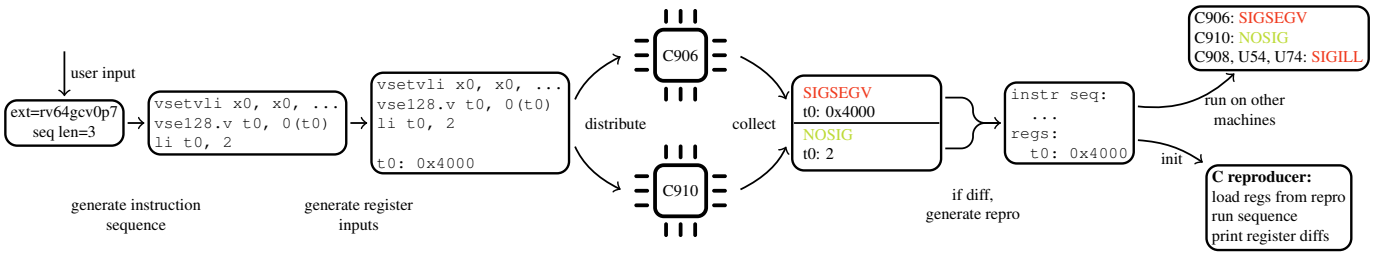

Fig. 1: Overview of RISCVuzz. The user selects the enabled extensions and sequence length. The server generates an instruction sequence and register inputs for the clients and compares the results. Differences are logged as a reproducer file.

runs the provided instruction sequence, and reports the results to the server. In case of instructions that access arbitrary unmapped memory, execution is interrupted by a segmentation fault. Since RISCVuzz should compare memory differences too, RISCVuzz incorporates a routine for mapping these unavailable pages. When the client detects a segmentation fault, it tries to map two pages, the page causing the fault and the subsequent page to handle corner cases where accesses might span two pages. This lazy mapping of pages has the advantage that only a small amount of memory has to be scanned for modifications caused by the instruction sequence. The mapped pages are filled once with all '0's and once with all '1's. This is to detect writes of only '0's or only '1's respectively too. This procedure is repeated until a threshold is reached[1] or until no segmentation fault is triggered anymore by the instruction sequence. After mapping the pages, the testcase is restarted. The differences on all memory pages are recorded and included in the results.

As discussed in Section III-B, RISCVuzz faces two challenges: non-deterministic instruction effects ($\mathcal{C}2$) and the integrity of the client ($\mathcal{C}3$). In the following, we discuss how we tackle these challenges in the design of RISCVuzz.

1) $\mathcal{C}2$: Noise Removal: Since we assume every difference in register values to be a bug, we must ensure this does not happen randomly. The first step in this direction is using static compilation. Static compilation removes noise from differences that occur in shared libraries and ensures that we can run the same binary on all machines. We use Nix [13] for the static building to ensure that our builds stay reproducible across different machines employed for compilation. Reproducibility is essential since any change in the binary layout of the client implementation could introduce new noise or break framework integrity ($\mathcal{C}3$). Additionally, we unmap the `vdso`, `vdso_data`, and `vvar` sections since loading from them results in differing values between kernels. Another noise source are instructions that naturally introduce noise, such as the `rdcycle` instruction. To remove these noise sources automatically, we execute each instruction in the space of selected extensions twice and exclude those that do not produce the same result.

2) $\mathcal{C}3$: State Protection: As RISCVuzz executes arbitrary code sequences, these code sequences can modify its internal state. Thus, RISCVuzz has to protect its internal state to ensure a correct reporting of instruction effects. Specifically, RISCVuzz has to ensure the integrity of its internally used *registers*, *memory regions*, and *control flow*. Figure 2 illustrates the design of the integrity-providing sandbox we discuss in the following.

a) Registers: For the integrity of *registers*, we employ two routines that ensure the register state is saved and restored correctly. When a target instruction triggers a signal, such as an illegal instruction, the kernel saves the architectural state of the CPU. We simply copy this state and return to the execution loop with a `longjmp`, which restores the register state. When no signal is triggered, we use an instruction sequence similar to the one used by the kernel when handling an interrupt. This sequence saves/restores every register to/from memory before and after executing the instruction sequence.

b) Memory: Memory state is the second dimension to protect. While single instructions can only access memory close to the current register contents defined by the inputs, instruction sequences, and non-RISC instructions can arbitrarily shift these inputs and access any memory. Further, we want to be able to provide various inputs, such as '-1', via the registers. Shifting the value '-1', which results in the register value `0xffff...`, results in addresses potentially pointing to the program stack. We leverage the large virtual address space to "hide" the internal data in a region that is difficult to overwrite accidentally. The kernels we run on our CPUs all use the Sv39 paging mode [60], i.e., 512 GB of virtual space is available for "hiding". Consequently, we move the data section to a "safe" region of memory that we experimentally determined. Further, we switch to a new stack that we set to this region as well. While this approach is heuristic, it works well in practice.

c) Control Flow: The third dimension is *control flow*. No instruction sequence must jump out of RISCVuzz's logic or infinitely lock up the client. We embed placeholder instructions into a padding of `ebreak` instructions to ensure that control flow is restricted to our path. These placeholder instructions are replaced in memory by the runner code. The `ebreak` padding ensures that the generated signal data is accurate. Padding with `nops` would hide the target of relative jumps and branches since the runner would fall through to the last instruction after the `nop`-sled. We again rely on the size of the virtual address space to "hide" the client logic. We move the sandbox to its own memory range and use absolute jumps to get into and out of the runner sandbox. Two instructions in the sandbox load the last two registers, which are needed for the absolute jump and as the base for loading the registers.

[1]For our tests, we use 10 as a threshold which we experimentally found to yield good results.

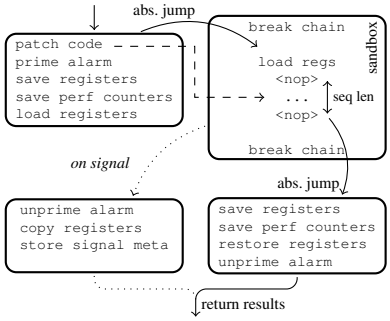

Fig. 2: The runner sandbox includes `nop` placeholders surrounded by `ebreak` instructions. The runner patches the placeholder instructions, primes the alarm, saves registers, loads the supplied fuzzing registers, and jumps into the sandbox. The executed instruction sequence either returns via the runner, or a signal, and returns the results.

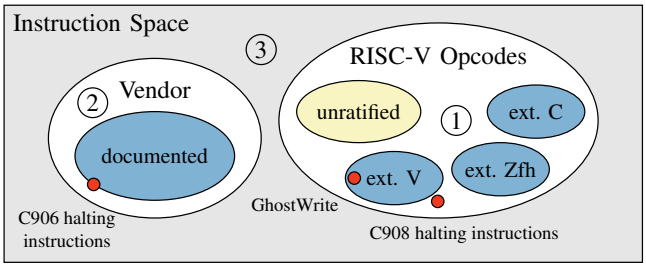

Fig. 3: Overview of the RISC-V instruction space. RISC-V Opcodes (①) covers most parts of the RISC-V ISA specification. The specification reserves parts of the address space for custom vendor extensions (②). Other parts are either reserved for future use or unclaimed (③). The dots describe where our discovered bugs are in the instruction space.

We handle infinite loops by priming an alarm before jumping into the sandbox. This alarm interrupts the client after a configurable timeout, breaking out of any loop. The `ebreak` padding further minimizes the chance of infinite loops compared to padding with `nops`.

C. Server

The server is the pivot of our setup. Figure 1 illustrates the logic of the server. It generates instruction sequences and input registers and sends these to the clients (Section IV-C1). The clients run the test cases and report the resulting architectural states to the server (Section IV-C2). The server compares these states and logs differences as simple reproducer files, which can be used for further analysis in a simple C program or by running the reproducer on other machines (Section IV-C3).

1) C1: Sequence Generation: Instead of generating random 4-byte sequences, we use a bottom-up approach based on the instruction encoding to gradually increase the covered instruction space. Based on specific bits in the instruction encoding, RISC-V allows to classify the type of instruction and whether an instruction is a standard instruction (①) or a vendor-specific instruction (②), leading to groups as illustrated in Figure 3. This approach allows selectively including and excluding ISA extensions in our tests. Consequently, this makes it easy to first fuzz only the undocumented space (③). This drastically shrinks the search space by $85.51\,\%$ (Table II). Note that we do not have to separate ISA extensions exactly. The encoding-based filtering is only a rough but deterministic guidance technique to prevent the fuzzer from wasting resources on instruction encodings that mainly consist of instructions with large immediate encodings. We still strive to cover the entire 4-byte instruction space of RISC-V but want to focus on more promising parts first. In the following, we describe in more detail how this bottom-up approach works.

TABLE II: Distribution of 4-byte RISC-V instruction space as documented by RISC-V Opcodes. All ratified (official) parts of the ISA cover $84.03\,\%$ of the instruction space. The vector extensions and T-Head extension cover only small parts of the instruction space. Overall, $85.51\,\%$ of instructions are specified, the rest ($14.49\,\%$) are unknown or not specified.

ISA part	Percentage
Ratified + unratified	$85.02\,\%$
Ratified	$84.03\,\%$
Vector extension (`v`)	$1.05\,\%$
Vector extension (`v0p7`)	$0.81\,\%$
T-Head vendor extension	$0.39\,\%$
Overall known	$85.51\,\%$

a) Instruction Classification: We use the official RISC-V Opcodes repository [19] for building our filters since it encodes all standard RISC-V instructions in a machine-parsable format. Further, it clusters the instructions into their respective extension. Moreover, it includes some of the unratified extensions.

b) Instruction Exclusion: Encoding-based filtering also allows for excluding instructions or entire instruction classes, such as CSR-based instructions. As these instructions can change the behavior of instructions on the current core, they require extra handling to avoid introducing false positives in the differences. We leave the coverage of these CSR-based instructions to future work.

c) Instruction Selection: The server randomly chooses instructions from the instructions used for fuzzing and assembles them based on the documented encoding. We generally initialize immediate parts of the instruction with values that might lead to corner cases, e.g., '-1' or '0'. To ensure that any instruction encoding can be achieved, though, we mix in a random immediate in 1 out of 8 cases. For the rest of the fields, including register fields, we provide the required number of random bits.

Not defined or missing instructions in RISC-V Opcodes are chosen by generating a random 4-byte value that cannot be decoded to a valid instruction. For any non-documented instruction, RISCVuzz does not have to fill any bitfields, as a full instruction encoding with all bits set is already chosen. Thus, RISCVuzz covers the entire instruction space with this approach.

2) Input Distribution: The server distributes the generated fuzzing inputs and collects the results (cf. Figure 1). Since we want to achieve high throughput fuzzing on the clients (cf.

Section V), we implement 3 optimizations for the transfer. First, we only send back registers and memory contents that *changed* during the execution of the instruction sequence. Second, we restrict ourselves to a list of values of register contents. This list of values is shared between client and server. The server then transmits one byte to select the respective value on the client side. This reduces data consumption from 4 to 1 byte per register for general-purpose registers and saves 15 bytes per register for vector registers. Third, we send batches of fuzzing inputs and ensure enough fuzzing inputs are buffered in the client. These optimizations ensure that the clients are never idle and the network is efficiently used.

3) Logging Differences: In the next step, the server compares the collected architectural states. If it finds a difference, it logs a reproducer file for the fuzzing input. This reproducer can then rerun the fuzzing input on selected machines or automatically create a simple C-based program consisting of the instruction sequence and the register inputs (cf. Figure 1). This reproducer file can then be compiled and executed as a standalone binary on any RISC-V CPU for further manual analysis.

While RISCVuzz only adds the necessary code to the reproducer, i.e., the instruction sequence and architectural state initialization, the reproducer is not necessarily minimal. However, in practice, the reproducer is typically small enough for an analysis. Still, if necessary, program reduction techniques, such as those discussed by Solt et al. [51], can be used to reduce the reproducer further.

V. Evaluation

In this section, we evaluate RISCVuzz. We evaluate the general performance of testing instruction sequences (Section V-A), summarize the findings of RISCVuzz (Section V-B), including the most severe finding, GhostWrite, and evaluate how long it takes RISCVuzz to discover our findings (Section V-C).

A. Fuzzing Performance

This section focuses on different performance metrics of RISCVuzz. All experiments use 1 core of an Intel Core i9-13900K as the server.

1) General Throughput: To assess the general throughput of RISCVuzz, we benchmark a fuzzing run with only the base ISA with 1 client instance on each of the target CPUs (cf. Section III-C). We test 18 194 (C906) to 59 205 (C908) instructions per second on average. Figure 4 provides the results for all CPUs. Comparing the results to the fastest state-of-the-art RISC-V RTL fuzzer Cascade [51] shows that fuzzing on hardware cores is orders of magnitude faster. Cascade achieves $\frac{2181}{256} = 9$ instructions per second. We divide the reported throughput by 256 since the Cascade evaluation uses a 512-core machine while we only use 2 cores (client and server). However, Cascade already uses a sequence length of 10 000 to achieve this throughput, while we use a sequence length of 1.

Note that we cannot evaluate the bare-metal performance of Cascade. RTL fuzzers such as Cascade [51] or DifuzzRTL [23] generate bare-metal ELF binaries, i.e., binaries that use privileged instructions to setup the interrupt vector table, enable the FPU, or reset performance counters to a shared state. Running these binaries on a Linux host is not possible by design and requires significant modifications to the design of ELF generation, e.g., adapting the exception handling approach to userspace logic. Further, running these bare-metal ELF binaries one by one on a CPU is an entirely new problem. For example, on some boards this would require removing the microSD card, copying the new ELF binary onto it, inserting it, and powering on the board again for every testcase.

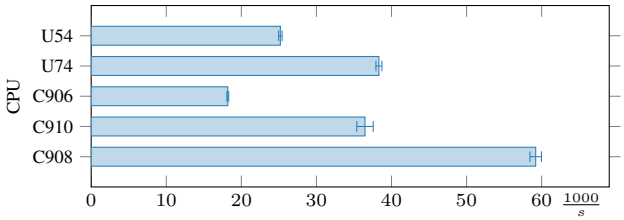

Fig. 4: Relative performance of each tested CPU. The C908 is by far the fastest CPU in our test. The C910 and U74 perform similarly. C906 and U54 are the slowest CPUs.

Takeaway Testing on hardware cores is orders of magnitude faster than on emulated cores.

2) Multi-Core Scaling: Fuzzing throughput can be improved by deploying fuzzing clients to more than one CPU core. Such parallelization relies on the fact that each fuzzing corpus should not impact the others. Thus, they can be arbitrarily distributed to cores. Switching to 2 cores on the C910 nearly doubles performance from 39 994 to 71 825 $\frac{\text{instr.}}{s}$, while employing one more core again improves performance but only by half of the increase we see when going from 1 to 2 cores (92 943 $\frac{\text{instr.}}{s}$). Adding the last core only marginally improves performance because the server tops at 97 207 $\frac{\text{instr.}}{s}$. As the C906 is a single-core CPU, we can instead use multiple CPUs, which comes close to a linear increase in throughput. nearly resembling a linear increase in throughput.

Takeaway Multiple cores can be used to increase fuzzing throughput. For single-core machines, multiple machines can be joined to achieve the same effect.

3) Sequence-Length Scaling: We evaluate the impact of the sequence length on the performance. If there is no exception, we rely on the retired instructions counter to collect the number of executed instructions. Otherwise, we infer the number from the program counter at which the signal is triggered. When the trapped program counter is not in bounds of the sandbox, we assume that only 1 instruction was executed. This can happen when a jump or branch instruction is executed.

Figure 5 shows the result of increasing the sequence length on the C906. The performance increases up to a sequence length of 5 and then gradually decreases with further increasing sequence length. The results are as expected since, at some point, increasing the sequence length only rarely leads to more instructions executed per iteration, as some earlier instruction might already raise an exception (cf. Section C). The added network overhead of sending one more instruction absorbs this slight increase and decreases performance overall. A sequence length of 3 is a good tradeoff since adding more

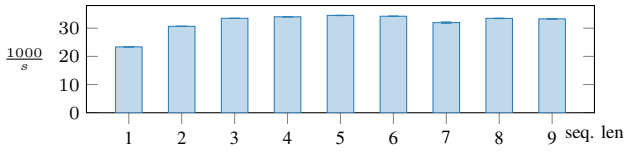

Fig. 5: The fuzzing performance on the C906 increases up to sequences of 5 instructions, then falls gradually with increasing sequence length.

instructions only slightly improves performance while causing more congestion on the network, potentially hindering other clients from receiving data.

Note that Cascade uses advanced techniques for generating longer valid instruction sequences [51]. Although we suspect such techniques would improve the throughput of RISCVuzz, we leave improving the sequence generation for future work.

B. Findings

In this section, we summarize the findings of RISCVuzz. We categorize the findings into address-handling bugs, decoder bugs, ISA incompatibilities, and fault-reporting issues. We discuss the findings with the highest security impact, i.e., GhostWrite and the C906 and C908 CPU-halting vulnerabilities, in more detail in Section VI and Section VII, respectively. Figure 3 visualizes where RISCVuzz finds the most severe bugs. GhostWrite is in the vector extension. The C908 halting instructions are illegally encoded vector instructions close to but outside the vector extension. The C906 halting instructions are on the edge of the documented vendor extension since they use an edge case in the instruction encoding.

Note that RISCVuzz automatically finds architectural differences that are in most cases bugs. The analysis whether these differences are security vulnerabilities still requires a manual analysis. However, due to the small reproducers created by RISCVuzz, the manual analysis is in many cases relatively quick. GhostWrite produces differences when fuzzing the 0.7.1 vector extension between C906 and C910. While the illegally-encoded vector-store instructions generate a segmentation fault on illegal memory addresses, the C910 generates no exception. We provide an example of such a difference logged into a reproducer file in Section D. During manual inspection of the instruction behavior, by varying register values, we observe kernel crashes when passing addresses in the physical kernel range which motivates further analysis of these faulty instructions (cf. Section VI). The C906 and C908 CPU-halting instruction sequences directly crash the fuzzing client, therefore no further analysis of such reproducers is needed as such a denial of service can always be considered a security problem. The other bugs we summarize below generate similar patterns, i.e., either hangs or differences, which motivate further manual inspection.

a) Address-handling: RISCVuzz finds different bugs around virtual address handling. The `vse128.v` instruction on the C910 does not translate the provided virtual address to a physical address but instead interprets it directly as a physical address, giving attackers a physical write primitive (cf. Section VI). Additionally, on the C910, reading from physically-backed virtual address '0' locks the CPU, requiring a hard reset. On the C906, C908, and C910, a load to a non-canonical address is stuck until an interrupt arrives if the canonical part of the address is a valid address. RISCVuzz generates such addresses if any upper bits of a valid virtual address are modified by an instruction, e.g., by an `xor`, before the load happens.

b) Decoder Bugs: RISCVuzz discovers decoding bugs on different hardware CPUs and in emulators. We find CPU-halting instruction sequences on the C906 and C908 that we suspect to be decoder issues (cf. Section VII). On the C906 and C910, RISCVuzz discovers `fence` and `fence.i` instructions that raise an illegal-instruction exception, although they are valid according to the ISA specification. The RISC-V standard reserves these instructions for "finer-grain fences in future extensions" and dictates that "implementations shall ignore these fields" [59]. Conversely, RISCVuzz discovers instructions that do not raise such an exception although they are invalid. For example, the C906 and C910 execute the half-precision floating-point instructions `fsqrt.h` and `fmv.x.h` even when the `rs2` field is $\neq 0$ [18]. Finally, for the latest versions QEMU 9.0.0 and QEMU 8.2.2 (Emulator D and C), RISCVuzz discovers that cache-block management instructions such as `cbo.inval` crash QEMU with a segmentation fault. For QEMU 7.2.0 (Emulator B), RISCVuzz discovers that truncating vector conversion instructions such as `vfncvt.rtz.x.f.w` crash QEMU. However, as the crash is due to an assertion, we do not expect that this is further exploitable.

c) ISA Incompatibility: The C906 and C910 are not fully compatible with the ISA specifications. These CPUs do not ignore writes to bits 8 to 10 of the `fcsr` register. Both the C910 and the C908 support a subset of the vector extension. This manifests itself in some of the instructions doing nothing, others doing unexpected things (cf. Section VI), and some not being implemented at all. Interestingly, the subset of instructions also differs between the two CPUs.

d) Fault-reporting Issues: On all tested CPUs, RISCVuzz discovers bugs during fault reporting. Overall, there are various inconsistencies in the raised signal for exceptions. SiFive CPUs tend to raise bus faults, whereas T-Head CPUs raise segmentation faults. Additionally, the reported program counter of the fault and the faulting address are not always correct. On the C910, the reported address for faults is rounded up to the next multiple of 16 if the address modulo 16 is larger than 8. The C908 reports segmentation faults for valid non-aligned addresses, where the correct behavior is to raise a bus error.

C. Time to Bug

In line with other papers on fuzzing [43], [51], we provide the fuzzing time to find the bugs. We use all extensions, i.e., all ratified and unratified extensions, during fuzzing. This is the worst case for finding the bugs, since the only restriction we pose on the fuzzing space is to use only documented instructions. We further test on a single core. Thus, the numbers from Section V-A1 apply.

We find GhostWrite within the first second of fuzzing as no special encoding in the broken instruction is needed to make the bug visible. Thus, the fuzzer only needs to select one of the

```
1 ; t0 = physical address, a0 = byte to be written
2 vsetvli zero, zero, e8, m1
3 vmv.v.x v0, a0
4 ; encoded: 0x10028027
5 vse128.v v0, 0(t0)
```

Listing 1: Code of GhostWrite. `vsetvli` and `vmv.v.x` set up the vector engine's internal state and the byte to be written. The non-standardized `vse128.v` instruction (provided as machine code `0x10028027`) performs the physical write.

8 broken instructions out of 1283 possible instructions when enabling all extensions as outlined above.

We find the C906 halting instruction sequence bug within the first 10 s. The slightly longer time to bug can be explained by lower fuzzing throughput on the C906 (cf. Section V-A1) and by slightly more involved conditions that the broken instruction needs to satisfy, e.g., using the same registers in the encoding of the instruction. On the C908, RISCVuzz finds the undocumented CPU-halting instruction in under 15 min of iterating over the undocumented space.

For the other documented instruction findings, the time to bug is typically below 1 s. However, for some findings, fuzzing times of up to 30 s are required to reveal them.

VI. GhostWrite: Writing Arbitrary Physical Memory

In this section, we analyze GhostWrite, the arbitrary physical write primitive RISCVuzz finds on the C910. In Section VI-A, we reverse-engineer the prerequisites and microarchitectural properties, showing that GhostWrite can deterministically write attacker-chosen values to attacker-chosen physical addresses with byte granularity. Sections VI-B to VI-D demonstrate 3 end-to-end attacks as case studies, using GhostWrite for reading and writing arbitrary physical memory and executing arbitrary code with kernel- and machine-mode privileges, fully circumventing virtual memory.

A. Analysis

Listing 1 shows the assembly code of GhostWrite. A `vsetvli` instruction sets up the vector engine state. `vmv.v.x` moves the byte to be written to a vector register. The `vse128.v` instruction performs the actual write with a target address in a general-purpose register.

1) Instruction Encoding: The instruction disassembles to a vector unit-stride store instruction from the unsupported vector extension 1.0. This instruction should operate on virtual memory and store vector registers continuously to target virtual addresses stored in a general purpose register. We reduce the instruction's encoding to its minimal form and perform tests on each component of the instruction encoding.

The *source registers* encoded in the instruction work as intended, though only 1 byte is written. The *destination register* encoding also works as intended, besides interpreting the address as a physical instead of a virtual address. The encoding of the *effective element width* is 128-bit. Thus, the vector registers should be handled as 16-byte registers. Note that this encoding of 128-bit and higher is not standardized yet, but it is "expected to be used to encode expanded memory sizes" [17]. The encoded *effective element width* contrasts what we observe in practice, i.e., only one-byte stores. We further test the 256-, 512- and 1024-bit encodings of the instruction and find that they behave the same, i.e., write only one byte.

We further test the `nf` (number of fields) encoding of the instruction, which controls how many fields are stored to memory. Increasing `nf` shifts the used source vector register for the written byte by that exact amount. Thus, we only see the value of the last vector register in the group of fields. We suspect that the buggy instruction always writes to the same physical address, thereby dropping intermediate writes of other field values that should normally be continuously visible in memory. To further test this hypothesis, we measure the cycles the instruction takes to execute while varying the `nf` field and find that the instruction takes linearly more time to execute. This observation strengthens the hypothesis that multiple writes are scheduled, one for each field, but only the last one is visible since every write goes to the same address.

2) Memory Interaction: Based on the observed effects, we hypothesize that the instruction entirely circumvents the cache, directly writing to memory. We back this hypothesis using a series of experiments. We set up a base experiment in which we initialize a memory address `V`, backed by the physical memory address `P`, with a known value x_1. Next, we perform operations to ensure that the target memory address, i.e., `V`, is in a specific state before overwriting `P` with value x_2 using GhostWrite. Afterward, we check whether a memory read from `V` returns the original value x_1 or whether it was overwritten by the write gadget returning x_2. Our experiment shows that if `V` is flushed or evicted from the CPU cache before the write gadget is executed, the primitive works in 100 % of the test cases (n = 10 000). We observe that if the memory at `V` is cached in a non-dirty cache line before we use the write gadget, we need to evict or flush it from memory to make x_2 visible. Once the memory is no longer cached, we successfully read x_2 in 100 % of the tests. If the memory at `V` is cached in a dirty cache line, after flushing or evicting the cache line, the previous value x_1 remains in memory. These observations lead to the hypothesis that GhostWrite does not write through the cache hierarchy but directly to the physical memory without interfering with the cache state at all. This hypothesis explains why dirty cache lines can reset the state to x_1, as their value is written back to main memory. To further strengthen this hypothesis, we investigate the hardware performance monitor counters available on the C910. First, the counter that keeps track of dTLB misses (`mhpmcounter6`) does not count any event during the execution of the write primitive. Thus, we conclude that no virtual mapping is being resolved upon execution of the write primitive. Second, even if `V` is uncached, the counter keeping track of memory writes that miss the L1d cache (`mhpmcounter17`) and the L2d cache (`mhpmcounter21`) do not count events for our write primitive. This further strengthens our hypothesis that the write primitive does not interact with the cache hierarchy.

3) MMIO: GhostWrite can write values to any address in the physical address space, including memory-mapped input-output (MMIO). We use GhostWrite to write values of '0' and '0xff' to the first 8 bytes of the MMIO range on Board I [48]. With a voltmeter, we verify that this changes the state of the

GPIO pins. This demonstrates that the instruction bypasses any virtual memory mechanics and has full privileges.

4) Simulation: Although the C910 sources are released as openC910 [54], the vector extension cannot be enabled, as it is not part of the source. Any attempt to execute GhostWrite in the simulator fails. This aligns with discussions in the GitHub repository mentioning that the "openC910 didn't include the V extension because it wasn't officially final yet" [54].

Takeaway Even for open-source cores, the published source is not necessarily the code used for synthesizing the hardware. Thus, opaque-box testing techniques are needed even when the sources are public.

B. Page-Table Attack: From Write to Read

Our first case study turns GhostWrite into an arbitrary read gadget by rewriting page-table entries.

a) Threat Model: We assume that the physical memory and kernel configuration is unknown to the attacker. We only assume unprivileged native code execution on the target.

b) Attack: Our attack is inspired by the first privilege escalation using Rowhammer [44]. We fill the entire available physical memory with page tables by allocating large amounts of virtual memory. Specifically, we map the same file numerous times until the physical memory is exhausted. We need to spawn multiple processes to exhaust physical memory with page tables, as current RISC-V kernels use the Sv39 virtual addressing mode, where virtual addresses only use 39 bit. Filling the entire $512\,\mathrm{GB}$ of virtual address space only spawns $1 + 2^9 + \left(2^9\right)^2 = 262\,657$ page tables, which is only around $1\,\mathrm{GB}$ of physical memory. We use GhostWrite to overwrite one of the two least significant bytes of the page frame number of a potential page-table entry (PTE). Given that the memory is filled with page tables, we choose a random address in the second half of physical memory, in line with our analysis of the distribution of page tables in physical memory (Section E). If, after the modification, one of our page mappings does not map to the initially mapped file anymore, we know that we successfully overwrote a PTE. We verify that by reading from every mapping and comparing the read value to a fixed marker value. Note that we need to evict the TLB before employing this scan since the TLB might shadow our PTE modification. When no such virtual address is found, we write the PFN byte on a different physical page.

Once such a virtual page is found, we know that we have full control over a PTE and its corresponding virtual address. Thus, we can rewrite the PFN to any physical address to read the content of the address. We can also change the permission bits in the PTE to provide write access if required.

c) Evaluation: We run the attack successfully on 3 different CPUs, Board G, Board H, and Board I, with 3 different DRAM configurations, $4\,\mathrm{GB}$, $8\,\mathrm{GB}$, and $16\,\mathrm{GB}$. The attack also works from within a Docker container. We evaluate the attack on Board H with $8\,\mathrm{GB}$ of memory. We reboot the machine between each run of the attack. Our attack is successful in all 20 tries, resulting in a success rate of $100\,\%$. The attack takes 32 to $94\,\mathrm{s}$. 14 out of 20 times, overwriting the first randomly-selected address leads to a successful attack. In the other cases, up to 3 addresses have to be tried.

C. Kernel and Machine Mode Attacks: From Write to Execute

Our second case study demonstrates how an attacker can use GhostWrite to gain arbitrary code execution in the kernel and machine mode, thus elevating privileges.

a) Threat Model: We assume that the attacker has unprivileged native code execution on the target. We assume the attacker knows the physical memory layout of the kernel. This assumption is viable in practice, as the physical memory layout is highly predictable. Alternatively, an attacker can employ an arbitrary read gadget (e.g., Section VI-B) to scan for the kernel. For escalating privileges to root, our end-to-end exploit assumes the presence of a setuid binary such as `sudo` or `su` that uses the `getuid` syscall to determine whether a user is already root and, therefore, does not need to authenticate. To gain code execution in machine mode, we assume the presence of OpenSBI with a known version and physical memory location.

b) Attack: To gain code execution in the kernel, the attacker uses GhostWrite to overwrite the code of a system call handler. Then, the attacker triggers the corresponding syscall to execute the injected payload. Since virtual memory is completely bypassed by GhostWrite, the same principle can be used from containers or virtualized environments to attack a known host. In our proof-of-concept implementation, the attacker overwrites the `getuid` syscall to always return '0'. On Linux systems, the user ID '0' is reserved for the privileged root user. Then, the exploit executes the `su` setuid binary. If `getuid` returns '0', `su` assumes a user is already root and skips authentication, leading to privilege escalation to the root user. Once a privileged shell is obtained, the exploit uses GhostWrite to restore the original `getuid` to not break any legitimate functionality of other applications and to purge any traces of the attack.

For code execution in machine mode, our proof-of-concept overwrites parts of the function `sbi_ecall_base_handler`, which handles `ecalls` in OpenSBI. Since our previous attack enables code execution in the kernel, we assume an attacker can trigger arbitrary SBI `ecalls`. In our proof of concept, we patch the ecall handler for `SBI_EXT_BASE_GET_MVENDORID` to return 42 and verify the return value using a kernel module. In a real-world scenario, an attacker could place arbitrary code at any physical address and patch a jump to their payload into the `ecall` handler.

c) Evaluation: We evaluate all attacks successfully on the 3 C910 boards (G, H, and I). The attacks take less than $1\,\mathrm{s}$, since the addresses are known and only a physical write with GhostWrite is needed. The OpenSBI binaries, implementing the machine-mode functionality for RISC-V systems, are mapped at physical address '0' on the C910-based systems. The kernel code and data follow at `0x200000`. We verify that this layout is stable across reboots. As GhostWrite always works (Section VI-A), and the physical layouts of kernel and OpenSBI do not change, the exploit's success rate is $100\,\%$.

D. Ineffective Faults: From Write to Indirect Read

In our third case study, we demonstrate how GhostWrite can be combined with ineffective fault attacks [9], [11], [65]

to read secrets such as cryptographic keys indirectly. We demonstrate this by recovering TLS signing keys via an oracle that only exposes whether a signature was successful.

a) Threat Model: We assume the victim runs a server using OpenSSL with TLS 1.3 on the same machine as the attacker. The attacker acts as a malicious client, repeatedly connecting to the TLS server and triggering the handshake process. This scenario is practical when the attacker runs inside virtual machines or containers. The attacker can easily exhaust the physical memory, as described in Section VI-B. Consequently, we assume that the attacker can influence the physical address of the private key and one of the RSA-CRT parameters using memory massaging [30].

b) Attack: We target the TLS signing step in the OpenSSL library, explicitly focusing on scenarios where the server selects the RSA algorithm for digital signatures. In TLS 1.3, the server must sign a hash that encapsulates the handshake messages exchanged with the client before the handshake is finished. With such a signature, the client can verify the server's authenticity. OpenSSL's TLS implementation employs the RSA-CRT (Chinese Remainder Theorem) optimizations for efficiency. If faults happen during the signing, leading to a wrong signature, the server attempts to re-sign using the traditional textbook RSA method. This additional verification step effectively mitigates Bellcore attacks [2], [4].

To introduce a faulty signature, an attacker has to corrupt both of the two signing algorithms. The attacker first corrupts the parameters used for RSA-CRT employing GhostWrite. That forces the server to start using the textbook RSA algorithm. Then, the attacker corrupts the private key byte-by-byte in physical memory. For each byte, the attacker attempts to establish an SSL connection 256 times, each with a different possible value written to that specific byte. Only if the written value is correct, the attacker observes a successful SSL connection. The attacker uses this feedback as a side channel to recover the private key. Similarly, the attacker can recover the parameters used for RSA-CRT by switching to iteratively modifying the content of the parameters.

Using statistical ineffective fault attacks to recover the private key typically involves a prolonged correlation analysis [11], due to the difficulty of analyzing the biased distribution of each fault. In contrast, our attack uses a physical write that allows the attacker to modify the private key byte by byte. This capability significantly simplifies the attack by reducing the complexity of determining the correct bits of the key.

c) Evaluation: We successfully mount the attack on Board H. The victim uses an unmodified OpenSSL version 3.0.9. We observe a leakage rate of $1.10\,\mathrm{bit/s}$ over 10 trials. On average, it takes the attacker 30 minutes to fully recover the private 2048-bit RSA key or half of the time to recover the 1024-bit parameters of RSA-CRT.

VII. CPU-Halting Instruction Sequences

In this section, we analyze the instruction sequences RISCVuzz finds for halting the C906 and C908, requiring a hard reset. We analyze the sequences on hardware, reproduce the findings in the simulation of the C906, and present an end-to-end denial-of-service attack from within Docker.

```
1 th.lbib t0, (t0), 0, 0
2 frcsr t0
3 li t0, 0
```

Listing 2: The interaction of the `th.lbib` instruction with the same register as source and destination, a CSR read, and an unrelated operation on the register halts the C906.

A. Analysis

a) C906: Listing 2 shows the instruction sequence halting the C906. The core of the sequence is the `th.lbib` instruction from the custom `XTheadMemIdx` extension. This vendor extension provides additional memory operations such as increment-address-before-load (`th.lbib`). The halt occurs in combination with using the same register for source and destination operand, a subsequent CSR read, and any subsequent interaction with the register provided to the instruction. In the example code, we read a CSR using the unprivileged `frcsr` instruction. However, any other instruction reading a CSR, such as `rdcycle`, can also be used. While the example uses the load immediate instruction (`li`), the last instruction can be any instruction interacting with the used register. Unrelated instructions can be part of the sequence if they do not read from or write to the used register (`t0` in the example).

The T-Head vendor extension docs [56] mention that using the same register as source and destination is not a valid encoding. Compilers that support the vendor extension do not support compiling assembly code with this faulty encoding. However, without the subsequent operations involving a CSR and another operation on the register, no CPU halt occurs. We further discover that 13 other instructions from the `XTheadMemIdx` extension are vulnerable (cf. Listing 3 in Section B) in the same way. Surprisingly, the variants with 3 source registers are not vulnerable.

Takeaway CPU vulnerabilities exist for both single instructions and instruction sequences.

b) C908: The instructions discovered by RISCVuzz on the C908 correspond to the vector mask store/load instructions `vsm.v` and `vlm.v`. Setting any of the bits 29 to 31 in the encoding of these instructions crashes the machine. Note that these bits should be all unset, i.e., zero, when the instruction is assembled correctly. Other bits seem unaffected.

Takeaway Testing the entire possible encoding space is necessary, as vulnerabilities are in the documented and undocumented range.

B. Bug Reproduction in Simulator

In contrast to the other tested CPUs, the source code of the C906 is available, and the source contains the same vulnerability as the hardware CPU. Thus, we can also reproduce the vulnerability in the simulation of the Verilog source. We run the source from the official T-Head repository of the C906 [53] using the ICARUS Verilog compilation system [62]. The simulation machine is an Intel Core i9-13900K with 16 GB RAM running Ubuntu 22.04.

We reduce the instruction sequence to a minimal 24 B bare-metal binary containing only 6 instructions. Running this sequence reliably stops the simulator with the error message that the CPU is stuck and no instructions are retiring anymore. This happens after 11.5 µs CPU time. It takes the simulation 2.5 min to reach this point.

C. Case Study

To assess the impact of the vulnerability, we evaluate in which contexts it can be executed to halt the CPU. The straightforward scenario is an *unrestricted native environment* as has been used by RISCVuzz. Executing the instruction sequence as a privileged or unprivileged user immediately results in the CPU being halted. We verify the C906 behavior on two different boards, Board D and E, using two different operating systems, Debian 11 and Debian 12. Additionally, we verify the C908 behavior on Board F with Debian 13. While executing the instruction in machine mode also halts the CPU, there is no realistic threat model where this is relevant. However, attackers can also use the instruction sequence in more restricted environments. We verify that executing the instruction in an unprivileged Docker container also halts the CPU. Thus, sandboxing mechanisms that work on the operating-system level cannot prevent an attacker from halting the CPU. Furthermore, given that all involved instructions are unprivileged instructions, we also expect that sequence to work from a virtual machine. Unfortunately, we cannot verify that, as no current hypervisor supports the C906 or C908.

VIII. Mitigations

In this section, we discuss mitigations for GhostWrite (Section VI) and the CPU-halting sequences (Section VII).

GhostWrite. Disabling the vector extension is a viable mitigation for GhostWrite. We use a kernel module to verify this mitigation on the C910. The CPU throws an illegal instruction exception when executing the instruction [60], making the gadget unusable for an attacker.

We benchmark the impact of this mitigation on standard memory operations such as `memcpy` and `memset`. We use the RISC-V vector benchmarking suite `rvv-bench` [3] on Board H. We compare the fastest vector implementation of `memcpy` and `memset` to the fastest of glibc and musl libc. We observe a performance hit of up to 33 % for `memcpy` and 8 % for `memset`. Benchmarking the mitigation on a full-system level is currently not possible, since no distribution uses the vector extension in the kernel and standard libraries.

C906 Halting Sequence. There is no mitigation for the C906 halting instruction sequence that can be used for DoS from an unprivileged process. Since no special condition is needed to execute the C906 halting instruction sequence, we argue that the only option for mitigating the vulnerability is to disable one of the instructions. Unfortunately, the T-Head vendor extension that includes the broken instructions cannot be disabled: "The `th.sxstatus.THEADISAEE` bit is not expected to be cleared. The behavior of clearing this bit is undefined" [56]. We verify that we cannot clear this bit from a kernel module.

Takeaway Optional hardware features should have the capability to be deactivated.

C908 Halting Instructions. The C908 halting instructions are in the vector-extension range. Disabling the vector extension fortunately prevents exploitation. Recompiling the Linux kernel without vector support leads to an illegal-instruction exception when executing the instruction, mitigating the DoS of these instructions.

We perform the same benchmark as in Section VIII since the mitigation is the same. We run `rvv-bench` on Board F and find that the performance of `memcpy` and `memset` decreases by up to 77 % and 2 %, respectively.

IX. Related Work

Undocumented Instructions. Armshaker [52] is an approach that iterates over the entire 32-bit ARM instruction space to find undocumented instructions. Similarly, Dofferhoff et al. [12] propose a tool to find undocumented instructions for RISC architectures like ARMv8 and RISC-V using disassemblers as a ground truth. Both works discovered multiple emulator bugs and inconsistencies in the respective ISA standards. DifuzzRTL [23] and Morfuzz [64] also fuzz undocumented instructions on RISC-V. However, in contrast to RISCVuzz, DifuzzRTL and Morfuzz require a perfect simulator to detect misbehaviors in the executed code. Sandsifter [14] shows that it is feasible to search for undocumented x86 instructions by exploiting a side channel to infer up to which byte an instruction was successfully decoded, dealing with the 15-byte instruction space. In contrast to previous work, RISCVuzz does not require any ground truth and can exhaustively test the entire instruction space.

Differential CPU Fuzzing. Differential fuzzing is a well-known software fuzzing technique that was recently applied to CPU fuzzing. Tavis Ormandy found critical security vulnerabilities by comparing code generated by their fuzzer against a serialized variant of the code [35], [36]. For this so-called Oracle Serialization, the initial code is modified by adding memory fences between the individual instructions. Such techniques make it possible to find differences that depend on speculative or out-of-order execution. Bugs that do not depend on such optimizations are invisible to this technique. For example, GhostWrite behaves the same when fences are added and hence cannot be detected by this approach. Further, cross-vendor or cross-generation bugs cannot be discovered with this technique, since results are only compared to other cores on the same CPU. RISCVuzz finds these classes of bugs too, as it compares the behavior against different CPUs from potentially different vendors. SiliFuzz [45] targets x86 CPUs with the goal of finding electrical defects on single cores instead of bugs. As SiliFuzz does not compare different CPUs but different cores on one CPU, the found bugs of SiliFuzz and RISCVuzz are different. For our tested CPUs, all cores on the same CPU behave the same, as the differences are due to implementation errors of the CPU and not due to electrical defects of single cores. Qin et al. [37] and Jiang et al. [24] compare CPUs with emulators and disassemblers to build stealthy malware by exploiting different runtime behaviors. In contrast to our paper, they discover and exploit bugs in software, not in CPUs.

Model Fuzzing. TheHuzz [26] and DifuzzRTL [23] are fuzzers that target the register-transfer level (RTL) model of CPUs. The benefit of targeting the CPU's RTL is that these fuzzers can be used during development and that emulation of the cores can provide coverage information to guide the fuzzer's search. Solt et al. [51] improve upon that approach by generating more complex instruction sequences to improve fuzzing throughput and discover bugs with more complex conditions. The major drawback of all RTL-based fuzzers is that they require a complete RTL to work in the first place. In contrast, RISCVuzz targets the orthogonal problem of finding bugs in opaque-box hardware CPUs with the advantage of faster fuzzing throughput and testing the actual deployed CPUs. Due to its speed, RISCVuzz fully explores all undocumented instructions with a bottom-up approach (Section IV-C1) rather than a mutaion approach. Guidance [25] or fuzzing by proxy [45] could also be used for RISCVuzz. However, even without this guidance, RISCVuzz produces many results, leaving this guidance for future work.

X. Discussion

In this section, we discuss the impact of our findings, coverage and ground truth, an outlook on future problems, and the need for compliance testing.

a) Impact: While there are still not many 64-bit RISC-V boards on the market, they already gain traction. Scaleway provides cloud instances with the TH1520 SoC [42], which contains a C910 CPU. We verified that the used CPU is indeed vulnerable, and we reported our findings to Scaleway. The Shandong University in China also has a RISC-V cluster using a variant of the C910 CPU [46]. Unfortunately, we do not have access to this system to test if this C910 variant is also vulnerable. Mitigating GhostWrite is only possible by disabling the entire vector extension, resulting in a reduced feature set and lower performance for specific workloads. Worse, the CPU halting sequence on the C906 has no mitigation, making these CPUs essentially unsuitable for running untrusted code or deployment in multi-user systems.

We assume that with a microcode layer, as on x86, GhostWrite could be mitigated. An x86 microcode update can hook and patch instructions [5], which could have been used to hook the broken vector instruction and simply raise an illegal-instruction exception. Given the increasing complexity of RISC-V CPUs, we advocate such a microcode layer on RISC-V to have the possibility of mitigating CPU vulnerabilities.

b) Ground Truth: RISCVuzz can be used in scenarios where no ground truth, i.e., an emulator or another reference implementation, is available because it does not require a golden model. Cases where no ground truth is available include (custom) ISA extensions like T-Head's custom draft vector extension [57], CPUs where the source code is not (fully) available like with GhostWrite, or where no source code nor documentation about a feature is available at all as with the CPU-halting instructions on the C908. Even if emulators are available, they might be inconsistent [24] or based on a different code base, as we see for the C910.

c) Coverage & Sequence Complexity: Generating complex instruction sequences that increase coverage over time is difficult for opaque-box fuzzing since we have no adequate feedback channel. Similarly, evaluating diversity graphs of our sequences is not possible, as we have no coverage information. However, the results of MorFuzz [64] suggest that more complex sequence generation approaches might be beneficial. Note that these problems are inherent to opaque-box fuzzing and not a particular weakness of our approach. However, even with our simple sequence generation, we uncover most of the bugs in seconds due to the high throughput of running them on hardware. Recent work on software fuzzing also suggests that even simple input generation can be efficient [33]. We leave improving upon our simple sequence generation or using hardware side channels for coverage for future work.

d) Compliance: The findings from RISCVuzz demonstrate numerous architectural differences in instructions across CPU vendors and even across CPUs of the same vendor. These findings also suggest that several of these differences violate the ISA specification, making it difficult to write applications that run correctly on all RISC-V CPUs. Thus, we advocate an extensive compliance-testing framework for RISC-V. While there is an architectural test suite [20], many parts are not covered, including the vector extensions we exploited.

e) Outlook: At the time of writing, only T-Head and SiFive have commercially available off-the-shelf machines with general-purpose 64-bit RISC-V CPUs. Still, even with this limited selection of vendors and CPUs, RISCVuzz finds a large number of bugs and inconsistencies. In the future, we expect to see more vendors building CPUs based on custom designs. Combined with the unregulated use of the ISA and the possibility of creating custom vendor extensions, we expect this state to worsen. Architectural inconsistencies will become especially relevant for trusted-execution environments and virtualization, as current vendor extensions might not consider the future-proofness of their extensions concerning potentially different privilege levels. Similarly, with RISC-V support in Linux and Android, vendor customizations might undermine security guarantees or lead to unstable systems.

XI. Conclusion

In this paper, we introduced RISCVuzz, a differential CPU fuzzing framework for RISC-V hardware CPUs for automatically discovering architectural CPU bugs. RISCVuzz compares the architectural results of instruction sequences without relying on CPU source code or any emulator. RISCVuzz discovered 3 severe security vulnerabilities and numerous other bugs on 5 different CPUs. On the T-Head C910, RISCVuzz discovered GhostWrite, an instruction sequence that allows unprivileged attackers to write arbitrary values directly to physical memory, entirely circumventing virtual memory and its protection. We demonstrate that GhostWrite can also be used to read memory and to inject attacker code into kernel and machine mode. Further, we investigated two "halt-and-catch-fire instructions" on two different CPUs, the T-Head XuanTie C906 and T-Head XuanTie C908, and showed how they lead to unprivileged denial of service. RISCVuzz discovered most bugs and vulnerabilities within seconds, showing the efficacy of our post-silicon fuzzing approach. We outperform state-of-the-art RTL-based fuzzers in instruction execution by orders of magnitude, making it a valuable extension to these fuzzers.

ACKNOWLEDGMENT

This work was supported in part by Semiconductor Research Corporation (SRC) Hardware Security Program (HWS) and by a Google Research Scholar award. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Berkley*, 2016.

[2] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete results and practical countermeasures," in *CHES*, 2002.

[3] O. Bernstein, "rvv-bench: Risc-v vector benchmark," 2023. [Online]. Available: https://github.com/camel-cdr/rvv-bench

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," 2001.

[5] P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, and M. Schwarz, "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode," in *WOOT*, 2023.

[6] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture," in *USENIX Security*, 2022.

[7] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security*, 2019, extended classification tree and PoCs at https://transient.fail/.

[8] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Tech. Rep., 2015.

[9] C. Clavier, "Secret external encodings do not prevent transient fault analysis," in *CHES*, 2007.

[10] R. R. Collins, "The Pentium F00F Bug," 1998. [Online]. Available: http://www.rcollins.org/ddj/May98/F00FBug.html

[11] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography," in *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[12] R. Dofferhoff, M. Göebel, K. Rietveld, and E. Van Der Kouwe, "IScanU: A portable scanner for undocumented instructions on risc processors," in *International Conference on Dependable Systems and Networks*, 2020.

[13] E. Dolstra, A. Löh, and N. Pierron, "Nixos: A purely functional linux distribution," in *Journal of Functional Programming*, 2010.

[14] C. Domas, "Hardware Backdoors in x86 CPUs," *Black Hat US*, 2018.

[15] L. Foundation. (2023). [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/tree/arch/riscv/boot/dts/thead

[16] R.-V. Foundation. (2019) Risc-v "v" vector extension 0.7.1. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1

[17] ——. (2021) Risc-v "v" vector extension 1.0. [Online]. Available: https://wiki.riscv.org/display/HOME/Ratified+Extensions

[18] ——. (2021) Risc-v "zfh" and "zfhmin" standard extensions for half-precision floating-point, version 1.0. [Online]. Available: https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions

[19] ——, "riscv-opcodes," 2022. [Online]. Available: https://github.com/riscv/riscv-opcodes

[20] ——, "RISC-V Architecture Test SIG," 2023. [Online]. Available: https://github.com/riscv-non-isa/riscv-arch-test

[21] M. Frysinger, "vdso(7) — linux manual page," 2024.

[22] L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs," in *S&P*, 2023.

[23] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *S&P*, 2021.

[24] M. Jiang, T. Xu, Y. Zhou, Y. Hu, M. Zhong, L. Wu, X. Luo, and K. Ren, "Examiner: Automatically locating inconsistent instructions between real devices and cpu emulators for arm," in *ASPLOS*, 2022.

[25] N. Kabylkas, T. Thorn, S. Srinath, P. Xekalakis, and J. Renau, "Effective processor verification with logic fuzzer enhanced co-simulation," in *MICRO*, 2021.

[26] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities," in *USENIX Security Symposium*, 2022.

[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[28] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.

[29] Krimsky, "RISC-V Single Board Computers," 2023. [Online]. Available: http://krimsky.net/articles/riscvsbc.html

[30] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P*, 2020.

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[32] W. M. McKeeman, "Differential testing for software," 1998.

[33] B. P. Miller, M. Zhang, and E. R. Heymann, "The relevance of classic fuzz testing: Have we solved this one?" *IEEE Transactions on Software Engineering*, vol. 48, 2022.

[34] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[35] T. Ormandy, "Reptar," 2023. [Online]. Available: https://lock.cmpxchg8b.com/reptar.html

[36] ——, "Zenbleed," 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[37] S. Qin, C. Zhang, K. Chen, and Z. Li, "iDEV: Exploring and exploiting semantic deviations in arm instruction processing," in *ISSTA*, 2021.

[38] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults," in *AsianHOST*, 2019.

[39] RISC-V Collaboration, "riscv-gnu-toolchain," 2024. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[40] RISC-V Foundation, "RISC-V Exchange," 2023. [Online]. Available: https://riscv.org/exchange/

[41] RISC-V International, "Operating Systems," 2024. [Online]. Available: https://wiki.riscv.org/display/HOME/Operating+Systems

[42] Scaleway. (2024) The world's first RISC-V servers available in the cloud. [Online]. Available: https://labs.scaleway.com/en/em-rv1/

[43] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *USENIX Security*, 2022.

[44] M. Seaborn, "Exploiting the DRAM rowhammer bug to gain kernel privileges," March 2015, retrieved on June 26, 2015. [Online]. Available: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[45] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," *arXiv:2110.11519*, 2021.

[46] A. Shah. (2023) China deploys massive risc-v server in commercial cloud. [Online]. Available: https://www.hpcwire.com/2023/11/08/china-deploys-massive-risc-v-server-in-commercial-cloud/

[47] SiFive, "HF105 Datasheet," 2022. [Online]. Available: https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf

[48] Sipeed, "Sipeed wiki," 2021. [Online]. Available: https://wiki.sipeed.com/en/index.html

[49] ——, "RISC-V 64bit chip (C910) run Android 10," 2022. [Online]. Available: https://twitter.com/SipeedIO/status/1457529282134089734

[50] ——. (2023) Lichee Console 4A. [Online]. Available: https://sipeed.com/licheepi4a

[51] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: Cpu fuzzing via intricate program generation," 2024.

[52] F. Strupe and R. Kumar, "Uncovering hidden instructions in Armv8-A implementations," in *Hardware and Architectural Support for Security and Privacy*, 2020.

[53] T-Head, "openC906," 2021. [Online]. Available: https://github.com/T-head-Semi/openc906

[54] ——, "openC910," 2021. [Online]. Available: https://github.com/T-head-Semi/openc910

[55] ——, "C906," 2022. [Online]. Available: https://www.t-head.cn/product/c906

[56] ——, "T-Head Extension Spec," 2022. [Online]. Available: https://github.com/T-head-Semi/thead-extension-spec

[57] T-Head, "Xtheadvector," 2022. [Online]. Available: https://github.com/XUANTIE-RV/thead-extension-spec/blob/master/xtheadvector/intrinsics.adoc

[58] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[59] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Vol. I: Unprivileged ISA, Version 20191213," 2019.

[60] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203," 2021.

[61] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v compressed instruction set manual, version 1.7," *EECS Department, University of California, Berkeley*, 2015.

[62] S. Williams, "Icarus verilog," 2024. [Online]. Available: https://steveicarus.github.io/iverilog/

[63] Xcalibyte, "Roma Laptop Pre-order," 2022. [Online]. Available: https://xcalibyte.com.cn/en/roma-preorder/

[64] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "{MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *USENIX Security*, 2023.

[65] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Transactions on computers*, vol. 49, 2000.

[66] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

Appendix A
Details on Used Boards

Table III extends Table I by the board models, memory configuration, kernel, and OS version.

Appendix B
C906 CPU-halting Instructions

Listing 3 lists the other broken instructions from the `XTheadMemIdx` vendor extension on the C906. Any of these instructions can be used instead of the `th.lbib` instruction in Listing 2 to halt the CPU.

Appendix C
Distribution of Executed Instructions in Sequence

To further reason about the scaling of increasing the sequence length (cf. Figure 5), we collect the distribution of the number of executed instructions per sequence length. $\sim 40\,\%$ of the executed sequences stop right at the first instruction. Equivalently, this means that $\sim 40\,\%$ of instructions in the base ISA throw a signal when run in RISCVuzz. Consequently, every further instruction in the sequence decreases the probability of a longer sequence by $\sim 40\,\%$. This means the percentage of actually executed sequence lengths is roughly given by the function $prob\_exec(n) = 0.4^n$. That further points out why increasing the sequence length leads to diminishing returns.

```
th.lbib   th.lbia   th.lwuib  th.lbuib
th.lwia   th.ldia   th.lwib   th.lbuia
th.lhuia  th.lhia   th.ldib   th.lhib
th.lwuia  th.lhuib
```

Listing 3: List of instructions from the `XTheadMemIdx` extension that can be used in Listing 2 to halt the C906 CPU.

Appendix D
GhostWrite Sample Reproducer

Listing 4 shows an example of a reproducer file generated by RISCVuzz when comparing the results of vector instructions between C906 and C910, which both implement the 0.7.1 draft vector extension. The C906 generates a fault, while the instruction executes just fine on the C910. This hints at GhostWrite.

```
# signum differs
# si_addr differs
# si_pc differs
# si_code differs
#
# base:  C910 (lab46)
# signum:                          OK
# --------------------------------
# other: C906 (lab50)
# signum:                     SIGSEGV
# si_addr:      0x8000000000000000
# si_pc:                  0xe100178
# si_code:                      0x1

instr_seq:
- 0x5201f0a7
dis: ''
dis_opcodes:
- vse1024.v
regs:
  gp:
    gp: 0x8000000000000000
flags:
- "-DVECTOR"
```

Listing 4: Example of a reproducer that hints at GhostWrite. The C910 executes the vector-store instruction just fine, while the C906 generates a fault.

Appendix E
Physical Page Table Distribution

In Section VI-B, we use GhostWrite to overwrite page frame numbers in physical memory to transform it into an arbitrary physical read primitive. We fill the entire physical memory with page tables so that the probability of hitting such a page table when writing at a random address in memory is high. In this section, we experimentally verify this.

We create $1\,500\,000$ last-level page tables, filling up the entire $8\,\mathrm{GB}$ of memory on Board H by mapping a file repetitively to memory in multiple processes. We then record

TABLE III: Overview of tested RISC-V boards. We use CPUs from 2 vendors with varying extensions.

ID	Board Model	CPU	CPU Vendor	Relevant Extensions	Memory	OS	Kernel
A	BeagleV Fire	U54	SiFive	-	1.5 GB	Ubuntu 23.04	6.1.33
B	StarFive VisionFive2	U74	SiFive	-	8 GB	Ubuntu 22.04.1	6.5.0
C	Sipeed Nezha	C906	T-Head	`v0p7`, `zfh`, `XTheadMemIdx`	1 GB	Debian 13	5.14.0
D	Lichee RV Dock				512 MB	Debian 11	5.4.61
E	Lichee RV Dock				512 MB	Debian 12	5.14.0
F	CanMV Kendryte K230	C908	T-Head	`v`, `zfh`, `XTheadMemIdx`	512 MB	Debian 13	5.10.4
G	BeagleV Ahead	C910	T-Head	`v0p7`, `zfh`, `XTheadMemIdx`	4 GB	Ubuntu 23.04	5.10.113
H	LicheePi4A				8 GB	Debian 12	5.10.113
I	LicheePi4A				16 GB	NixOS	5.10.113
J	Milk-V Meles				8 GB	Debian 12	5.10.113

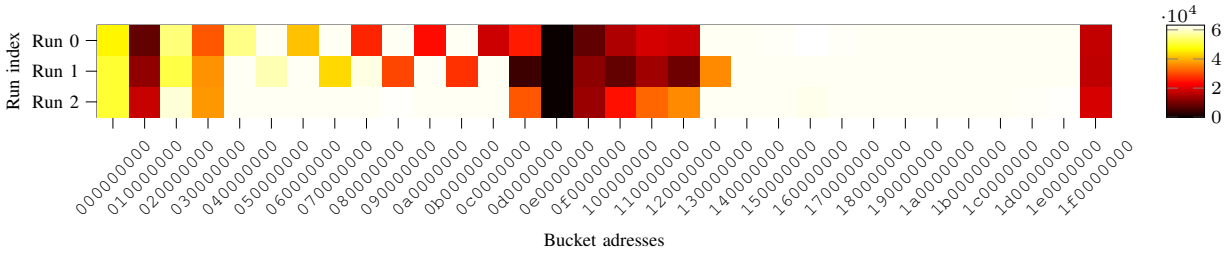

Fig. 6: Physical distribution of page tables when filling 8 GB of memory with page tables over 3 runs. Accumulated into 32 buckets of 256 MB.

the physical addresses of these last-level page tables in 3 runs. Figure 6 visualizes the distribution of the collected addresses. Page tables are nearly uniformly distributed over the entire physical memory, with a gap in the middle of memory and at the start, where kernel and OpenSBI reside.