RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs

Fabian Thomas
fabian.thomas@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Eric García Arribas
eric.garcia-arribas@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Lorenz Hetterich
lorenz.hetterich@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Daniel Weber
daniel.weber@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Lukas Gerlach
lukas.gerlach@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Ruiyi Zhang
ruiyi.zhang@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Michael Schwarz
michael.schwarz@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Abstract

The open and extensible RISC-V instruction set has enabled many new CPU vendors and implementations, but most commercial CPUs are closed-source, significantly hindering vulnerability analysis—especially for bugs exploitable from unprivileged user space.

We present RISCover, a user-space framework for detecting architectural vulnerabilities in closed-source RISC-V CPUs. It compares instruction-sequence behavior across CPUs, identifying deviations without source code, hardware changes, or models, and achieving orders-of-magnitude speedups over RTL-based methods. Unlike prior work, RISCover runs user code on Linux directly on real hardware, exposing vulnerabilities exploitable by unprivileged attackers. Evaluated on 8 off-the-shelf CPUs from 3 different vendors, it uncovers 4 previously unknown vulnerabilities.

Notably, GhostWrite lets unprivileged code write chosen bytes to physical memory, enabling arbitrary data leakage and full machine-mode execution, while 3 unprivileged "halt-and-catch-fire" bugs halt CPUs and misaligned zero-stores silently corrupt data. Our results highlight the pressing need for post-silicon fuzzing techniques. RISCover complements existing RTL-level fuzzers by enabling rapid and automated security analysis of closed-source CPUs.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

CPU Fuzzing; RISC-V; Architectural CPU Vulnerabilities

ACM Reference Format:
Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. 2025. RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765141

1 Introduction

RISC-V is still a young instruction set architecture (ISA). While the first CPUs were mainly soft cores designed for emulators and FPGAs [6, 12], there are now several hardware cores available on the market [62, 63, 68, 70, 71]. These cores are already used in laptops [66, 81], servers [56, 60], workstations [46], mobile phones [65], cars [37], and gaming consoles [66]. T-Head reports that more than 4 billion of their CPUs have been sold [40]. The available CPUs implement the base ISA and typically additional ratified extensions, such as compressed instructions or floating-point support [20–22, 79]. Ongoing work focuses on thoroughly testing these extensions to ensure CPUs implement them correctly [24]. However, RISC-V also supports vendor-specific custom extensions, such as the T-Head XuanTie extensions (`XTheadMemIdx`, `XTheadVec`), which are supported by major compilers [26].

Unfortunately, most high-end cores, such as the T-Head XuanTie C908 and C920, SiFive U74 and P550, or SpacemiT X60, lack publicly available source code. Thus, previous approaches focusing on finding vulnerabilities at the register-transfer level (RTL) [34, 39, 67] are not applicable for researchers or third parties. Prior work often concentrates on functional correctness rather than user-exploitable security issues and requires hours to days to identify problems.

Many of these problems can only be triggered from machine or supervisor mode and are not relevant for the security of shared systems, such as the cloud or mobile devices. In contrast, running in user mode acts as an automated filter for bugs exploitable by unprivileged users. Moreover, RTL-fuzzing typically uses minimal SoCs, e.g., without DRAM [34, 39, 67], missing vulnerabilities that arise from interactions of system components. The required analysis time further increases with CPU complexity, as simulation requires more resources. Additionally, a recent study from Bölcskei et al. [8] shows that structural coverage-based fuzzers are outperformed by black-box fuzzers in bug discovery speed.

In this paper, we ask the following research question:

Can we automatically and efficiently discover architectural CPU vulnerabilities which are triggerable from user space on closed-source RISC-V CPUs across different vendors?

To answer this question, we present our proof-of-concept tool RISCover (RISC-V vulnerability discovery), a differential CPU fuzzing framework that operates from user space to analyze RISC-V CPUs for *architectural security vulnerabilities*. RISCover exploits the inherent property of a well-defined ISA that the *architectural* result of *every* deterministic instruction must be the *same across different CPUs* if supported. RISCover crafts and executes unprivileged instruction sequences on different RISC-V CPUs from Linux user space. Any deviation from the majority vote concerning the output or side effect, i.e., system crash, is reported as a potentially misbehaving sequence. As we cannot run bare-metal code or perform hardware resets from user space, RISCover requires a custom sandbox that safely executes instruction sequences and captures exceptions. RISCover works on any system with Linux, making it versatile and easily deployable.

The reported sequences are inspected to determine if they represent vulnerabilities. For example, denial-of-service attacks are considered medium-severity [1–4] when executable from user space. Sequences can be as simple as single instructions to complex chains of thousands [67]. We rely on fast, short, weighted random sequences, biasing instructions toward rarely used ones based on our analysis of real-world RISC-V binaries. Due to RISCover's design and applicability to hardware cores, it executes up to 59 205 instructions per second on a single CPU core. This allows RISCover to find bugs and vulnerabilities within *seconds to minutes*, significantly outperforming existing methods requiring hours to days. In addition to new bugs, we rediscover 22 bugs found by previous work.

We evaluate RISCover on 8 RISC-V CPUs from 3 vendors: T-Head XuanTie C906/C908/C910/C920, SiFive U54/U74/P550, and SpacemiT X60. This test set consists of all currently consumer-available RISC-V CPUs running a 64-bit Linux operating system. Additionally, we evaluate RISCover on 4 emulators. In total, RISCover discovers 4 architectural vulnerabilities and numerous bugs. Notably, RISCover discovers GhostWrite, an unprivileged instruction sequence on the C910 and C920 that allows a user-space attacker to write controlled bytes to chosen *physical memory* locations, including attached devices. The other 3 flaws belong to the class of "halt-and-catch-fire" CPU vulnerabilities [15], enabling unprivileged denial-of-service attacks, one each on the C906, C908, and X60. RISCover also discovers a firmware bug in the M-mode code of the BeagleV Fire (SiFive U54) that makes misaligned zero-stores not write 0 but -1, leading to silent data corruption. For the C906 and C910 (the only cores with source availability), we verify that the state-of-the-art RTL fuzzer Cascade [67] does not find them by design, as the vulnerability is either not in the published source or outside its sequence generation. In contrast, RISCover rediscovers 22 out of 23 bugs that Cascade found in softcores [67]. Additionally, we discover many other architectural bugs in CPUs and emulators, most within seconds of fuzzing. These include undocumented instructions, address-handling bugs, decoder bugs, ISA incompatibilities, fault-reporting issues, and segmentation faults in QEMU.

We demonstrate the security impact of our findings by building an end-to-end attack with GhostWrite that lets unprivileged users read and write arbitrary memory, including machine-mode code and devices mapped via MMIO. We demonstrate that this vulnerability can also be exploited by unprivileged users in the cloud by showing it on Scaleway TH1520 instances. For the instructions that halt the CPU, we show that they can be used by any unprivileged application and also work from inside Docker containers.

Our results provide insights into the current state of hardware RISC-V CPUs. Vendor extensions can easily lead to bugs and exploitable vulnerabilities, likely as existing stress tests only focus on the ratified ISA [24] and valid instruction encodings [14]. An additional insight is that even for open-source cores, such as the C910, the hardware implementation differs from the released source. We emphasize that our approach is orthogonal to RTL fuzzing, covering scenarios RTL fuzzing cannot. While pre-silicon verification methods and formal analysis remain essential, recent vulnerabilities (e.g., Zenbleed [53], Reptar [52], ÆPICLeak [10], or CacheWarp [83]) in mature CPUs from Intel and AMD underscore the necessity for robust post-silicon fuzzing. Even if the source code is available, RISCover might find bugs introduced by the synthesis, which are therefore invisible in the RTL. Hence, we argue that post-silicon fuzzing is a valuable complement to existing pre-silicon fuzzers [34, 39, 57, 67]. 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 [9, 48, 52, 55, 83].

Contributions. We summarize our contributions as follows.

- We present RISCover, a user-space differential CPU-fuzzing framework for finding architectural CPU vulnerabilities on closed-source RISC-V CPUs.
- We automatically test 8 consumer CPUs across 3 vendors and discover 4 new user-space-triggerable vulnerabilities—including a physical memory write primitive (GhostWrite) and 3 denial-of-service sequences.
- We additionally discover numerous architectural bugs and undocumented instructions with unclear security impact.
- We demonstrate full system compromise via physical memory writes and privileged code execution, even in the cloud.

Responsible Disclosure. We reported all the security-critical vulnerabilities to the manufacturers, i.e., T-Head, SpacemiT, BeagleBoard, SiFive and QEMU. GhostWrite is tracked as CVE-2024-44067 and is mitigated in Linux from version 6.14 onward by disabling the vector extension. We also reported GhostWrite to Scaleway since they offer C910-based bare-metal machines in the cloud. Scaleway provided instructions to customers for manually rolling out kernel patches that disable the vector extension, mitigating GhostWrite.

Availability. https://github.com/cispa/RISCover-artifacts.

2 Background

This section covers the relevant background required for the remainder of the paper. We introduce the RISC-V instruction set architecture alongside some of its core features. We discuss CPU vulnerabilities and how fuzzing can be used to find them.

2.1 RISC-V

RISC-V is an open instruction set architecture (ISA) consisting of a mandatory core instruction set and optional extensions [77, 78]. An example of an extension is the vector extension [21] that is implemented in the T-Head XuanTie C908 and SpacemiT X60. In addition, vendor-specific extensions extend RISC-V cores with additional functionality [20, 27, 72].

Privilege Levels. RISC-V has three privilege levels: User (U) for unprivileged applications, Supervisor (S) for operating systems, and Machine (M) for full hardware control and low-level operations [78]. 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.

2.2 CPU Vulnerabilities

While software-based side-channel attacks have been known for decades [42], recent years revealed more critical CPU vulnerabilities [41, 44, 52, 53]. The first major class were transient execution attacks [11, 41, 44], such as Meltdown [44] and Spectre [41]. These exploit CPU optimizations like out-of-order and speculative execution to leak data across boundaries such as user/kernel space or even virtual machines [11, 41, 44, 75]. However, they remain limited to read primitives, as they follow the architectural specification but leave microarchitectural traces.

In contrast, architectural bugs are mismatches between CPU specification and implementation. For example, the Pentium F00F bug let unprivileged users lock systems via an invalid instruction [15]. More recently, several CPU bugs caused denial of service [1–4] or even direct data leakage [10, 52, 53, 83]. Unlike transient execution, architectural bugs often allow reliable exploitation primitives with severe impact [10, 53, 83].

2.3 CPU Fuzzing

Fuzzing tests hardware or software with random inputs to thoroughly check for unexpected behavior. While fuzzing cannot prove the complete absence of bugs, it is highly effective in quickly finding them [34, 39, 67]. Differential fuzzing [45] compares multiple implementations of the same specification, flagging divergences as potential bugs without requiring any golden model.

With the rise of severe CPU vulnerabilities, fuzzing has been applied both pre-silicon [16, 34, 39, 67] and post-silicon [47, 50, 80]. Many works have targeted transient execution attacks [13, 16, 28, 30, 33, 47, 50, 51, 73, 74], while fewer efforts focus on architectural vulnerabilities [34, 39, 67]. Most approaches are pre-silicon, but Bölcskei et al. [8] show that structural-coverage fuzzers can be slower than black-box fuzzers. This motivates further exploration of black-box post-silicon fuzzers, especially as few exist for architectural vulnerabilities and mostly on x86 [52, 53].

3 Methodology

This section outlines our overall methodology. Section 3.1 presents the main idea, Section 3.2 challenges, and Section 3.3 the fuzzing targets. Implementation details are provided in Section 4.

3.1 Idea

We exploit the property of the ISA that the *architectural* result of *every* supported instruction has to be the *same across different CPUs*. Consequently, this also applies to *instruction sequences*. Thus, we consider any architectural effect that differs between CPUs an *instruction anomaly* that needs investigation, excluding instructions only supported on one CPU. These instruction anomalies are likely due to a bug or, worse, a security vulnerability in the CPU.

Further, we restrict testing to user space, which acts as an automated filter for finding security-critical bugs in contrast to generic bugs. For example, denial-of-service vulnerabilities in M- or S-mode have limited impact as they are not part of typical threat models, while bugs triggerable from user space can be a significant threat. Moreover, the client can run sandboxed, e.g., in a container or an Android app, making it applicable to many devices.

The main advantage of this approach compared to previous CPU fuzzers searching for architectural bugs [34, 39, 67] is that we require neither source code, golden models, nor privileged execution. Thus, this approach can automatically find *user-triggerable security vulnerabilities* on closed-source CPUs, such as the T-Head C908 or SpacemiT X60. Moreover, running code on hardware CPUs is faster than emulating them, especially with increasingly complex CPUs.

As we cannot exhaustively test all instruction sequences, we select instructions randomly with a probability biased by the inverse frequency of it appearing in real-world code. In other words, our focus is more on instructions that are not used frequently. We show that this "weighted randomness" is fast and highly effective in finding new and re-discovering known vulnerabilities.

3.2 Challenges

While the idea of differential CPU testing is intuitive, we identify challenges in both design and implementation that arise from our two main properties: finding security vulnerabilities triggerable from user space, and analyzing CPUs without available source code. Conceptually, there are the following 3 main challenges:

$C1$: Sequence Generation. The first challenge is generating instruction sequences that are effective at finding "interesting" effects. In contrast to RTL fuzzers, we do not have feedback for coverage and thus have to rely on the number of bugs and time-to-bug as metrics for evaluating the quality of our instruction sequences. Moreover, we want to discover vulnerabilities stemming from undocumented instructions and can thus not fully rely on golden models as previous work [34, 39, 59, 67]. With a large search space for instruction sequences, we have to find a trade-off between coverage of the encoding space and the number of tested instructions. In Section 4.3.1, we describe how RISCover solves that challenge by using a bottom-up approach to gradually increase the search space using instruction types inferred from instruction encodings, combined with a weighted random selection of instructions based on instruction frequency of real-world code.

C2: 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. In contrast to RTL fuzzers that can modify, reset, and control the CPU source code, we do not have this amount of control over the CPU and its microarchitectural state. 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. Moreover, as an entire operating system is running on the CPU, we cannot easily modify all architectural state to our liking, e.g., create arbitrary memory mappings. Finally, memory reads from certain addresses, such as Linux vDSO [25] 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 differences are not introduced by the tested instruction sequence. In Section 4.2.1, we describe how we prevent the reporting of non-deterministic effects by minimizing the sources of non-determinism and automatically filtering the results of the remaining non-deterministic instructions.

C3: Test-framework Integrity. The test framework has to record the architectural effects of instruction sequences. Thus, from a high-level perspective, the architectural state, such as register and memory content, must be saved before and after executing the sequence. RTL fuzzers can directly retrieve this state from the emulator or add custom instructions to store this state [67], which is not possible for CPUs without source code. As RISCover runs purely in user space, it has to handle all corner cases that would change the saved architectural state or the internal state of the test framework, e.g., control-flow changes or stack manipulations. We describe the implementation details of this approach in Section 4.2.2.

3.3 Fuzzing Targets

We test all widespread commercially available 64-bit RISC-V cores that support booting a Linux distribution. At the time of writing, there are 3 manufacturers of silicon RISC-V CPUs—SiFive, SpacemiT, and T-Head Semiconductors. We test on 3 SiFive (U54, U74, and P550), 1 SpacemiT (X60) and 4 T-Head (C906, C908, C910, and C920) CPU models. Tested CPUs are mounted on single-board computers (U54, U74, C906, C908, C910), a laptop (X60), workstations (C920 and P550), or a cloud server (C910). All targets run Linux distributions—primarily Ubuntu or Debian.

All tested cores support at least the base ISA, the standard extensions, and compressed instructions, i.e., RV64GC. The C908 and X60 support the ratified RISC-V vector extension (v), while C906 and C910 use the `XTheadVec` extension supported by GCC [26]. Further, we test on different QEMU versions from version 6 (default on Ubuntu 22.04) to 9 (newest).

4 RISCover Framework

This section describes RISCover (RISC-V vulnerability discovery), our proof-of-concept implementation of the methodology described in Section 3. Specifically, Section 4.1 describes and motivates the design of RISCover. Section 4.2 and Section 4.3 discuss relevant design and implementation details of the client and server components of RISCover, respectively.

4.1 Design

Figure 1 shows a high-level overview of our proof-of-concept implementation. RISCover uses a centralized design: One server orchestrating testing, and multiple clients, i.e., RISC-V CPUs, connected to the server. A client receives an instruction sequence plus input over the network, runs it, and reports back the resulting state, e.g., register values and changed memory contents. The server is responsible for generating test cases, distributing them to clients, collecting results, and analyzing those results. We choose this design to reduce the task of the clients to a minimum as the tested devices are constrained in resources. The tested RISC-V CPUs cannot keep up with, e.g., powerful x86 CPUs which we use for the server. Further, in contrast to the clients, the server infrastructure can also be more easily scaled. An alternative approach that decreases network overhead at the cost of increasing CPU overhead is generating inputs on-device. We evaluate this alternative approach but find that on-server sequence generation is beneficial for our set of devices and constraints in Section 5.1.1. Furthermore, the tested devices offer limited storage (e.g., 8 GB on Board A), making it impractical to store all results locally. Additionally, over 99 % of the results can be immediately discarded once compared on the server as the respective inputs generate no architectural difference or are filtered by generic filters. We leave exploring alternatives like pre-computing result checksums for each input to future work. RISCover is implemented in 4 k lines of C and 6 k lines of Python.

4.2 Client

The client is a runner of server-provided test cases. To execute test cases, the client sets the registers as specified, runs the provided instruction sequence, and reports the results to the server. To track memory modifications of the test case without having to scan the entire memory, RISCover implements lazy memory mapping. The test case is first executed without additionally mapped pages, leading to segmentation faults on memory accesses. RISCover maps and keeps track of the required pages for the faulting accesses. The result of a test case is used once no more faults are observed or a threshold of mapped pages is exceeded. Consequently, only a small, known number of pages have to be checked for modifications.

4.2.1 *C2: Removing Non-determinism.*

Since we assume every difference in register values to be a bug, we must ensure this does not happen non-deterministically. There are instructions that are inherently not deterministic, e.g., the `rdcycle` instruction. Additionally, the operating system influences values of memory locations and performance counters [27]. As a first step to removing non-determinism introduced by the operating system, we use static compilation for our binaries and unmap non-essential user-accessible operating-system mappings, such as vDSO. To fully remove the remaining non-determinism, we exclude instructions that do not produce the same result twice on different cores of the same CPU. This is opposed to SiliFuzz [59] that reports test cases that have a different outcome on different cores of the same CPU to find CPU defects (instead of vulnerabilities).

4.2.2 *C3: State Protection.*

As RISCover executes arbitrary code sequences in user space, these sequences can modify RISCover's internal state. Thus, RISCover has to protect its internal state to

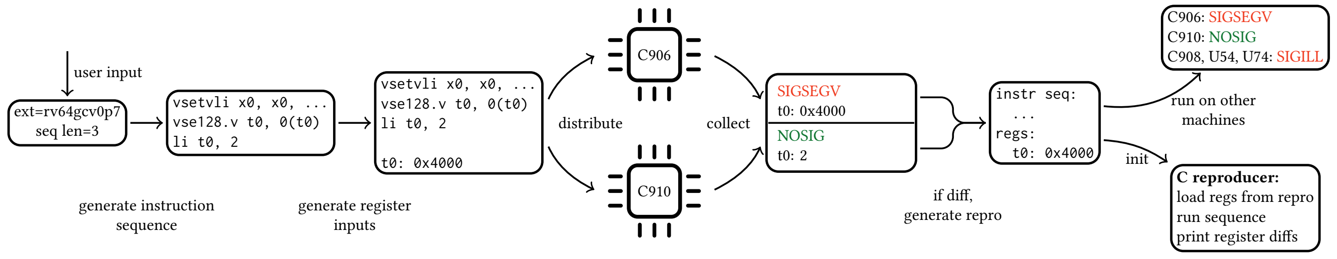

Figure 1: Overview of RISCover. The user selects ISA extensions and sequence length. The server generates instruction sequences and register inputs, distributes them to the clients, and compares results. Differences are logged as a reproducer file.

ensure a correct reporting of instruction effects. In contrast to RTL fuzzers, the CPU cannot be reset after executing a test case. Thus, RISCover requires a robust sandbox design to ensure that test cases do not influence the remaining system while still being able to execute arbitrary instruction sequences. Specifically, RISCover 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.

Registers. If a test case exits normally, i.e., without an exception, we use an instruction sequence that stores all registers to memory. Otherwise, e.g., after an illegal-instruction exception, the operating system automatically saves the registers before jumping into the signal handler. We copy this state and return to the execution loop with a `longjmp` that restores the register state.

Memory. We leverage the large virtual address space to "hide" the internal data of RISCover in a region that is difficult to overwrite accidentally. The kernels we run on our CPUs all use the Sv39 paging mode [78], i.e., 512 GB of virtual address space is available for "hiding". Consequently, we move the data section to a "safe" region of memory that we experimentally determined. It is neither close to the test cases nor at the beginning or end of virtual memory to prevent accidental accesses by small positive or negative values that are used as addresses. Further, we switch to a new stack that we hide in this region as well. While this approach is heuristic, it works well in practice.

Control Flow. As we cannot reset or bring the CPU into a clean state from user space, a test case must not jump out of RISCover's logic or lock up the client. We pad test cases with `ebreak` instructions to ensure any relative jump triggers an exception and returns control back to RISCover. We rely on the size of the virtual address space to "hide" the sandbox. We move the sandbox to its own memory range and use absolute jumps to get in and out of the runner sandbox. We handle infinite loops by priming an alarm before jumping into the sandbox, interrupting possibly infinite loops.

4.3 Server

The server generates instruction sequences and input registers and sends these to the clients, which run the test cases and report the resulting states back. The server compares these states and logs differences as reproducer files that can be used for further analysis.

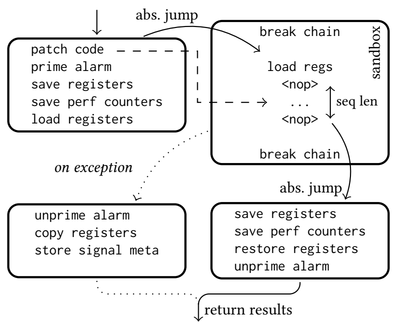

Figure 2: The runner sandbox includes `nop` placeholders surrounded by `ebreak` instructions. The runner patches the placeholder instructions, primes the alarm, saves registers, loads the supplied fuzzing registers, and jumps into the sandbox. The executed instruction sequence either returns via the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

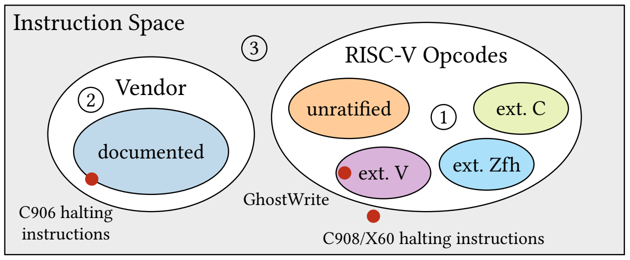

Figure 3: Overview of the RISC-V instruction space. RISC-V Opcodes ① covers most parts of the RISC-V ISA specification. The specification reserves parts of the address space for custom vendor extensions ②. Other parts are either reserved for future use or unclaimed ③. The dots describe where our discovered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

4.3.1 C1: Sequence Generation. We use a bottom-up approach based on the instruction encoding and their real-world frequency 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

Table 1: 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 and T-Head extensions cover only small parts of the instruction space. Overall, 85.51 % of the instructions are specified, and 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 (`XTheadVec`)	0.81 %
T-Head vendor extension	0.39 %
Overall known	85.51 %

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 1). 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.

Instruction Classification. We use the official RISC-V Opcodes repository [23] for building our filters since it encodes all standard RISC-V instructions and several unratified instructions in a machine-parsable format. Further, it clusters the instructions into their respective extension.

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, we exclude them to avoid introducing false positives in the differences. Note, however, that we permit the `frcsr` and `fscsr` instructions, which read and modify the Floating-Point Control and Status Register (`fcsr`) [77], as this CSR is part of the CPU's well-defined architectural state. We leave the coverage of other CSRs to future work.

Weighted Instruction Selection. The server generates instruction sequences using a bottom-up approach guided by real-world instruction frequencies, derived from extensive analysis of Debian packages (1.36 billion disassembled instructions from 84 164 software packages). Each instruction is selected inversely proportional to its real-world frequency, prioritizing rarely used or undocumented instructions, which have a higher potential for undiscovered vulnerabilities. Immediate values are selected from a set of predefined "interesting" corner-case values (e.g., zero, negative numbers, maximal integer values). They are augmented by randomly chosen values in 1 out of every 5 instructions, ensuring broad encoding coverage. Further, we initialize source and target registers of instructions with a limited set of 5 possible registers starting from `x0` to cause dependencies between instructions in a sequence. Again, we provide a random register in 1 of 6 cases. For the rest of the fields, we provide the required number of random bits. The sequence length is fixed at runtime—our evaluations show that lengths of 3-5 instructions provide an optimal trade-off between vulnerability discovery effectiveness and computational efficiency.

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 undocumented instruction, RISCover does not have to fill any bitfields, as a complete instruction encoding with all bits set is already chosen. Thus, RISCover covers the entire instruction space with this approach.

While RISCover focuses on rarely used or undocumented instructions to maximize vulnerability discovery potential, it also uncovers several architectural bugs within commonly used instructions (cf. Section 5.2). However, these cases typically represent functional discrepancies rather than directly exploitable security vulnerabilities, reaffirming our strategy to emphasize testing of the less frequently validated and more complex instruction spaces.

4.3.2 Input Distribution. The server distributes generated fuzzing inputs and collects results (cf. Figure 1). Since we want to achieve high throughput fuzzing on the clients, we highly optimize the amount of data transferred. We only send back registers and memory contents that *changed* during the execution of the instruction sequence, restrict RISCover to a fixed set of possible register contents, 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 that the network is efficiently used.

4.3.3 Logging Differences. The server compares the collected architectural states and logs a reproducer file for the fuzzing input if it discovers a difference. This reproducer file can then be compiled and executed as a standalone binary on any RISC-V CPU for further analysis. While the reproducer is not necessarily minimal, it is typically small enough for an analysis. Still, if necessary, program reduction techniques, such as those discussed by Solt et al. [67], can be used to reduce the reproducer further.

5 Evaluation

We evaluate the general performance of testing instruction sequences (Section 5.1), summarize the findings of RISCover (Section 5.2), including the most severe finding, GhostWrite, and evaluate how long it takes RISCover to discover our findings (Section 5.3). We compare RISCover to the state-of-the-art RTL fuzzer Cascade [67] (Section 5.4), and evaluate the efficacy of our sequence generation (Section 5.5).

5.1 Fuzzing Performance

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

5.1.1 Core Utilization & Framework Overhead. To validate the efficiency of the design of RISCover, we measure CPU utilization and client-side overheads such as sequence generation and network communication. All benchmarks are performed on the fastest tested CPU, the C908 (Board G), using instruction sequences of length 3.

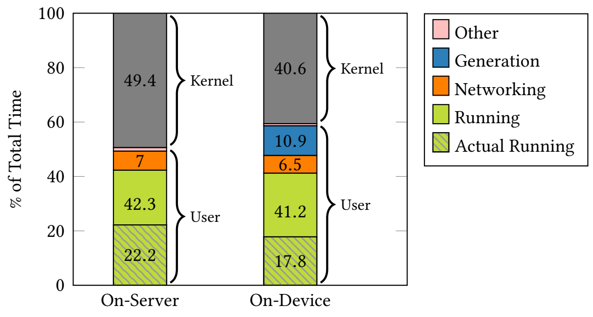

Figure 4: Stacked bar plot comparing time spent in different client sections for on-server and bare-bones on-device sequence generation. On-device generation reduces kernel overhead, but this benefit is canceled out by the extra cost of input generation, yielding no net performance improvement. Both approaches show comparable efficiency in the actual running of inputs, indicated by the hatched pattern.

We achieve 100 % CPU utilization on the C908, as confirmed via `htop`. This demonstrates that RISCover can fully saturate even fast RISC-V CPUs like the C908.

To quantify client overhead, we instrument the client to measure time spent in key phases. Further, we measure time spent in kernel and user space and compute the fraction of time spent on the key phases as the product of the fraction of time in user space and fraction of time in the key phase. Figure 4 shows the results in a stacked bar plot. With the default on-server sequence generation, 42.3 % of the time is spent running transmitted inputs, while 22.2 % in that time is spent on actually running the sequence. Networking, i.e., receiving inputs and sending results, accounts for 7 % of time. Overall, the client spends 50.6 % of total time in user space. Note that this includes signal handling which cannot be cleanly separated from networking. These results indicate that RISCover has a low network overhead (< 20 %).

We further benchmark the overhead of a bare-bones implementation of on-device sequence generation. Here, the client generates the inputs on-device using a shared RNG implementation and seed. Figure 4 again shows the results. In this setup, 10.9 % of time is spent on input generation. 41.2 % is spent running the on-device-generated inputs. Networking overhead drops to 6.5 %. The overall user-space time increases to 59.4 %. This again shows that the signal handling overhead is rather high, as even with reduced networking we still only spent around 60 % of time in user space.

These results show that on-server sequence generation (42.3 % running inputs) is on par with on-device sequence generation (41.2 % running inputs) and does not introduce unnecessary overhead. This further indicates that on-server sequence generation is beneficial for lower-end devices or when handling more complex sequence generation.

Takeaway On-server and on-device sequence generation perform comparable. On-server generation can be beneficial when sequences are complex or the DUT is slow.

Table 2: Discovered novel vulnerabilities and affected CPUs, showing the vulnerability type, affected models, related ISA extensions, and time-to-bug.

Vulnerability	CPU Models	Extension	Time-to-Bug
Privilege Escalation	C910, C920	`XTheadVec`	<1 s
Denial-of-Service	C906	`XTheadMemIdx`	<2 min
Denial-of-Service	C908	Vector	<15 min
Denial-of-Service	X60	Vector	<50 min
Wrong Misaligned Zero-Stores	BeagleV Fire (U54)	Base ISA	<5 s
User-space Hangs	C906, C908, C910, X60	Base ISA	<5 s
Undocumented Instructions	P550	Base ISA	<1 s
Decoder Faults	C906, C910	Base ISA	<1 s
ISA Incompatibilities	C908, C910	Vector, Base ISA	<2 min
QEMU Segfaults	QEMU 8.2.2, 9.0.0	Vector, Cache Ops	<30 s

5.1.2 General Throughput. To assess the general throughput of RISCover, we benchmark a fuzzing run with only the base ISA with 2 clients (C906 and C908). We test 18 194 (C906) to 59 205 (C908) instructions per second on average.

Comparing the results to the fastest state-of-the-art RISC-V RTL fuzzer Cascade [67] shows that fuzzing on hardware cores is orders-of-magnitude faster. Cascade achieves 110 and 26 instructions per second on the openC906 and openC910, respectively, fuzzing on two Intel Xeon E5-2697 v4 (Figure 5 in Appendix E). Note though, that this is already with a sequence length of 10 000, while we use a sequence length of only 3 instructions.

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

This further highlights a desirable property of our approach: While RTL fuzzing negatively scales with increased complexity of the fuzzed cores, our approach scales with increased complexity since the hardware becomes faster. Good examples here are out-of-order and speculative execution present in the openC910. Both increase the performance of the CPU but complicate its simulation.

Takeaway RTL-fuzzing scales inversely with CPU complexity, while hardware core testing scales with it.

5.1.3 Sequence-Length Scaling. With our approach, performance increases up to a sequence length of 5 and then gradually decreases with further increasing sequence length. The results are 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. A sequence length of 3 is a good tradeoff since adding more instructions only slightly improves performance while causing more congestion on the network, potentially hindering other clients from receiving data.

5.2 Findings

In this section, we summarize findings around GhostWrite and the CPU-halting instructions, which we discuss in more detail in Section 6 and Section 7, respectively. Further, we discuss illegal misaligned zero-store behavior on the BeagleV Fire (SiFive U54, Board A), undocumented instructions on the SiFive P550 and categorize other findings into address-handling bugs, decoder bugs, ISA incompatibilities, and fault-reporting issues. Table 2 provides an overview, including the fuzzing time required for each bug.

Beyond bug discovery, we evaluate the exploitability of bugs found by RISCover. Notably, RISCover automatically finds architectural differences triggerable by an unprivileged user from user space. The analysis of whether these bugs are security vulnerabilities still requires manual analysis. However, our reproducer files typically contain fewer than 5 instructions, allowing rapid manual analysis to classify discovered anomalies into functional bugs or exploitable vulnerabilities. Our analysis process involves systematically varying register contents and immediate values to observe architectural effects, confirm consistency across reboots and environments, and investigate potential security implications such as arbitrary memory accesses or CPU denial-of-service conditions. The average time required per reproducer to confirm vulnerability status ranges from seconds to a few minutes, significantly accelerated by the minimal and focused nature of our generated reproducers.

Bug Sources. The location of buggy instructions in the encoding space hints at the bug source. Figure 3 visualizes where RISCover finds the most severe bugs. GhostWrite is in the vector extension, hinting at a bug in the vector engine. The C908 and X60 halting instructions are illegally encoded vector instructions close to but outside the vector extension. The C906 halting instructions are on the edge of the vendor extension since they use an edge case in the instruction encoding. Other bugs, including illegal misaligned zero-stores lie in the base ISA as summarized in Table 2.

GhostWrite. GhostWrite produces differences when fuzzing the `XTheadVec` extension between C906 and C910. While the illegally-encoded vector-store instructions generate a segmentation fault on illegal memory addresses on the C906, the C910 generates no exception. We provide an example of such a difference logged into a reproducer file in Appendix C. 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 6).

Undocumented & CPU-Halting Instructions. We find CPU-halting instruction sequences on the C906, C908, and X60 which directly crash the fuzzing client. No further analysis of such reproducers is needed as such a denial of service is always considered a security problem [1–4]. On the SiFive P550, we find 41 943 072 undocumented instruction encodings. These are split over two opcodes with opcode `0xb` having 20 971 535 and opcode `0x2b` having 20 971 537 encodings. We suspect these to be decoder bugs as they encode additional base instructions like shifts or immediate loads, which also have valid documented encodings.

Wrong Misaligned Zero-Stores. RISCover finds that misaligned zero-stores—such that use the zero register as source—show illegal behavior on the BeagleV Fire (SiFive U54, Board A). These non-deterministically write −1 of different sizes, e.g., 4- or 8-byte for a 8-byte wide store, instead of 0. While the ISA specification leaves open if misaligned stores should be trapped or if and how they are implemented, e.g., via trap or in hardware, it does not permit such illegal behavior. By comparing the speed of misaligned and aligned stores, we verify that misaligned stores indeed take longer, as they need special handling as the manual suggests [61]. We verify that this is a bug in the M-mode firmware of the BeagleV Fire that emulates these misaligned stores. This issue is fixed in the latest version of the firmware.

Address-handling. RISCover finds different bugs around virtual address handling. On the C910, reading from physically-backed virtual address '0' locks the CPU. On the T-Head XuanTie C906, C908, and C910, and on the SpacemiT X60, a load to a non-canonical address is stuck until an interrupt arrives if the canonical part of the address is a valid address.

Decoder Bugs. On the C906 and C910, RISCover 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" [77]. Conversely, RISCover 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 ≠ 0 [22]. Finally, for QEMU 9.0.0 and QEMU 8.2.2, RISCover discovers that cache-block management instructions such as `cbo.inval` crash QEMU with a segmentation fault. This is fixed with QEMU 9.1.2. For QEMU 7.2.0 (Emulator B), RISCover 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.

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. Further, the C910 and the C908 support only a subset of the vector extension. This manifests itself in some of the instructions doing nothing, others doing unexpected things (cf. Section 6), and some not being implemented at all. Interestingly, the subset of instructions also differs between the two CPUs.

Fault-reporting Issues. On all tested CPUs, RISCover 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 raises a segmentation fault instead of a bus error for a valid misaligned address.

5.3 Time to Bug

In line with other papers on fuzzing [57, 67], we provide the fuzzing time to find the bugs. We use all extensions, i.e., all ratified and unratified extensions, during fuzzing. We find GhostWrite within the first second of fuzzing, as no particular encoding in the broken instruction is needed to make the bug visible. Thus, the fuzzer only needs to select one of the 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 2 minutes. The longer time to bug can be explained by lower fuzzing throughput on the C906 and by more involved conditions that the broken instruction and the sequence need to satisfy, e.g., using the same registers in the encoding of the instruction and the following instructions. On the C908 and X60, RISCover finds the undocumented CPU-halting instructions in under 15 min and 50 min of iterating over the undocumented space, respectively.

For the other documented instruction findings, the time to bug is typically below 1 s. However, for some findings, fuzzing times of up to 2 min are required to reveal them (cf. Table 2).

5.4 Comparison to RTL Fuzzer Cascade

We compare RISCover to the fastest state-of-the-art RISC-V RTL CPU fuzzer Cascade [67], evaluating the bug-finding efficacy of both approaches. We use the openC906 and openC910, the open-source designs T-Head claims are used in the C906 and C910 CPUs, respectively. These are the only cores that are available as off-the-shelf hardware and RTL source. We compare our sequence generation with the complex sequence generation of Cascade, showing that our sequence generation can also find 22 out of 23 user-space-triggerable bugs Cascade found. Additionally, we show that even after 14 days of fuzzing, Cascade does not find the bugs RISCover discovers on the C906 and C910, but only CSR value mismatches without security implications.

5.4.1 Rediscovering Cascade Bugs. We analyze all bugs Cascade found on RTL designs that are in scope for RISCover, i.e., triggerable from user space, and demonstrate that our sequence generation generates a test case for 22 of the 23 bugs. The verification functions are created based on the bug descriptions of the Cascade paper, the corresponding code of the Cascade time-to-bug analysis, and the publicly available information on the bugs from bug trackers. Due to Cascade executing its code sequences in machine mode, RISCover's user-space approach can, by design, not find bugs that require access to privileged CSRs or a disabled FPU. Hence, we exclude them in our analysis, as they are also not a security problem when considering an unprivileged attacker. Table 4 (Appendix B) contains the results of all 23 in-scope bugs that Cascade found. We see that while our weighted sequence generation is not always faster than our unweighted sequence generation, e.g., for bug C1, it is on par for most bugs while bringing a significant speedup for bugs that are hard to find with an unweighted sequence generation, e.g., for bug V4. The only bug we do not rediscover is bug V12, as our approach does not identify transient execution vulnerabilities. For more details on why we choose this design decision, we refer the reader to Section 10. Assuming a hardware implementation of the cores, RISCover would discover all 22 bugs in under 10 s, with 18 bugs in the first 500 ms.

5.4.2 Fuzzing OpenC906 & OpenC910. We integrate the openC906 and openC910 into Cascade to evaluate the efficacy of Cascade on these complex cores. We adapt existing Cascade and T-Head test benches to fit the softcores but additionally modify the T-Head provided RTL code and the ISA simulator Spike [35] to enable fuzzing. On the RTL level, changes include increasing the available address space and modifying the bootup location of the softcore.

Discovered Bugs. We run Cascade with up to 100 blocks (1000 to 2000 instructions) on the openC910 and openC906 for 14 days on 64 cores of two Intel Xeon E5-2697 v4. In total, we discover 3 identifiable mismatches regarding CSRs and several unknown mismatches. We find that `MINSTRET` and `MEPC` can be off by 1 and 4 respectively for openC906 and openC910, and that `MCAUSE` can be off by 4 in the openC906. Cascade does not find GhostWrite and the C906 halting instruction sequence for multiple reasons. First, as we show in Section 6.1.3, GhostWrite is not present in the open-source code of the C910. Second, Cascade does not set up virtual memory. Therefore, there is no way to detect a broken or skipped translation. Third, Cascade does not embed illegal encodings such as GhostWrite or the custom C906 halting instructions nor vector instructions into the sequences. RISCover, on the other hand, does not find these CSR-related mismatches as we exclude CSRs during fuzzing since they can produce arbitrary differences and are partially privileged.

Fuzzing Throughput. To measure the fuzzing performance on the openC910 and openC906, we run the same fuzzing execution throughput experiment as Cascade for 1-core. We reproduce the results of Cascade nearly perfectly, i.e., find the sweet spot of program length to be at around 10 000 instructions. The openC906 and openC910 are even slower to simulate than the slowest core (BOOM) Cascade evaluated. They only achieve 110 and 26 fuzzing instructions per second compared to 556 for the BOOM core. This is expected since these softcores come with a much larger non-minimal SoC and are, therefore, bigger and more complex. The full results for all cores are in Appendix E.

5.5 Sequence Generation Evaluation

In addition to showing that our sequence generation is effective at finding CPU bugs (cf. Section 5.4), we evaluate the effect of the weighted random instruction selection. We use RISCover once with purely randomly chosen instructions and once with the weighted random approach. In both cases, we generate instruction sequences of length 3. Unsurprisingly, we find the same bugs with either configuration. However, we discover the C906 hang in 26 s without weights, while it only takes 19 s with weights.

To show that the weights are a generic improvement, we analyze their impact on the time to bug for the 22 Cascade bugs RISCover finds. For 18 bugs, the weighted random selection improves the time to bug, and only for 4 bugs it is slower (cf. Table 4 in Appendix B).

6 GhostWrite: Writing Physical Memory

In this section, we analyze GhostWrite, the arbitrary physical write primitive RISCover finds on the C910 and C920. In Section 6.1, 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. Section 6.2 demonstrates how to use GhostWrite for reading arbitrary physical memory and executing arbitrary code with kernel- and machine-mode privileges.

6.1 Analysis

Listing 1 shows the assembly code of GhostWrite with the relevant instruction with machine code `0x10028027`. 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.

```
; t0 = physical address, a0 = byte to be written
vsetvli zero, zero, e8, m1
vmv.v.x v0, a0
; encoded: 0x10028027
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.

The *source registers* encoded in the instruction work as intended, however 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. 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.

6.1.1 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. 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.

6.1.2 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 a LicheePi4A [64]. 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.

6.1.3 Simulation. In the openC910 [70], 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" [70].

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

6.2 Exploiting GhostWrite

In this section, we show how GhostWrite can be exploited for privileged code execution and for arbitrary memory leakage.

Threat Model. We assume unprivileged native code execution on a target running up-to-date Linux on a C910 or C920 CPU.

Privileged Code Execution. To gain code execution in the kernel, the attacker uses GhostWrite to overwrite the code of a system call handler that can be executed by triggering the corresponding syscall. 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.

```
th.lbib t0, (t0), 0, 0
frcsr t0
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.

We have a 100 % success rate on 3 different C910 boards (H, I, and J in Table 5). The attacks take less than 1 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.

Reading Memory. By rewriting page-table entries, we can convert GhostWrite to a primitive for reading arbitrary memory. We fill the entire available physical memory with page tables by mapping the same file numerous times until the physical memory is exhausted. 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. 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. 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 have complete control over a PTE and its corresponding virtual address [58]. Thus, we can rewrite the PFN to any physical address to read the content of the address.

We run the attack successfully on all 3 boards with 3 different DRAM configurations, 4 GB, 8 GB, and 16 GB. The attack also works from within a Docker container. 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 s.

7 CPU-Halting Instruction Sequences

In this section, we analyze the instruction sequences for halting the C906, C908, and X60. 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.

7.1 Analysis

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).

We further discover 13 other instructions from the `XTheadMemIdx` extension that are vulnerable in the same way (cf. Listing 4 in Appendix A). The variants with 3 source registers are not vulnerable.

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

C908. The instructions found by RISCover 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 zero.

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

X60. Similar to the C908, the discovered halting instructions on the X60 seem to be close to vector store/load instructions. We find that different encodings of the invalid instructions raise a segmentation fault in user space, further strengthening this hypothesis.

7.2 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 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. While this also demonstrates that the bug can be triggered in all privilege modes, it is only a security problem for the user mode.

7.3 Case Study

To assess the impact of the halting vulnerabilities, we evaluate in which contexts they can be executed to halt the CPU. The straightforward scenario is an *unrestricted native environment* as has been used by RISCover. Executing the instruction sequence as an unprivileged user immediately results in the CPU being halted. We verify the C906 behavior on two different boards using Debian 11 and Debian 12. Additionally, we verify the C908 behavior with Debian 13. 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 hypervisor supports the affected CPUs.

8 Mitigations

In this section, we discuss mitigations for GhostWrite (Section 6) and the CPU-halting sequences (Section 7).

Table 3: Analysis of whether prior CPU fuzzing work could theoretically find the bugs discovered by RISCover. RTL-fuzzing approaches can only detect the C906 hang, as it is the only CPU where the bug is present in the available softcore. T-Head instructions still need to be added to the golden model and sequence generation for Cascade. SiliFuzz cannot find any of the bugs, as none appear in disassemblers it relies on.

	Hardware+ user-mode	GhostWrite	C906 hang	C908 hang	K1 hang
DifuzzRTL [34] TheHuzz [39]	✗	✗[B]	~[D]	✗[C]	✗[C]
Cascade [67]	✗	✗[B]	~[DE]	✗[C]	✗[C]
SiliFuzz [59]	✓	✗[A]	✗[A]	✗[A]	✗[A]
RISCover	✓	✓	✓	✓	✓

[A]Not in disassembler [B]Bug not in RTL [C]RTL not available [D]If T-Head instructions added to Spike [E]If T-Head instructions added to sequence generation

GhostWrite and C908/X60 Halting Instructions. Disabling the vector extension is a viable mitigation for GhostWrite and the halting instructions on the C908 and X60. The CPU throws an illegal instruction exception when executing the affected instructions [78], making the gadgets 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` [7]. 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` on the C910 (Board I). On the C908 (Board G), the performance of `memcpy` and `memset` decreases by up to 77 % and 2 %, respectively. On the X60 (Board L), we observe a decrease of up to 65 % and 11 %. 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. Since no special condition is required 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" [72]. We verify that we cannot clear this bit from a kernel module.

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

9 Related Work

We analyze related work, with an overview in Table 3 showing whether it could theoretically find the bugs discovered by RISCover. Most bugs remain undetectable due to unavailable RTL or different goals and methodologies, as we detail below.

Undocumented Instructions. Armshaker [69] and Dofferhoff et al. [18] search for undocumented ARM and RISC instructions using disassemblers as ground truth, uncovering emulator bugs and ISA inconsistencies. DifuzzRTL [34] and Morfuzz [82] also fuzz undocumented RISC-V instructions but require machine mode and a perfect simulator. Sandsifter [19] shows the feasibility of exploring x86's larger instruction space via a decoding side channel. Unlike these, RISCover requires no ground truth, exhaustively tests all instructions, and operates purely from user space, exposing security-relevant bugs.

Differential CPU Fuzzing. Ormandy [52, 53] used Oracle Serialization, i.e., inserting memory fences to detect optimization-dependent differences. Such methods miss bugs like GhostWrite, which behave identically with fences, and cannot reveal cross-vendor or cross-generation issues. In contrast, RISCover compares CPUs from different vendors. SiliFuzz [59] targets electrical defects x86 cores, yielding results orthogonal to ours. Qin et al. [54] and Jiang et al. [36] compare CPUs with emulators and disassemblers for stealthy malware, exploiting software rather than CPU bugs.

Model Fuzzing. TheHuzz [39] and DifuzzRTL [34] fuzz RTL using emulation to guide search, while Cascade [67] generates more complex sequences for higher throughput. RTL fuzzers are valuable in development but require complete RTL, typically unavailable for commercial cores (the C906 being the only exception). RISCover instead targets opaque hardware CPUs, offering faster throughput and testing of deployed systems. Its speed enables full exploration of undocumented instructions via a bottom-up approach rather than mutation. While guidance [38] or fuzzing by proxy [59] could extend RISCover, its current form already produces many results.

Fuzzers for Microarchitectural Vulnerabilities. Work on transient execution (e.g., Revizor [50], Transynther [47]) targets speculation, whereas RISCover uncovers architectural bugs reproducible without speculation. Other efforts study instruction-level timing differences [17, 29, 32, 76, 80] or architectural side channels [73], which yield weaker attacker capabilities. Unlike RTL fuzzers [34, 67], RISCover needs no RTL or golden model, making it complementary to existing microarchitectural fuzzing and formal verification.

Time-Multiplexed Testing. Time-multiplexed testing, close to RISCover and RTL fuzzers, duplicates execution contexts to catch faults. EDDI-V [31] and RMT-V [31] build on EDDI tests [49] to detect silicon faults via temporal redundancy within one pipeline. These approaches excel at random or manufacturing faults on single CPUs. RISCover instead applies spatial redundancy across different vendors' CPUs, detecting systemic design and implementation bugs.

10 Discussion

We discuss applicability to other ISAs, limitations of our current implementation, and implications for future CPUs.

Other Architectures. RISCover is architecture-agnostic and can be extended to other ISAs such as ARM or x86, given engineering effort to handle ISA-specific details. However, it requires multiple hardware implementations or accurate emulators for meaningful differential comparisons. For architectures or extensions with a single vendor or identical multi-core silicon, the efficacy of differential fuzzing diminishes.

Vulnerability Types. Our methodology targets architectural bugs triggerable from user space; microarchitectural bugs relying on speculation or out-of-order execution [47, 50] fall outside our current scope. These require side channels to infer hidden state, which are noisy and ill-suited for exhaustive testing. In principle, RISCover could be extended to leverage architectural interfaces on other ISAs, but given extensive existing work [13, 16, 28, 30, 33, 47, 50, 51, 74] and the orthogonal nature of the problem, we exclude this from our prototype.

Coverage. Coverage is a common metric to compare fuzzers, but RISCover cannot obtain coverage from silicon RISC-V CPUs. While RTL sources for the C906 and C910 exist, using them is impractical. First, simulation is prohibitively slow: our time-to-bug analysis shows sufficient coverage within minutes on hardware, but even an optimistic 100 000× slowdown in simulation would require years. FPGA-based coverage, as shown by Laeufer et al. [43], would also demand significant engineering and powerful hardware. Second, RTL coverage would be misleading since public RTL deviates from silicon. For example, the C910 RTL lacks the vector extension incorporating GhostWrite (cf. Section 6.1.3). Third, RISCover needs a full Linux environment with advanced OS features, and even a single boot in simulation would take days.

We therefore evaluate coverage as bug coverage. RISCover finds all but one bug discovered by Cascade (cf. Section 5.4.1), which achieves state-of-the-art RTL coverage. This suggests comparable effectiveness. Moreover, RISCover exercises diverse parts of the microarchitecture, uncovering bugs in vector units, floating point, decoder, memory system, and firmware (cf. Section 5.2).

CPU Testing Suites. Vendor validation suites, such as Arm's [5], increase robustness but often miss undocumented or improperly implemented instructions that RISCover reveals. Similarly, the tool RISCV-DV [14] is widely used for RISC-V verification but only generates valid ratified instructions, unlike our approach.

Benefits of Post-silicon Fuzzing. Bugs discovered post-silicon cannot be patched in existing hardware, but since designs are reused, fixes propagate to future generations. Thus, post-silicon fuzzing benefits both security researchers and vendors, who gain from high fuzzing throughput as soon as engineering samples exist.

Consequences for Future CPUs. Insights from RISCover allow vendors to integrate preemptive mitigations into new designs, preventing recurring flaws. For instance, silicon could support selective disabling of problematic instructions.

11 Conclusion

In this paper, we introduced RISCover, a differential RISC-V CPU fuzzing framework for automatically discovering architectural security vulnerabilities in hardware CPUs. RISCover compares the architectural results of instruction sequences without relying on CPU source code or any emulator. Within minutes, RISCover discovered 4 severe security vulnerabilities that can be exploited from user space and several other bugs on 8 different CPUs. On the T-Head XuanTie C910 and C920, RISCover discovered GhostWrite, an instruction sequence that allows unprivileged attackers to write arbitrary values directly to physical memory, entirely circumventing virtual memory and its protection. Further, we investigated 3 "halt-and-catch-fire instructions" on 3 different CPUs, the T-Head XuanTie C906 and C908, and the SpacemiT X60. 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 the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS) and a Google Research Scholar Award. The views expressed are those of the authors and do not necessarily reflect those of the sponsors.

References

[1] 2015. CVE-2015-5307. https://nvd.nist.gov/vuln/detail/cve-2015-5307 CVE-2015-5307.

[2] 2015. CVE-2015-8104. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8104 CVE-2015-8104.

[3] 2018. CVE-2018-12207. https://nvd.nist.gov/vuln/detail/CVE-2018-12207 CVE-2018-12207.

[4] 2021. CVE-2021-26339. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-26339 CVE-2021-26339.

[5] Arm. 2016. System Validation at ARM: Enabling Our Partners to Build Better Systems. https://developer.arm.com/-/media/ArmDeveloperCommunity/PDF/SystemIP/System_Validation_at_ARM_Enabling_our_partners_to_build_better_systems.pdf?revision=a82b4e93-4118-4a3a-ba5f-70e38f6b4616&hash=88D9B7CF58BE13B124E43EE538D21F4D

[6] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Berkley* (2016).

[7] Olaf Bernstein. 2023. rvv-bench: RISC-V Vector benchmark. https://github.com/camel-cdr/rvv-bench

[8] Matej Bölcskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2025. Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection. In *USENIX Security*.

[9] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. 2023. CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode. In *WOOT*.

[10] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*. Extended classification tree and PoCs at https://transient.fail/..

[12] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report.

[13] Anirban Chakraborty, Nimish Mishra, and Debdeep Mukhopadhyay. 2024. Shesha: Multi-head Microarchitectural Leakage Discovery in new-generation Intel Processors. *arXiv preprint* (2024).

[14] CHIPS Alliance. 2023. riscv-dv. https://github.com/chipsalliance/riscv-dv

[15] Robert R. Collins. 1998. The Pentium F00F Bug. http://www.rcollins.org/ddj/May98/F00FBug.html

[16] Alvise de Faveri Tron, Raphael Isemann, Hany Ragab, Cristiano Giuffrida, Klaus von Gleissenthall, and Herbert Bos. 2025. Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks. In *USENIX Security*.

[17] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W Fletcher. 2024. Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *S&P*.

[18] Rens Dofferhoff, Michael Göebel, Kristian Rietveld, and Erik Van Der Kouwe. 2020. IScanU: A portable scanner for undocumented instructions on risc processors. In *International Conference on Dependable Systems and Networks*.

[19] Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. *Black Hat US* (2018).

[20] RISC-V Foundation. 2019. *RISC-V "V" Vector Extension 0.7.1.* https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1

[21] RISC-V Foundation. 2021. *RISC-V "V" Vector Extension 1.0.* https://wiki.riscv.org/display/HOME/Ratified+Extensions

[22] RISC-V Foundation. 2021. *RISC-V "Zfh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point, Version 1.0.* https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions

[23] RISC-V Foundation. 2022. riscv-opcodes. https://github.com/riscv/riscv-opcodes

[24] RISC-V Foundation. 2023. RISC-V Architecture Test SIG. https://github.com/riscv-non-isa/riscv-arch-test

[25] Mike Frysinger. 2024. vdso(7) — Linux manual page.

[26] GCC Team. 2024. GCC 14 Release Series - Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-14/changes.html Retrieved 2024-09-20.

[27] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. 2023. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *S&P*.

[28] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. 2021. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *ISCA*.

[29] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*.

[30] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. 2023. Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions. In *USENIX Security*.

[31] Ted Hong, Yanjing Li, Sung-Boem Park, Diana Mui, David Lin, Ziyad Abdel Kaleq, Nagib Hakim, Helia Naeimi, Donald S Gardner, and Subhasish Mitra. 2010. QED: Quick error detection tests for effective post-silicon validation. In *IEEE International Test Conference*.

[32] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *MICRO*.

[33] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security*.

[34] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *S&P*.

[35] RISC-V International. 2024. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim

[36] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. EXAMINER: Automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In *ASPLOS*.

[37] joopdehoop. 2023. *Firmware update for 'Chinese' F133 head unit.* https://www.reddit.com/r/CarAV/comments/18ivjsg/firmware_update_for_chinese_f133_head_unit/

[38] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO*.

[39] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *USENIX Security Symposium*.

[40] Kelly Le. 2024. Alibaba's Damo Academy plans to launch latest version of its XuanTie RISC-V processor this year. https://www.scmp.com/tech/big-tech/article/3255830/alibabas-damo-academy-plans-launch-latest-version-its-xuantie-risc-v-processor-year Retrieved 2024-09-10.

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[42] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.

[43] Kevin Laeufer, Vighnesh Iyer, David Biancolin, Jonathan Bachrach, Borivoje Nikolić, and Koushik Sen. 2023. Simulator independent coverage for RTL hardware languages. In *ASPLOS*.

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.

[45] William M McKeeman. 1998. Differential testing for software. (1998).

[46] Milk-V. 2023. *Milk-V Pioneer.* https://milkv.io/pioneer

[47] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*.

[48] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*.

[49] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*.

[50] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing black-box CPUs against speculation contracts. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[51] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *IEEE S&P*.

[52] Tavis Ormandy. 2023. Reptar. https://lock.cmpxchg8b.com/reptar.html

[53] Tavis Ormandy. 2023. Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html

[54] Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. 2021. iDEV: Exploring and exploiting semantic deviations in ARM instruction processing. In *ISSTA*.

[55] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *AsianHOST*.

[56] Scaleway. 2024. *The world's first RISC-V servers available in the cloud.* https://labs.scaleway.com/en/em-rv1/

[57] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security*.

[58] Mark Seaborn. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html Retrieved on June 26, 2015.

[59] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. Silifuzz: Fuzzing cpus by proxy. *arXiv:2110.11519* (2021).

[60] Agam Shah. 2023. *China Deploys Massive RISC-V Server in Commercial Cloud.* https://www.hpcwire.com/2023/11/08/china-deploys-massive-risc-v-server-in-commercial-cloud/

[61] SiFive. 2021. https://starfivetech.com/uploads/u54_core_complex_manual_21G1.pdf

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

[63] SiFive. 2024. HF106 Datasheet. https://www.sifive.com/document-file/hifive-premier-p550-datasheet

[64] Sipeed. 2021. Sipeed Wiki. https://wiki.sipeed.com/en/index.html

[65] Sipeed. 2022. RISC-V 64bit chip (C910) run Android 10. https://twitter.com/SipeedIO/status/1457529282134089734

[66] Sipeed. 2023. *Lichee Console 4A.* https://sipeed.com/licheepi4a

[67] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security*.

[68] SpacemiT. 2024. SpacemiT Key Stone K1. https://www.spacemit.com/en/key-stone-k1/

[69] Fredrik Strupe and Rakesh Kumar. 2020. Uncovering hidden instructions in Armv8-A implementations. In *HASP*.

[70] T-Head. 2021. openC910. https://github.com/T-head-Semi/openc910

[71] T-Head. 2022. C906. https://www.t-head.cn/product/c906

[72] T-Head. 2022. T-Head Extension Spec. https://github.com/T-head-Semi/thead-extension-spec

[73] Fabian Thomas, Michael Torres, Daniel Moghimi, and Michael Schwarz. 2025. ExfilState: Automated Discovery of Timer-Free Cache Side Channels on ARM CPUs. In *CCS*.

[74] M Caner Tol, Kemal Derya, and Berk Sunar. 2025. μRL: Discovering Transient Execution Vulnerabilities Using Reinforcement Learning. *arXiv preprint arXiv:2502.14307* (2025).

[75] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[76] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and verification of side-channel security for open-source processors via leakage contracts. In *CCS*.

[77] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual, Vol. I: Unprivileged ISA, Version 20191213.

[78] Andrew Waterman, Krste Asanović, and John Hauser. 2021. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203.

[79] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. 2015. The RISC-V compressed instruction set manual, version 1.7. *EECS Department, University of California, Berkeley* (2015).

[80] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security*.

[81] Xcalibyte. 2022. Roma Laptop Pre-order. https://xcalibyte.com.cn/en/roma-preorder/

[82] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *USENIX Security*.

[83] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*.

A C906 CPU-halting Instructions

Listing 4 lists other broken `XTheadMemIdx` instructions on the C906, any of which can replace `th.lbib` in Listing 2 to halt the CPU.

B Cascade Bug Rediscovery

Table 4 shows bug discovery statistics for RISCover, with weighted generation speedups and estimated time to bug.

Table 4: We report statistics on bugs found and rediscovered by RISCover. We record the number of instructions needed until the first trigger, the theoretical runtime on the slowest CPU, the C906, and the speedup of weighted sequence generation.

Cascade Bug ID	Bug Description	Triggered	Instr. Needed	Weighted Instr. Needed	Weighted Speedup
V1	Non-deterministic conversion from single-precision float to int	✓	35 (≈ 1.92 ms)	31 (≈ 1.70 ms)	**12.9%**
V2	fmin with one NaN does not always return the other operand	✓	667 (≈ 36.66 ms)	579 (≈ 31.82 ms)	**15.2%**
V3	Conversion from double to float may pollute the mantissa	✓	143 (≈ 7.86 ms)	123 (≈ 6.76 ms)	**16.3%**
V4	Dependent arithmetic/muldiv FPU operations may yield incorrect results	✓	3782 (≈ 207.87 ms)	2836 (≈ 155.88 ms)	**33.4%**
V5	Equal registers may be considered distinct by fle.s and feq.s	✓	234 (≈ 12.86 ms)	194 (≈ 10.66 ms)	**20.6%**
V6	flt.s may return 1 when operands are equal	✓	978 (≈ 53.75 ms)	846 (≈ 46.50 ms)	**15.6%**
V7	Under some microarchitectural conditions, square root may be imprecise	✓	68 (≈ 3.74 ms)	61 (≈ 3.35 ms)	**11.5%**
V8	Single-precision muldiv followed by conversion may pollute the mantissa	✓	14196 (≈ 780.26 ms)	12713 (≈ 698.75 ms)	**11.7%**
V9	Dependent arithmetic/muldiv operations may cause largely wrong output	✓	2388 (≈ 131.25 ms)	1667 (≈ 91.62 ms)	**43.3%**
V12	Hang on speculatively executed compressed FPU instructions	✗	-	-	-
V14	Some register comparisons are still incorrect despite a partial fix	✓	12 (≈ 0.66 ms)	11 (≈ 0.60 ms)	**9.1%**
P6	Spurious exception when decoding fence instructions	✓	44 (≈ 2.42 ms)	52 (≈ 2.86 ms)	-15.4%
K1	RaWaW double-hazard may cause a wrong register value to be forwarded	✓	105 (≈ 5.77 ms)	84 (≈ 4.62 ms)	**25.0%**
K5	Incorrect decode logic for fence and fence.i	✓	67 (≈ 3.68 ms)	82 (≈ 4.51 ms)	-18.3%
C1	Double-precision multiplications yield wrong sign when rounding down	✓	138 (≈ 7.58 ms)	219 (≈ 12.04 ms)	-37.0%
C2	Single-precision floating-point operations may treat NaNs as zeros	✓	1323 (≈ 72.72 ms)	1317 (≈ 72.39 ms)	**0.5%**
C3	Division by NAN incorrectly sets NX and NV fflags	✓	609 (≈ 33.47 ms)	555 (≈ 30.50 ms)	**9.7%**
C4	The inexact (NX) flag not set in case of overflow or underflow	✓	12 (≈ 0.66 ms)	11 (≈ 0.60 ms)	**9.1%**
C5	Division of zero by zero incorrectly sets the DZ flag	✓	14203 (≈ 780.64 ms)	13428 (≈ 738.05 ms)	**5.8%**
C6	Plus and Minus infinity microarchitectural structures are inverted	✓	21272 (≈ 1169.18 ms)	20268 (≈ 1113.99 ms)	**5.0%**
C7	Infinities are not rounded properly and stick to infinity	✓	40572 (≈ 2229.97 ms)	41818 (≈ 2298.45 ms)	-3.0%
C10	Under some microarchitectural circumstances, wrong NAN conversion	✓	2435 (≈ 133.84 ms)	2131 (≈ 117.13 ms)	**14.3%**
B1	Static rounding is ignored for fdiv.s and fsqrt.s	✓	71 (≈ 3.90 ms)	60 (≈ 3.30 ms)	**18.3%**
-	GhostWrite	✓	38 (≈ 2.09 ms)	39 (≈ 2.14 ms)	-2.6%
-	C906 halting sequence	✓	470496 (≈ 25859.95 ms)	354672 (≈ 19493.90 ms)	**32.7%**

```
#               C910
# signum:                         OK
# --------------------------------
#               C906
# signum:                    SIGSEGV
# si_addr:      0x8000000000000000
# si_pc:                 0xe100178
# si_code:                     0x1
instr_seq: [ vse1024.v ]
regs: gp: 0x8000000000000000
```

Listing 3: Reproducer that hints at GhostWrite. The C910 executes the vector-store instruction, while the C906 faults.

Table 5: Tested RISC-V boards with CPUs from 3 vendors.

ID	Board Model	CPU	Vendor	Ext.	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	HiFive Premier P550	P550	SiFive	-	16 GB	FUSDK	6.6.21
D	Sipeed Nezha			†, ‡	1 GB	Debian 13	5.14.0
E	Lichee RV Dock	C906	T-Head	†, ‡	512 MB	Debian 11	5.4.61
F	Lichee RV Dock			†, ‡	512 MB	Debian 12	5.14.0
G	CanMV-K230	C908	T-Head	†	512 MB	Debian 13	5.10.4
H	BeagleV Ahead			†, ‡	4 GB	Ubuntu 23.04	5.10.113
I	LicheePi4A	C910	T-Head	†, ‡	8 GB	Debian 12	5.10.113
J	LicheePi4A			†, ‡	16 GB	NixOS	5.10.113
K	Milk-V Meles			†, ‡	8 GB	Debian 12	5.10.113
L	Banana Pi BPI-F3	X60	SpacemiT	v	4 GB	Armbian	6.1.15

† zfh, XTheadMemIdx ‡ XTheadVec

C GhostWrite Sample Reproducer

Listing 3 shows an example of a GhostWrite reproducer file.

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

Listing 4: List of instructions from the `XTheadMemIdx` extension that can be used in Listing 2 to halt the C906 CPU.

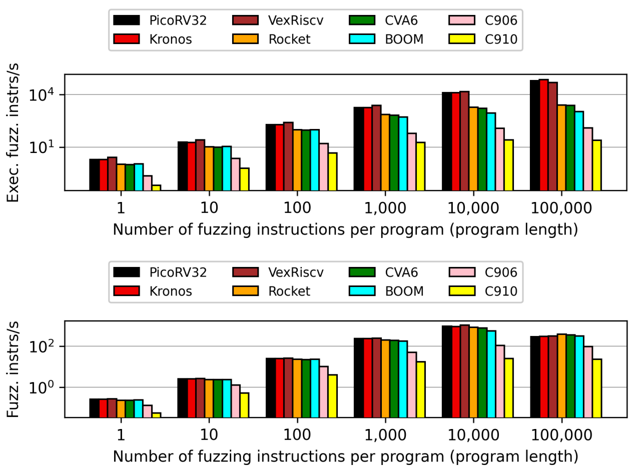

Figure 5: Raw executed instructions per second (top) and Cascade fuzzing throughput (bottom) for openC906 and C910.

D Details on Used Boards

Table 5 lists the boards and CPUs we used for evaluating RISCover.

E Cascade Fuzzing Throughput Evaluation

Figure 5 shows Cascade throughput on openC910 and C906.