

# TrapFuzz: Understanding and Detecting Recovery Failure Issues in Multi-Trap Mechanisms of CPUs

Yuanliang Chen

Renmin University of China  
Beijing, China  
sard.chen@gmail.com

Zhen Yan

Tsinghua University  
Beijing, China  
yanzhen20011121@163.com

Xudong Zhang

Tsinghua University  
Beijing, China  
zxd\_canxin@qq.com

Fuchen Ma

Tsinghua University  
Beijing, China  
fuchenma525@gmail.com

Zehong Yu\*

Tsinghua University  
Beijing, China  
yzhddding@gmail.com

Yu Jiang\*

Tsinghua University  
Beijing, China  
jiangyu198964@126.com

## Abstract

Modern RISC-V processors support a rich trap mechanism that handles exceptions and interrupts across multiple privilege levels. While the ISA specification defines the expected behavior of nested traps, real CPU implementations introduce many microarchitectural optimizations for trap entry, context saving, and trap return, which makes multi-trap handling particularly error-prone. However, existing CPU testing techniques rarely explore nested traps and therefore miss a large space of recovery-failure issues.

This paper first presents a study of real-world multi-trap issues in six widely used open-source RISC-V CPUs, analyzing their symptoms, triggering conditions, and root causes. We find that deeply nested trap sequences, where exceptions or interrupts recur inside trap handlers, are essential for exposing these correctness issues, yet are difficult to generate due to the combinatorial interactions of trap-entry constraints, privilege rules, delegation settings, and CSR state transitions. Guided by this study, we design TrapFuzz. TrapFuzz first builds a Multi-Trap Sequence Model that encodes trap-entry, delegation, priority, and trap-return constraints to generate legal nested-trap sequences. It then uses trap-state volatility as feedback to guide execution toward deeper and more error-prone trap interactions. Finally, TrapFuzz employs an MTSM-derived recovery oracle that validates CSR consistency, privilege-flow legality, and forward progress. In total, TrapFuzz detected 21 new recovery-failure issues in eight real-world RISC-V CPUs.

**CCS Concepts:** • **Networks** → **Protocol testing and verification**; • **Hardware** → **Fault models and test metrics**.

\*Zehong Yu and Yu Jiang are Corresponding authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04

<https://doi.org/10.1145/3845814.3850036>

**Keywords:** RISC-V CPUs, Fuzzing, Bug Detection

## ACM Reference Format:

Yuanliang Chen, Xudong Zhang, Zehong Yu, Zhen Yan, Fuchen Ma, and Yu Jiang. 2027. TrapFuzz: Understanding and Detecting Recovery Failure Issues in Multi-Trap Mechanisms of CPUs. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27)*, April 11–15, 2027, Heraklion, Greece. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3845814.3850036>

## 1 Introduction

RISC-V has rapidly evolved from a research-oriented ISA into a widely adopted open instruction set used in embedded, IoT, industrial, and increasingly high-performance systems [6, 13]. Its trap mechanism, which handles exceptions and interrupts across privilege levels, is critical to reliability and availability. Since trap handling governs fault recovery, privilege enforcement, and context switching, errors in these mechanisms can cause silent state corruption, execution deadlocks, or privilege inconsistencies, posing serious risks to deployed CPUs [23, 24, 33].

The trap mechanism in RISC-V is inherently complex [22]. Although the ISA defines architectural behavior, real CPU implementations introduce many microarchitectural optimizations, such as speculative trap entry, compressed context saving, interrupt replay, and pipeline-level hazard handling [5, 45, 48], which can introduce subtle trap-handling bugs. In practice, traps may recur inside trap handlers, forming a multi-trap mechanism in which exceptions or interrupts nest across privilege levels and depend on correct trap entry, delegation, prioritization, and return for proper recovery. Such nested behaviors are important in realistic systems with frequent interrupts and privilege transitions. We define **Recovery Failure Issues** as implementation faults in multi-trap mechanisms that cause a CPU to recover incorrectly from exceptions or interrupts across multiple trap levels.

To understand these issues, we conducted a comprehensive study of real-world multi-trap bugs across six widely used open-source RISC-V CPUs. For each bug case, we manually analyzed its report and corresponding fix to understand

its symptoms, root causes, and triggering conditions. Our study reveals two key insights. ① Most recovery failures are initially silent: 43/47 bugs (92%) do not immediately crash or hang when corrupted nested-trap state is produced. Of the 47 bugs, 35 (74%) are persistently silent, continuing execution incorrectly throughout the observed run, while the remaining 8 initially silent bugs later escalate to a crash or hang. Their root causes violate four key rule types of nested-trap execution: trap-entry constraints (how a trap initializes `mstatus` and `mepc`), interrupt-exception privilege rules (which traps may preempt others across privilege levels), delegation settings (whether a trap is routed to machine or supervisor mode), and return-state dependencies (how `mstatus` and the PC must be restored on `mret/sret`). ② While a small fraction of issues can be triggered by one or two simple traps, the majority (39/47, 83%) require longer nested-trap sequences ( $\geq 3$  traps) that systematically explore combinations of these four rule types.

Researchers have proposed techniques for validating RISC-V CPUs. Instruction-generation fuzzers such as DifuzzRTL [14], MorFuzz [44], Cascade [37], and INSTILLER [46] generate randomized or coverage-guided instruction streams and have uncovered functional bugs. However, because these approaches mainly exercise flat instruction sequences, they trigger only shallow traps and rarely construct the nested-trap chains required to expose deep recovery-failure issues. Robustness-testing tools such as WhisperFuzz [1] and HwFuzz [47] stress microarchitectural state or timing behavior to expose transient-error sensitivity. Although useful for resilience evaluation, they are not guided by architectural trap semantics and may create states unreachable by legal exception, interrupt, delegation, or trap-return behavior. Formal and semi-formal methods can verify specified hardware properties, but applying them to multi-trap recovery requires manually defining trap-entry, delegation, priority, nested-preemption, and return-state rules. Additionally, existing CPU testing tools lack an oracle for detecting silent recovery-failure issues in multi-trap mechanisms.

Effectively detecting recovery-failure issues in RISC-V CPUs requires addressing two key challenges: (1) **The first challenge** is the need for a precise recovery oracle that can detect silent anomalies in nested-trap execution. Most recovery failures cause no crashes or hangs, yet still manifest as incorrect CSR restoration, illegal privilege transitions, or loss of forward progress. Without such an oracle, these silent faults cannot be reliably distinguished from normal architectural behavior. (2) **The second challenge** is generating high-quality nested-trap sequences. Recovery failures are typically triggered by specific long trap chains shaped by trap-entry constraints, interrupt-exception privilege rules, delegation settings, and return-state dependencies. The combination of these rules greatly enlarges the exploration space, making naive or random testing ineffective at reaching deep and error-prone nested-trap handling logic.

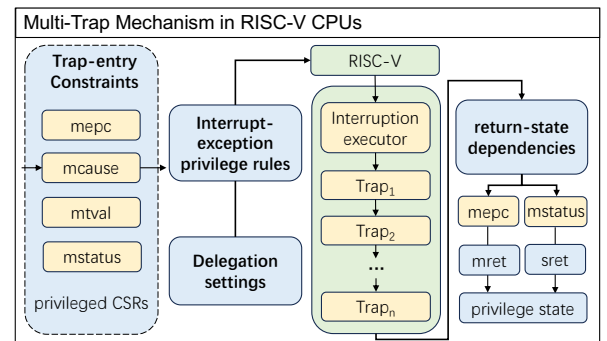
To tackle these challenges, we introduce TrapFuzz, a framework for detecting recovery-failure issues in multi-trap mechanisms of RISC-V CPUs. Guided by our empirical study, TrapFuzz first builds a Multi-Trap Sequence Model (MTSM) that encodes trap-entry rules, priority relations, delegation behavior, and return-state constraints, collapsing the large nested-trap space into a tractable legal state graph. TrapFuzz then uses trap-state volatility, computed from legal outgoing transitions, privilege-level dispersion, and recovery-critical CSR impact, to guide the fuzzer toward deeper and more error-prone trap interactions. Finally, TrapFuzz employs an MTSM-derived recovery oracle composed of a State-Consistency Detector, a Privilege-Flow Detector, and a Liveness Detector. These detectors validate CSR-state consistency, privilege-transition legality, and forward progress, enabling detection of recovery-failure issues.

We implemented TrapFuzz and evaluated it on eight widely used RISC-V CPUs. Compared with state-of-the-art CPU testing tools, Cascade [37], RISC-V DV [3], WhisperFuzz [1], and HwFuzz [47], TrapFuzz detects 3x–5x more issues and achieves 13%–26% higher coverage of trap-handling logic. In total, TrapFuzz identified 21 new recovery-failure issues, including 3 in NanoRV32 [7], 5 in CVA6 [9], 3 in VexRiscv [11], 2 in Rocket [8], 2 in Xiangshan [10], 1 in Kronos [38], 3 in BOOM [35], and 2 in srv32 [17].

In summary, we make three key contributions:

- We conduct a detailed study of real-world multi-trap recovery-failure issues in six widely used RISC-V CPUs and summarize their symptoms and triggering conditions.
- We design TrapFuzz, a testing framework that synthesizes high-quality nested-trap sequences and precisely detects recovery failure issues hidden in RISC-V CPUs.
- We implement and evaluate TrapFuzz on eight RISC-V CPUs. We will open-source TrapFuzz<sup>1</sup> for practical usage. For now, TrapFuzz has successfully detected 21 new issues.

## 2 Background



**Figure 1.** Typical multi-trap behavior in RISC-V CPUs, where each nested trap saves its privileged CSR context according to trap-entry, privilege, and delegation rules.

<sup>1</sup>TrapFuzz: <https://anonymous.4open.science/r/TrapFuzz-1452/>

**Multi-Trap Mechanism in RISC-V CPUs:** Trap handling is a fundamental control mechanism in RISC-V processors. When an exception or interrupt occurs, the CPU saves the interrupted state into privileged CSRs such as `mepc`, `mcause`, `mtval`, and trap-related fields in `mstatus`. This process is governed by four key rule classes: trap-entry constraints, interrupt-exception privilege rules, delegation settings (e.g., `medeleg` and `mideleg`), and return-state dependencies, which determine how traps are prioritized, routed, nested, and returned across privilege levels [16, 20, 40]. When a new exception or interrupt occurs inside a trap handler, the CPU enters a multi-trap execution chain. Correct recovery requires each trap context to be saved and restored precisely: `mepc` must be preserved across trap layers, `mstatus` fields such as `MPP` and `MPIE` must be updated correctly, and `mret/sret` must restore privilege and resume control flow. This makes multi-trap behavior more complex than single-level traps. Real implementations further increase this complexity through optimizations such as speculative trap entry, compressed CSR stacks, and pipeline replay [26, 36, 41]. As a result, bugs that do not appear under a single trap may surface only under recursive traps, leading to corrupted trap context, incorrect privilege transitions, failed trap return, or loss of forward progress [19, 27]. Because such failures depend on specific combinations of trap types, privilege interactions, and timing, existing CPU testing techniques rarely explore this space systematically.

### 3 Motivation Study

To better understand the characteristics of recovery-failure issues in practical RISC-V processors and to guide the design of our detection strategy, we conducted an empirical analysis of six widely used open-source RISC-V CPUs: NanoRV32, CVA6, VexRiscv, Rocket, Xiangshan, and Kronos. We collected bug cases from their public issue trackers (GitHub issues), patch histories, and discussion threads. After manually filtering out unrelated functional bugs, we identified 47 real-world issues that specifically relate to recovery failures in multi-trap mechanisms, as shown in Table 1.

**Table 1.** Number of recovery-failure issues we analyzed.

| NanoRV32 | CVA6 | VexRiscv | Rocket | Xiangshan | Kronos | Total |
|----------|------|----------|--------|-----------|--------|-------|
| 9        | 11   | 6        | 7      | 8         | 6      | 47    |

#### 3.1 Symptoms of Recovery-Failure Issues

We first examined the observable consequences, internal symptoms and root causes of each recovery failure issue to help design a precise recovery oracle.

**Finding 1: silent but critical consequences.** *Most recovery-failure issues (43/47) are initially silent when the recovery failure occurs, and 35/47 issues (74%) remain persistently silent throughout the observed execution.*

Among the analyzed cases, 43/47 bugs (92%) are *initially silent*: the CPU does not crash or hang when corrupted nested-trap state is first produced. These faults may overwrite trap-entry CSRs, misconfigure privilege levels, or restore an incorrect `mepc`, allowing errors to propagate unnoticed. Among them, 35 bugs (74%) are *persistently silent*, continuing execution incorrectly throughout the observed run, while the remaining 8 initially silent bugs later escalate to a crash or hang. Despite lacking immediate visible symptoms, such inconsistencies can lead to privilege violations, illegal control transfers, or incorrect execution, making them more dangerous than explicit crashes or hangs.

**Table 2.** Symptoms, internal state, and root causes of issues.

| Symptoms | Internal State         | Root Causes                                 | Number |
|----------|------------------------|---------------------------------------------|--------|
| Crash    | -                      | illegal jump                                | 1      |
| Hang     | -                      | Infinite loop                               | 3      |
| Silence  | overwritten trap state | violate trap-entry constraints              | 16     |
|          | incorrect privilege    | violate interrupt-exception privilege rules | 9      |
|          | CSR inconsistency      | violate return-state dependencies           | 12     |
|          | wrong delegation path  | violate delegation settings                 | 6      |

**Finding 2: obvious internal state.** *Most recovery failures produce clear inconsistencies in internal trap states, including overwritten trap-entry CSRs, incorrect privilege levels, CSR corruption across trap layers, or misrouted delegation paths.*

Although the symptoms of these recovery failures are subtle, through further analysis of each failure’s internal states during nested-trap execution, we find that before these issues lead to severe consequences, they consistently introduce observable inconsistencies in trap-related architectural state. Specifically, we observe four dominant classes of internal symptoms: overwritten trap-entry CSRs (34%), incorrect privilege levels after trap entry or return (19%), inconsistencies in critical CSRs such as `mepc` and `mstatus` across trap layers (26%), and misrouted delegation paths that place traps at incorrect privilege levels (13%). These inconsistent states typically appear early, within the first four trap layers, and often remain silent until later execution. This finding suggests that monitoring internal trap-state evolution (e.g., CSR correctness, privilege flow, delegation behavior) is an effective way to identify recovery failures.

**Finding 3: four types of root causes.** *Most recovery-failure issues stem from violations of four fundamental rule types that govern correct nested-trap execution in RISC-V.*

We conducted a root cause analysis for each recovery-failure issue by examining the patches and discussions associated with their fixes. We find that 34% of failures stem from violating trap-entry constraints, typically causing overwritten or corrupted trap-entry CSRs. Another 19% arise from incorrect interrupt-exception privilege handling, where illegal preemption leads to wrong privilege states. Additionally, 26% of issues are caused by broken return-state dependencies, resulting in inconsistent `mepc` or `mstatus` after nested `mret/sret`. The remaining 13% arise from incorrect delegation settings, routing traps to unintended privilege levels.

### 3.2 Triggering of Recovery-Failure Issues

We further analyzed the triggering conditions of each issue to understand the input patterns required to expose them, which guided our testcase generation strategy.

**Finding 4: nested triggering requirement.** *Most recovery failures (45/47) require nested trap sequences to be triggered.*

We found that only a small proportion (4%) of issues (2/47) can be triggered by a single trap, such as illegal instructions or simple load/store faults. The vast majority require specific combinations of synchronous exceptions, asynchronous interrupts, privilege transitions, and CSR side effects occurring in close temporal proximity. Hence, exploring multi-trap behavior is essential for revealing such recovery-failure issues.

**Table 3.** Nested trap depth and required trap-type diversity for triggering these recovery-failure issues.

| Trap Depth | Number of Issues | Required Trap Types                                                                                                                         |
|------------|------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| 1          | 2                | Illegal instruction, Load/Store access fault                                                                                                |
| 2          | 6                | Timer interrupt, Illegal instruction, External interrupt<br>Load fault, Environment call, Page fault                                        |
| 3          | 19               | External interrupt, Illegal instruction<br>Load/Store access fault, Page fault, Environment call                                            |
| 4          | 16               | Timer interrupt, External interrupt, Illegal instruction, Page fault<br>Load/Store access fault, Environment call, Instruction Access Fault |
| 5          | 4                | Timer interrupt, External interrupt, Illegal instruction, Page fault<br>Load/Store access fault, Environment call, Instruction Access Fault |

**Finding 5: short triggering depth but diverse trap-type combinations.** *All recovery failures can be triggered with no more than five nested trap layers, but require diverse trap-type combinations (e.g., illegal instruction, external interrupt).*

We found that 57% of recovery-failure issues are triggered within three nested traps, and the remaining 43% within four or five traps. This suggests that excessive sequence length is not the primary factor; instead, the triggering sequences consistently require diverse trap types and strict ordering constraints. For example, a CVA6 bug [30] involving incorrect MPIE restoration is triggered by an illegal instruction inside a timer-interrupt handler followed by an external interrupt, while a Rocket issue [42] appears only when a load fault occurs inside a high-priority interrupt handler. Across all cases, the specific combination and ordering of trap events, rather than sequence length alone, are essential for triggering such failures.

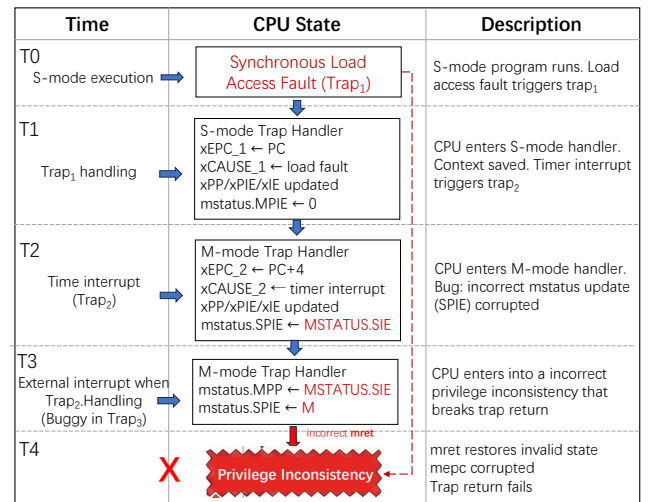
**Finding 6: volatile trap-state propagation windows.** *Recovery failures often arise when a new trap occurs during a volatile window, typically no more than 10 cycles, in which trap-entry updates have taken effect but the CPU has not yet reached a stable post-trap state.*

By analyzing fine-grained traces, we found that this trap-state propagation window appears when a new trap arrives shortly after the handler of a previous trap updates fields such as `mstatus`, delegation bits, or privilege indicators, but before these values are fully committed or restored. The window itself is legal under the RISC-V specification and does not indicate incorrect behavior, but it creates a period of high volatility in which small execution deviations can

propagate across trap layers. We observed that 79% of these recovery failures are triggered only when nested traps occur inside such volatile windows. This observation suggests that TrapFuzz can use trap-state volatility as feedback to optimize testcase generation.

### 3.3 A Motivating Example

Recovery-failure issues in multi-trap mechanisms are difficult to expose and may lead to severe consequences in RISC-V processors. One example is a CVA6 bug [43], where incorrect `mstatus` restoration during nested traps causes a privilege inconsistency that breaks trap return and eventually makes the CPU jump to an incorrect privilege level. Figure 2 illustrates the key steps that trigger this bug.



**Figure 2.** A recovery-failure bug in a multi-trap mechanism. Incorrect `mstatus` restoration breaks trap return.

The CPU initially executes in S-mode, where a synchronous load fault (Trap<sub>1</sub>) occurs. On trap entry, the CPU saves context into the target-level EPC/CAUSE/TVAL CSRs and updates the corresponding `xPP/xPIE/xIE` fields according to the privilege level. While executing the S-mode trap handler, a timer interrupt triggers a second trap (Trap<sub>2</sub>) in M-mode. Shortly after, an external interrupt causes a third trap (Trap<sub>3</sub>). For each trap entry, the CPU saves the current context and updates the corresponding `mstatus` fields. According to the RISC-V specification, the CPU should unwind these traps in order: Trap<sub>3</sub> returns to Trap<sub>2</sub> via `mret`, Trap<sub>2</sub> restores the privilege state and resumes Trap<sub>1</sub>'s S-mode handler, and Trap<sub>1</sub> finally returns to the original program. However, in this buggy implementation, Trap<sub>3</sub> incorrectly overwrites the MPP and SPIE fields in `mstatus`, contaminating the state later used by Trap<sub>2</sub> and Trap<sub>1</sub>. As shown in Figure 2, this incorrect privilege metadata propagates across nested traps. Consequently, `mret` attempts to return to an invalid privilege level, causing trap-return failure and a corrupted `mepc`.

This case highlights two key lessons. First, recovery-failure bugs often hide in deep multi-trap paths and require carefully aligned nested-trap sequences rather than single-level events. Here, the bug appears only when a synchronous exception is followed by a timer interrupt and then an external interrupt in close succession, forming a specific three-level nested trap chain. Second, a precise recovery oracle is essential for detecting such issues. In this case, the CPU neither crashes nor hangs; instead, subtle inconsistencies in `mstatus` and `mepc` silently corrupt internal state. Without fine-grained monitoring of CSR evolution, privilege transitions, and trap-entry/return behavior, the bug would remain hidden.

### 3.4 Limitation of Existing Methods

Existing CPU testing techniques are effective for functional validation but fall short in testing multi-trap mechanisms. Two mainstream categories dominate current practice, yet neither systematically explores multi-trap behaviors.

**Method 1: CPU Instruction Generation.** Instruction-generation tools such as DifuzzRTL [14], MorFuzz [44], Cascade [37], and INSTILLER [46] output randomized or coverage-guided instruction streams and have uncovered many functional bugs in RISC-V CPUs. However, they mainly exercise flat execution paths and thus trigger shallow traps. Because they do not model trap-entry rules, privilege transitions, or delegation behavior, they rarely construct the ordered nested-trap chains needed to expose recovery-failure issues. As a result, they provide limited coverage of deep multi-trap interactions and miss bugs requiring specific trap sequences.

**Method 2: Other CPU Robustness and Verification Methods.** Robustness-testing tools such as WhisperFuzz [1] and HWFuzz [47] stress microarchitectural state or timing behavior to expose transient-error sensitivity. However, they are not guided by architectural trap semantics and may create states unreachable by legal exception, interrupt, delegation, or trap-return behavior. Formal and semi-formal methods, such as riscv-formal [39], Kami [4], and UCLID5 [31], can verify specified hardware properties, but multi-trap recovery requires manually specifying trap-entry, delegation, priority, nested-preemption, and return-state rules.

**Key Insight of TrapFuzz.** The fundamental limitation of existing methods is that they do not treat multi-trap recovery as a rule-constrained recovery process. Instruction generators mainly explore instruction streams, robustness-testing tools may create states unreachable by legal trap behavior, and formal methods require manually specified multi-trap recovery properties. As a result, they cannot systematically construct legal nested-trap chains or validate whether recovery remains semantically correct. TrapFuzz addresses this gap by encoding trap-entry rules, priority relations, delegation behavior, and return-state constraints into a Multi-Trap Sequence Model (MTSM), and by using trap-state volatility as feedback to guide exploration toward deeper and more error-prone nested-trap interactions.

## 4 TrapFuzz Design

**Design goal:** From our perspective, a practical recovery failure detection tool should have the following properties.

- *General:* TrapFuzz is designed to detect recovery-failure issues in multi-trap mechanisms across diverse practical RISC-V CPUs, from in-order microcontrollers such as NanoRV32 to pipelined out-of-order processors such as Xiangshan and Kronos. The framework is independent of specific microarchitectural optimizations and applies to various core designs (e.g., CVA6, Rocket, and VexRiscv).
- *Efficient:* TrapFuzz is able to frequently exercise the multi-trap handling logic and effectively detect recovery failures in real-world RISC-V CPUs within 24 hours.
- *Accurate:* TrapFuzz is designed to have satisfying precision and recall to avoid reporting false positives.

### 4.1 TrapFuzz Workflow

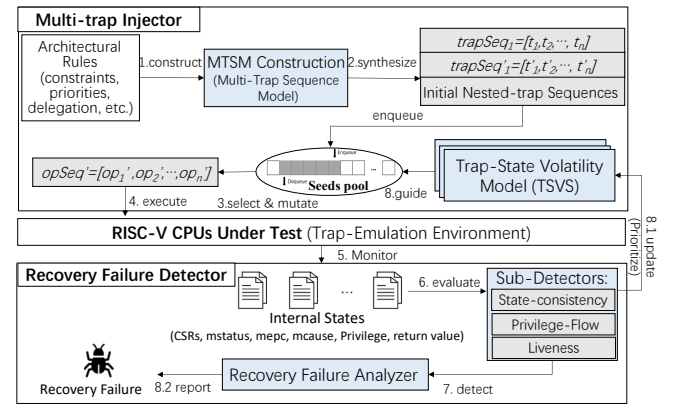


Figure 3. The workflow of TrapFuzz.

Figure 3 illustrates the workflow of TrapFuzz, which consists of two key components: the Multi-Trap Injector for generating nested-trap sequences and the Recovery Failure Detector for detecting recovery failures. (1) TrapFuzz constructs the Multi-Trap Sequence Model (MTSM) from trap-related clauses in the RISC-V privileged specification, including trap-entry constraints, priority rules, delegation settings, and trap-return constraints. (2) Using the MTSM, TrapFuzz synthesizes an initial corpus of legal nested-trap sequences and stores them in the seed pool. (3) The Multi-Trap Injector selects and mutates seeds to generate new sequences, guided by the Trap-State Volatility Score (TSVS), which favors states with more legal outgoing transitions, wider privilege changes, and greater impact on recovery-critical CSRs. Each sequence is then materialized into executable inputs, including an assembly/C program, trap handlers, initial CSR and memory/page-table states, and interrupt-timing configuration. (4) The target CPU executes the generated binary while TrapFuzz injects scheduled interrupts through its existing simulation interface in a controlled RTL simulation environment. (5) TrapFuzz monitors CSR updates (e.g., `mstatus`,

mepc, and mcause), privilege transitions, trap entries, and trap-return behavior. (6) The Recovery Failure Detector analyzes execution using three sub-detectors: State-Consistency, Privilege-Flow, and Liveness. (7) Silent anomalies, such as incorrect CSR restoration, privilege inconsistency, or forward-progress violations, are reported as recovery-failure bugs. (8) Sequences that improve TSVS coverage or trigger new oracle violations are prioritized and reinserted into the seed pool. TrapFuzz repeats steps (3)–(8) until testing terminates.

## 4.2 Multiple Trap Injector

Nested-trap behavior in RISC-V CPUs arises from the interaction between trap events and evolving privileged state during execution. Because trap types, nesting depths, delegation paths, and return states jointly affect recovery logic, the exploration space is highly complex and hard to traverse systematically. To address this, TrapFuzz constructs a Multi-Trap Sequence Model (MTSM) that encodes trap-entry rules, priority relations, delegation behavior, and return-state constraints as guarded state transitions. Guided by this model, the Trap Injector synthesizes and mutates multi-level trap sequences, using the Trap-State Volatility Score (TSVS) to prioritize architecturally legal executions that are likely to traverse volatile recovery states.

**Multi-Trap Sequence Model (MTSM).** TrapFuzz builds MTSM from the RISC-V privileged specification [34]. MTSM abstracts trap-related architectural behavior into states and guarded transitions, while the construction process that translates specification clauses into these transitions is described below. Formally, MTSM is a finite-state transition system  $M = (S, \mathcal{T}, s_0)$ , where  $S$  is the set of abstract trap states,  $\mathcal{T}$  is the set of legal trap-related transitions, and  $s_0$  is the initial trap state under the target CPU configuration. Each state is represented as  $s = \langle priv, phase, cause, deleg, ien, csr, depth \rangle$ , where *priv* is the current privilege mode, *phase* denotes whether the CPU is in normal execution or trap-handler execution, *cause* records the active or pending trap cause, *deleg* abstracts delegation bits such as medeleg, mideleg, *ien* abstracts interrupt pending and enable bits, *csr* abstracts recovery-critical CSR fields, and *depth* records the current nesting depth. Each transition  $\tau \in \mathcal{T}$  has the form  $\tau = (s, e, g, u, s')$ , where  $e$  is a trap-related event,  $g$  is a guard predicate,  $u$  is an update relation, and  $s'$  is the next abstract state. A transition is legal only if  $g(s, e)$  holds. The guard captures state-transition constraints such as privilege mode, trap cause, delegation bits, interrupt pending and enable bits, trap priority, supported optional features, and valid return state. The update relation captures the corresponding architectural effects, including writes to xEPC/xCAUSE/xTVAL, updates to xPP/xPIE/xIE, privilege-mode changes, and nesting-depth changes. TrapFuzz instantiates these constraints using the target CPU configuration. Thus, unsupported modes, delegation paths, virtual-memory traps, or trap causes do not

generate transitions. For WARL/WLRL CSR fields, generated and observed values are normalized according to target-supported encodings.

**Constructing transitions from the specification.** TrapFuzz constructs  $\mathcal{T}$  by translating trap-related clauses in the RISC-V privileged specification into guarded updates. The translation follows a clause-to-rule process. First, TrapFuzz selects clauses that change recovery-critical state, such as exception/interrupt entry, interrupt delivery, delegation, priority selection, and mret/sret. Second, for each selected clause, TrapFuzz separates its semantics into three parts: the trigger, the precondition, and the effect. The trigger becomes the event  $e$ , such as taking an exception, taking an interrupt, or executing xRET. The precondition becomes the guard  $g$ , such as privilege-mode checks, pending/enable checks, delegation-bit checks, priority checks, and return-state checks. The effect becomes the update  $u$ , such as writing xEPC/xCAUSE/xTVAL, updating xPP/xPIE/xIE, selecting the target privilege, restoring PC and privilege, or changing the nesting depth. Finally, TrapFuzz instantiates these guards and updates over the abstract state space  $S$  under the target CPU configuration. If a guard cannot be satisfied, the corresponding edge is not generated, which gives the legal transition set  $\mathcal{T}$ . Figure 4 illustrates this spec-to-MTSM construction process and shows how guards determine whether candidate transitions are kept or pruned.

(1) *Trap entry.* For an exception or interrupt with cause  $c$ , current PC  $pc$ , and trap value  $tval$  when applicable, TrapFuzz first computes the target privilege  $x$  using the delegation rule below, and then applies the trap-entry constraint. Its guard is:  $g_{\text{entry}} = \text{supported}(x) \wedge \text{legalEvent}(c, ien) \wedge \text{priorityOK}(c)$ . Here, *legalEvent* checks interrupt pending and enable bits for interrupts, and cause legality for synchronous exceptions. The predicate *priorityOK* checks whether  $c$  is the selected trap cause under the current priority rules when multiple events are pending. If the guard holds, the update enters the handler phase at privilege level  $x$ , saves the interrupted PC, cause, optional trap value, previous privilege, and interrupt-enable state into the target-level CSRs, increases the nesting depth, and clears xIE during trap entry. The notation xEPC, xCAUSE, xTVAL, xPP, xPIE, and xIE denotes the CSR fields at the target privilege level, such as mepc/mcause in M-mode and sepc/scause in S-mode.

(2) *Delegation resolution.* Delegation is evaluated as part of trap-entry construction and determines the target privilege  $x$ . For a synchronous exception delegated by medeleg, or an interrupt delegated by mideleg, TrapFuzz computes:

$$\text{target}(priv, c, deleg) = \begin{cases} S, & \text{delegateOK}(priv, c, deleg), \\ M, & \text{otherwise.} \end{cases}$$

The predicate *delegateOK* holds only when the trap comes from a lower privilege mode, the corresponding bit in medeleg or mideleg is set, and S-mode is supported by the target CPU. Otherwise, the trap is routed to M-mode. This rule removes

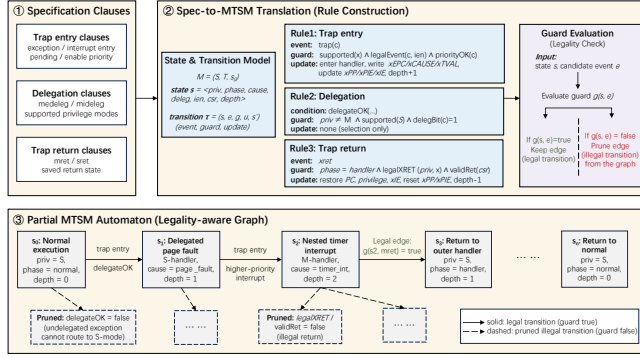


Figure 4. Spec-to-MTSM construction.

illegal routing choices from the MTSM graph, such as an undelegated exception entering S-mode or a delegated trap entering an unsupported mode.

(3) *Trap return*. For an xRET event, where x is M or S, TrapFuzz applies the trap-return constraint. Its guard is:

$$g_{ret} = (phase = handler) \wedge legalXRET(priv, x) \wedge validRet(csr).$$

The predicate legalXRET checks whether the return instruction is legal in the current privilege mode, and validRet checks whether the saved return state is valid. If the guard holds, the update restores the next PC and privilege mode from xEPC and xPP, sets xIE to the saved interrupt-enable value in xPIE, sets xPIE to 1, resets xPP according to the privileged specification and target configuration, decreases the nesting depth, and returns either to an outer trap handler or to normal program execution.

**Initial trap sequence generation.** Using MTSM, TrapFuzz synthesizes a graph of nested-trap interactions. Each node in the graph is an abstract trap state  $s \in S$ , and each candidate edge is produced by applying a trap-related event to that state, as shown in Figure 4. TrapFuzz keeps the edge as a legal transition  $\tau \in \mathcal{T}$  only when its guard is satisfied. This generation-stage check preserves architectural legality by pruning invalid cases such as disabled interrupts, unsupported delegation paths, illegal privilege transitions, and invalid mret/sret return states. Legal edges capture interactions such as exception-interrupt preemption, delegated trap redirection, and multi-level trap returns, yielding a compact nested-trap state graph from the large space of potential trap combinations. TrapFuzz generates its initial trap sequences by sampling bounded legal paths from this graph and materializing each path into an executable trap sequence (*trapSeq*). Guided by Finding 5 that most meaningful nested-trap behaviors arise within short interaction depths, TrapFuzz limits the length of these initial sequences to at most five traps, yielding legal seeds that exercise deeper and more error-prone regions of multi-trap handling logic.

**Trap-sequence materialization.** A trap sequence is not an abstract event label directly injected into the CPU, but a planned runtime chain of trap entries and returns realized by executable inputs. For example, a sequence may configure

medeleg/mideleg, enter S-mode, trigger a page fault by a load, enter the S-mode trap handler, enable a timer interrupt inside the handler, take an M-mode interrupt before returning, execute mret back to the S-mode handler, and finally execute sret to resume the original context. The sequence is therefore observed as an ordered trace of trap entries and trap returns. TrapFuzz materializes such a sequence as an assembly/C test program with trap handlers, initial CSR settings, memory/page-table states, and interrupt-timing configuration. Synchronous exceptions are induced by generated instructions and configured memory/privilege state, while asynchronous interrupts are delivered through existing test-bench interrupt lines or timer/PLIC models, without modifying internal RTL state. Thus, unlike plain instruction inputs, trap sequences include the trap-handling environment and timing configuration; unlike fault injection, TrapFuzz triggers traps through architecturally defined mechanisms rather than perturbing internal RTL state.

#### Algorithm 1: Trap-State Volatility Guided Fuzzing.

**Input** : *cpu*: RISC-V CPU under test  
*MTSM*: Multi-Trap Sequence Model  
*budget*: testing budget

**Output**:  $I_n$ : Recovery-Failure Issues

```

1  seedPool = initSequences(MTSM);
2  cov = 0;
3  while budget not exhausted do
4      trapSeq = seedPool.dequeue();
5      trapSeq' = mutate(trapSeq, MTSM);
6      async:
7          trace = cpu.execute(trapSeq');
8          state = MTSM.extractState(trace);
9      end async
10     vScore = MTSM.calcTSVS(state);
11     improved = updateTSVSCoverage(cov, vScore);
12     newBugs = checkOracle(trace, state, MTSM);
13      $I_n.append(newBugs)$ ;
14     if (improved = true) or (newBugs ≠ NULL) then
15         seedPool.enqueue(trapSeq');
16     end
17 end
    
```

**Trap-State Volatility Guided Fuzzing:** Inspired by Finding 6 that many failures arise within short volatile propagation windows, TrapFuzz uses trap-state volatility to guide exploration of the nested-trap space. At the core is the *Trap-State Volatility Score (TSVS)*, which estimates how sensitive a trap state is to subsequent legal trap interactions. A state has high volatility if small changes, such as delivering an interrupt, modifying delegation bits, or altering the return path, can substantially change privilege transitions or CSR evolution. For each trap state  $s$ , TrapFuzz computes TSVS

from three factors: (1) the number of legal outgoing transitions, (2) their privilege-level dispersion, and (3) their impact on recovery-critical CSRs. Let  $\text{Out}(s)$  denote the legal outgoing transitions. For each  $\tau \in \text{Out}(s)$ ,  $\Delta_{\text{priv}}(\tau)$  measures the normalized source-to-target privilege distance (0, 0.5, and 1 for zero-, one-, and two-level changes in the U/S/M hierarchy, respectively; 0 for single-mode targets).  $\Delta_{\text{csr}}(\tau)$  is the fraction of modified recovery-critical CSR fields, including xEPC, xCAUSE, xTVAL, xPP, xPIE, xIE, delegation, and interrupt pending/enable fields. TrapFuzz defines

$$\text{TSVS}(s) = |\text{Out}(s)| + \frac{\sum_{\tau \in \text{Out}(s)} \Delta_{\text{priv}}(\tau)}{\max(1, |\text{Out}(s)|)} + \frac{\sum_{\tau \in \text{Out}(s)} \Delta_{\text{csr}}(\tau)}{\max(1, |\text{Out}(s)|)}.$$

States with larger transition breadth, broader privilege dispersion, or higher CSR impact receive higher TSVS and are prioritized for mutation. TSVS is used as an exploration feedback signal; correctness is checked separately by the recovery oracle in Section 4.3.

Algorithm 1 outlines the fuzzing workflow. In each iteration, TrapFuzz dequeues a seed *trapSeq*, mutates it into a new sequence *trapSeq'* under MTSM constraints, and executes it on the target CPU. During execution, TrapFuzz records trap entries, CSR updates, privilege transitions, and return-state behavior at each trap level. After execution, it evaluates two complementary feedback signals. First, it checks whether *trapSeq'* reaches new high-TSVS states, indicating that the sequence has driven the CPU into previously unexplored volatile trap-state regions. Second, the recovery oracle checks for silent failures, including incorrect CSR evolution, illegal privilege flows, or loss of forward progress. If either condition holds, *trapSeq'* is treated as an interesting seed and reinserted into the pool for subsequent mutation and prioritization. By continuously favoring sequences that traverse high-volatility trap states, TrapFuzz efficiently explores deeper and more error-prone nested-trap behaviors, thereby uncovering subtle recovery failures.

### 4.3 Recovery Failure Detector

Inspired by Finding 2, the Recovery Failure Detector builds a spec-derived oracle from MTSM transitions and RISC-V privileged semantics. For each legal transition  $\tau = (s, e, g, u, s')$ , TrapFuzz derives expected recovery effects from  $u$ , including CSR updates, privilege changes, delegation, trap-return state, and forward progress. At runtime, the trace is mapped to an abstract MTSM path. TrapFuzz first validates that the generated sequence follows legal guarded transitions; illegal sequences are discarded rather than reported as failures. For legal sequences, a recovery failure is reported when observed CSR evolution, privilege flow, or liveness deviates from MTSM-derived expectations. Based on the symptoms and internal-state categories in Table 2, TrapFuzz uses three detectors: (1) the State-Consistency Detector for CSR consistency across trap entry and return; (2) the Privilege-Flow

Detector for privilege transitions and delegation; and (3) the Liveness Detector for forward progress after nested traps.

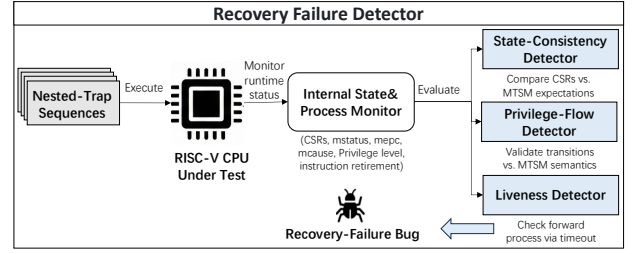


Figure 5. Recovery Failure Detector in TrapFuzz.

**State-Consistency Detector.** The State-Consistency Detector verifies architectural state across trap-entry and trap-return boundaries. For each trap level, TrapFuzz derives the expected CSR state from the MTSM, including target-level xEPC/xCAUSE/xTVAL updates, status-field updates such as xPP/xPIE/xIE, and return-state restoration. During execution, the detector collects CSR snapshots and compares them against these expectations. Deviations such as incorrect PC capture, stale privilege bits, malformed cause codes, improper delegation-mask application, or incorrect return-state reconstruction are reported as state-consistency failures.

**State-Consistency Detector.** The State-Consistency Detector verifies architectural state across trap-entry and trap-return boundaries. For each trap level, TrapFuzz derives the expected CSR state from the MTSM, including target-level xEPC/xCAUSE/xTVAL updates, status-field updates such as xPP/xPIE/xIE, and return-state restoration. During execution, the detector collects CSR snapshots and compares them against these expectations. Deviations such as incorrect PC capture, stale privilege bits, malformed cause codes, improper delegation-mask application, or incorrect return-state reconstruction are reported as state-consistency failures.

**Liveness Detector.** The Liveness Detector checks the forward-progress invariant after nested traps. TrapFuzz monitors instruction retirement, trap frequency and depth, and the responsiveness of mret/sret. Typical symptoms include persistent loops inside trap handlers or stalls where return instructions fail to restore the PC or privilege state. To distinguish legitimate long-running handlers from true liveness failures, TrapFuzz uses an adaptive timeout based on retirement increments, privilege transitions, and trap-exit attempts for the executed sequence. If no progress is observed, the execution is reported as a liveness failure.

**Bug Reproduction and De-duplication.** We record executed trap sequences in a timestamped log. When the recovery oracle reports a failure, TrapFuzz replays the logged trap interactions up to the report point, preserving the exact ordering of exceptions, interrupts, privilege transitions, and mret/sret paths to faithfully reconstruct the nested-trap context that triggered the failure for diagnosis and validation. TrapFuzz then performs deduplication inspired by AFL's testcase minimization process [21]: it iteratively removes or

merges traps while checking whether the failure persists, prioritizing entries that induce major MTSM state transitions so that the reproducing sequence remains minimal yet effective. Finally, two reports are treated as duplicates if their minimized sequences share the same structural pattern (identical trap types) and trigger the same failure type.

## 5 Implementation

We implemented TrapFuzz and evaluated it on eight diverse open-source RISC-V CPUs: the six CPUs used in our study, including NanoRV32, CVA6, VexRiscv, Rocket, Xiangshan, and Kronos, and two held-out CPUs, BOOM and srv32. We include the two held-out CPUs to evaluate whether TrapFuzz generalizes beyond the CPUs used to motivate the design and to avoid overfitting the MTSM to study-specific targets. We selected these CPUs because they cover different implementation styles, privilege configurations, and microarchitectural complexities. NanoRV32 is a PicoRV32 [12]-derived fork that adds optional Machine-mode privileged-ISA support, including trap-related CSRs, exception/interrupt handling, and mret logic. Since vanilla PicoRV32 relies on a custom interrupt mechanism rather than the standard privileged trap-entry and trap-return semantics required by TrapFuzz, we use NanoRV32 as the microcontroller-class target and enable the M-mode portion of MTSM for this CPU. CVA6 is a Linux-capable in-order core from OpenHW, VexRiscv is a configurable SpinalHDL-based microcontroller core, Rocket is a mainstream Chisel-based design used in many academic and commercial chips, Xiangshan and Kronos are modern research CPUs with advanced microarchitectural features, BOOM [35] is an open-source Chisel-based out-of-order RISC-V core, and srv32 [17] is a lightweight three-stage RV32IM soft core with trap and interrupt support. Together, they span different HDLs (Verilog, SystemVerilog, SpinalHDL, and Chisel), microarchitectures (microcontroller-class cores, in-order pipelines, configurable pipelines, and out-of-order designs), and deployment contexts, showing that TrapFuzz is platform-independent, HDL-agnostic, and applicable to diverse RISC-V CPUs.

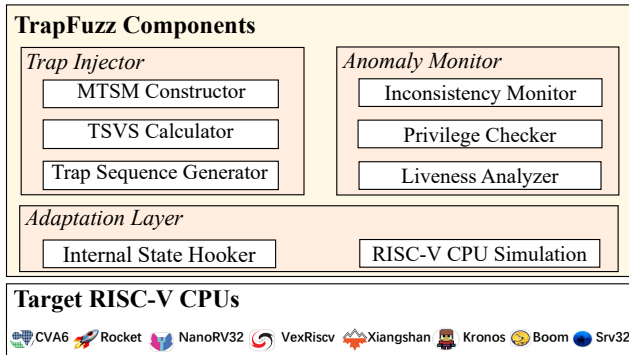
**Table 4.** Per-core interrupt delivery and legality check.

| CPU       | Interrupt delivery                      | Legality check                                     |
|-----------|-----------------------------------------|----------------------------------------------------|
| NanoRV32  | Top-level timer/external IRQs           | M-mode-only MTSM; unsupported paths pruned         |
| CVA6      | Testbench IRQs and timer/PLIC           | Privilege, delegation, enable, and priority guards |
| VexRiscv  | Core-level IRQ inputs                   | Supported trap causes and xRET states              |
| Rocket    | Timer/PLIC via SoC harness              | Delegation, privilege, and xRET guards             |
| Xiangshan | Diffitest/testbench IRQs and timer/PLIC | Pending/enable, priority, and return-state guards  |
| Kronos    | Verilator top-level IRQs                | Unsupported modes and illegal preemptions pruned   |
| BOOM      | Chipyard/Rocket-style timer/PLIC        | Target-specific MTSM instantiation                 |
| srv32     | Top-level IRQ input                     | M-mode-only trap-cause guards                      |

Figure 6 presents the core components of TrapFuzz, which are divided into three main parts. The first part is the Trap Injector, responsible for constructing high-quality multi-trap testcases that model key trap semantics and exercise the multi-trap mechanisms as much as possible. The second part is the Anomaly Monitor, consisting of the Inconsistency Monitor, Privilege Checker, and Liveness Analyzer. These modules observe execution behaviors and identify recovery failures. The third part is the execution adaptation layer, which interacts with the target RISC-V CPUs through simulation and collects internal trap-related states via the Internal State Hooker. Only the third part requires minor modifications when adapting to new CPUs. The rest of the section describes notable implementation details.

**Testcase emission and input delivery.** TrapFuzz materializes each selected *trapSeq* into a runnable RTL test, following conventional RISC-V simulation workflows. The generator emits an assembly/C testcase, trap-handler code, CSR initialization code, and memory/page-table setup, then compiles them into an ELF/binary image and loads it into the target simulation environment. Synchronous exceptions are induced by generated instructions with configured privilege, memory, and page-table states. As summarized in Table 4, asynchronous interrupts are delivered through target-provided simulation interfaces: for core-level testbenches, TrapFuzz drives existing top-level interrupt inputs; for SoC-like testbenches, it programs or stimulates timer/PLIC models. To reach sensitive timing windows, TrapFuzz observes trap-entry, CSR-update, handler, and return phases from execution traces, then mutates interrupt assertion cycles around these phases. Throughout testing, TrapFuzz does not write internal RTL registers or force pipeline state; it only controls architectural test inputs and existing testbench interfaces.

**Adaptation to New RISC-V CPUs:** Adapting TrapFuzz to other RISC-V CPUs requires little effort because its modules are well-encapsulated and loosely coupled. Developers only need to implement two CPU-specific interfaces. (1) The first is the ‘internal state hook’, which exposes the architectural states needed during trap-sequence execution by mapping CPU-specific CSR layouts, trap-entry indicators, and privilege-transition signals to TrapFuzz’s unified state abstraction. Most CPUs already provide trace signals (e.g., Diffitest commit logs and CSR updates in Xiangshan and Rocket-Chip) or debug outputs (e.g., Verilator waveforms and RVFI traces in NanoRV32 and VexRiscv), so adaptation typically requires only a small wrapper. (2) The second is



**Figure 6.** Implementation of TrapFuzz.

the ‘CPU simulation driver’. Since nearly all open-source RISC-V CPUs ship with their own simulation or testbench infrastructure, TrapFuzz can directly reuse these front-ends without modification. As a result, the execution interface requires negligible effort and simply invokes the existing simulation workflow to run generated trap sequences.

## 6 Evaluation

To evaluate the effectiveness of TrapFuzz, we compare it against five state-of-the-art CPU testing tools widely used in prior work: two instruction generators, Cascade and RISCVDV, two robustness CPU testing tools, WhisperFuzz and HWFuzz, and FORCE-RISCV [28], a RISC-V instruction-sequence generator for constrained architectural testing. We ran each testing tool under the same environment setup on a computer equipped with an Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz, 32 GiB of DDR4 memory, and running x86\_64 Ubuntu Linux 20.04. All target RISC-V CPUs were tested using their default configuration parameters. All experiments were repeated 10 times under identical conditions, and the average results are used in this paper. We designed experiments to address the following 4 research questions:

- **RQ1:** Is TrapFuzz effective in detecting recovery failure issues of real-world RISC-V CPUs?
- **RQ2:** Can TrapFuzz cover more trap handling logic within RISC-V CPUs compared with state-of-the-art methods?
- **RQ3:** Do the MTSM and trap-state volatility guided fuzzing effectively improve testing performance?
- **RQ4:** What is the accuracy of MTSM and Detector?

### 6.1 Bug Detection in real-world RISC-V CPUs

We applied TrapFuzz to all eight target RISC-V CPUs to evaluate its effectiveness in detecting recovery-failure bugs. For comparison, we ran five representative baselines under the same setup: Cascade, RISCVDV, WhisperFuzz, HWFuzz, and FORCE-RISCV. Because none provides an effective oracle for recovery failures, we equipped them with our three detectors, yielding *Cascade*<sup>+</sup>, *RISCVDV*<sup>+</sup>, *WhisperFuzz*<sup>+</sup>, *HWFuzz*<sup>+</sup>, and *FORCE-RISCV*<sup>+</sup>. Each experiment ran for 24 hours on each CPU. In total, TrapFuzz uncovered 21 recovery-failure bugs across the eight processors: 3 in NanoRV32, 5 in CVA6, 3 in VexRiscv, 2 in Rocket, 2 in Xiangshan, 1 in Kronos, 3 in BOOM, and 2 in srv32. Details of these new recovery failures are listed in Table 5.

As shown in Table 5, most recovery-failure bugs (12/21) arise from inconsistent CSR or trap-state updates, leading to mismatched *mstatus*, *mepc*, *mcause*, or trap metadata during nested traps. These inconsistencies violate assumptions required for correct trap return and often result in incorrect control flow, corrupted trap-context reconstruction, or silent propagation of erroneous architectural state. Six failures (#3, #7, #10, #15, #18, #21) stem from incorrect privilege transitions, where CPUs resume execution at an unintended

privilege level because of misordered CSR updates, stale privilege state, or outdated delegation information. Such violations weaken privilege isolation and may allow unauthorized execution at elevated privilege. The remaining three failures (#8, #13, #19) are liveness issues, where the CPU fails to make forward progress after nested traps because trap-return conditions are never satisfied or repeated trap replay and pipeline flushes prevent instruction retirement.

**Comparison with Existing Fuzzers:** In our 24-hour experiments, *Cascade*<sup>+</sup> detected 4 recovery-failure bugs (#2, #9, #18, #21), and *RISCVDV*<sup>+</sup> identified 5 bugs (#2, #9, #16, #18, #21). *WhisperFuzz*<sup>+</sup> found 6 bugs (#4, #5, #9, #16, #18, #21), while *HWFuzz*<sup>+</sup> detected 4 bugs (#2, #5, #9, #21). *FORCE-RISCV*<sup>+</sup> also detected 6 bugs (#2, #4, #9, #12, #18, #21). However, no baseline uncovered more than 6 of the 21 recovery-failure bugs, because these tools cannot systematically explore ordered nested-trap interactions. Triggering the missed bugs requires specific combinations of exceptions and interrupts, often involving subtle ordering constraints and multiple trap-entry and trap-return transitions. Existing instruction generators, robustness CPU tests, and FORCE-RISCV-based tests do not model trap-state semantics, delegation-dependent routing, or privilege-flow dependencies, preventing them from reaching these deeper execution paths. In contrast, TrapFuzz constructs a detailed model of trap behavior through MTSM and prioritizes trap sequences using TSVS, enabling it to explore complex multi-trap interactions that other tools fail to cover. This approach allows TrapFuzz to detect all 21 recovery-failure bugs, demonstrating its effectiveness in identifying recovery failures across real-world RISC-V CPUs and answering **RQ1**. Compared with other state-of-the-art CPU testing techniques, TrapFuzz found all the recovery failures that other methods found.

**6.1.1 Case Study.** We present a representative case to illustrate how a recovery failure arises in a real RISC-V CPU and how TrapFuzz identifies it. This case corresponds to failure #1 in Table 5, discovered on NanoRV32. The bug occurs in the trap-return logic during a three-level nested trap sequence, where an incorrect ordering of CSR updates leads to an inconsistent *mstatus.MPIE* state. As a result, the processor resumes execution at an incorrect privilege level after returning from nested traps. The RTL code snippet in Figure 7 describes this recovery failure and its fix.

**Root Cause:** In NanoRV32, when an exception is preempted by a timer interrupt, the core saves part of the CSR state, including *mstatus.MPIE* and *mstatus.MPP*. However, during trap return, the RTL incorrectly restores the entire *mstatus* register in one assignment (line 5). In a three-level nested trap sequence, the inner handler may legitimately update *MPIE*, but this update is later overwritten by the premature full-register restore. As a result, the privilege bits and interrupt-enable flags become inconsistent at *mret*, causing the CPU to resume execution with an invalid privilege

**Table 5.** TrapFuzz discovered all 21 bugs, while other SOTA tools detected no more than 6.

| #  | Platform  | Bug Type      | The Root Cause Analysis                                                                                                                 |
|----|-----------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| 1  | NanoRV32  | Inconsistency | Incorrect synchronization between mstatus.MPIE and mepc during closely timer interrupts, leading to inconsistent states.                |
| 2  | NanoRV32  | Inconsistency | Early overwrite of mcause before nested-trap entry completes, causing the trap stack to record inconsistent nesting depth.              |
| 3  | NanoRV32  | Privilege     | Misordered updates to MPP and interrupt-enable bits result in returning to the wrong privilege level after mret.                        |
| 4  | CVA6      | Inconsistency | Loss of intermediate mtval updates when an interrupt arrives during exception handling, corrupting fault metadata chain.                |
| 5  | CVA6      | Inconsistency | Nested traps trigger a race between CSR writeback and pipeline commit, yielding mismatched mstatus values.                              |
| 6  | CVA6      | Inconsistency | Trap-entry micro-ops are reordered under heavy interrupt load, causing inconsistent snapshots of mcause.                                |
| 7  | CVA6      | Privilege     | Incorrect delegation resolution leads to S-mode exceptions resuming with a privilege level higher than intended privilege.              |
| 8  | CVA6      | Liveness      | Multiple pending traps delay required CSR updates, causing the core to stall before executing mret.                                     |
| 9  | VexRiscv  | Inconsistency | Plugin scheduling reorders trap-entry micro-ops, producing mismatched CSR snapshots across nested traps.                                |
| 10 | VexRiscv  | Privilege     | mret enables interrupts prematurely before restoring privilege level, violating required return semantics.                              |
| 11 | VexRiscv  | Inconsistency | Trap entry overwrites mepc earlier than required, breaking architectural ordering rules during nested traps.                            |
| 12 | Rocket    | Inconsistency | Delayed CSR writeback interacts incorrectly with speculative trap entry, causing inconsistent mstatus values.                           |
| 13 | Rocket    | Liveness      | Nested S-mode exceptions trigger repeated ROB flushes, ultimately resulting in non-forward progress.                                    |
| 14 | Xiangshan | Inconsistency | Out-of-order replay reissues trap-entry micro-ops in non-architectural order, leading to mismatched CSR observations.                   |
| 15 | Xiangshan | Privilege     | Delegated traps use an outdated privilege vector, allowing return to a higher privilege level after nested S-mode traps.                |
| 16 | Kronos    | Inconsistency | Back-to-back external interrupts cause intermediate CSR updates to be dropped, leaving an inconsistent trap context.                    |
| 17 | BOOM      | Inconsistency | Nested interrupts expose a race between trap metadata writeback and pipeline recovery, producing inconsistent mepc/mcause.              |
| 18 | BOOM      | Privilege     | A stale privilege state is reused during delegated trap return, causing mret/sret to resume at an unintended privilege level.           |
| 19 | BOOM      | Liveness      | Repeated trap replay and pipeline flushes prevent the handler from reaching a stable return point, leading to loss of forward progress. |
| 20 | srv32     | Inconsistency | Back-to-back exception and interrupt entries overwrite saved trap context before the handler observes the previous mepc/mcause state.   |
| 21 | srv32     | Privilege     | Interrupt-enable restoration is applied before the saved privilege state is fully restored, causing an incorrect post-mret privilege.   |

**Table 6.** Bugs found by each tool.

| Tool                     | Bug Count | Bug ID#s                  |
|--------------------------|-----------|---------------------------|
| TrapFuzz                 | 21        | #1 - #21                  |
| Cascade <sup>+</sup>     | 4         | #2, #9, #18, #21          |
| RISCV-DV <sup>+</sup>    | 5         | #2, #9, #16, #18, #21     |
| WhisperFuzz <sup>+</sup> | 6         | #4, #5, #9, #16, #18, #21 |
| HWFuzz <sup>+</sup>      | 4         | #2, #5, #9, #21           |
| FORCE-RISCV <sup>+</sup> | 6         | #2, #4, #9, #12, #18, #21 |

```

1  /* Faulty trap return logic in NanoRV32 */
2  always @(posedge clk) begin
3      if (trap_ret) begin
4          // Restore mstatus from saved copy
5          mstatus <= saved_mstatus; // Overwrites MPIE
6          state
7          // Restore only MPP and MPIE bits following
8          nested-trap rules
9          mstatus.MPIE <= saved_mstatus.MPIE;
10         mstatus.MPP <= saved_mstatus.MPP;
11         mepc <= saved_mepc;
12     end
13 end

```

**Figure 7.** A multi-trap RTL bug in NanoRV32.

configuration. This bug is triggered only under the ordered sequence: (*exception* → *timer interrupt* → *nested exception inside the interrupt handler* → *inner-trap MPIE update* → *unconditional full mstatus restore* → *mret*). The key condition is that the nested exception should occur after the initial mstatus save but before the faulty restore in the next trap.

Such a three-level nested window is unlikely to be exercised by random instruction generators or uncoordinated interrupt injectors. Coverage-guided fuzzers such as Cascade and RISCV-DV do not model trap hierarchies or nested-interrupt ordering constraints, so they rarely generate the precise interleavings needed to expose this faulty trap-return path. Fault-injection tools such as WhisperFuzz and HWFuzz may occasionally create cascaded traps, but the resulting

states are nondeterministic and insufficient for systematically exploring ordered multi-trap behaviors. In contrast, TrapFuzz synthesizes nested-trap sequences with the MTSM and prioritizes high-impact transitions through TSVS, allowing it to steer execution toward the timing window where the incorrect mstatus overwrite occurs. Its recovery detectors further examine CSR transitions and privilege flow at each trap level, making inconsistencies in mstatus or mepc immediately visible. As a result, TrapFuzz can reliably trigger and identify this recovery failure.

**Table 7.** Recovery-failure issues reproduced by TrapFuzz.

| NanoRV32 | CVA6  | VexRiscv | Rocket | Xiangshan | Kronos | Total |
|----------|-------|----------|--------|-----------|--------|-------|
| 9/9      | 10/11 | 6/6      | 7/7    | 6/8       | 6/6    | 44/47 |

**Historical Failures Evaluation.** We also evaluated TrapFuzz and other tools against 47 historical recovery-failure issues that we had previously analyzed. Specifically, we configured the six target RISC-V CPUs with their historical versions and ran each tool to test each CPU over 24 hours. TrapFuzz successfully detected 44 of the 47 historical recovery failures, Cascade<sup>+</sup>, RISCV-DV<sup>+</sup>, WhisperFuzz<sup>+</sup>, and HWFuzz<sup>+</sup> only detected 4, 7, 11, and 8 recovery failures, respectively, as shown in Table 7. Compared to these methods, TrapFuzz detects 3x to 10x more historical recovery failures.

Among the three historical recovery failures that TrapFuzz does not reproduce, one in CVA6 and two in Xiangshan rely on proprietary microarchitectural behaviors that are not exposed in their publicly available RTL models. These issues involve timing-dependent interactions (e.g., speculation throttling, replay scheduling, or commit-stage arbitration) that manifest only in specific hardware implementations but are absent in deterministic RTL simulation. Since TrapFuzz operates at the granularity of architecturally visible state and relies on cycle-accurate simulation rather than proprietary hardware features, these microarchitectural conditions cannot be triggered in the current execution environment.

## 6.2 Effectiveness on Trap Handling Coverage

Since higher coverage generally increases the likelihood of finding errors, we also evaluated TrapFuzz’s ability to cover the trap-handling logic of the target RISC-V CPUs. Following prior work [15, 37], we used a recent version of a widely used simulator to collect coverage and then extracted trap-handling coverage by post-processing the full reports. Specifically, we retained only items whose source locations fall within trap-handling code regions, including exception entry, interrupt dispatch, privilege-transition logic, and CSR update paths (e.g., mcause, mstatus, mepc, and return-state handling). Table 8 reports the results. TrapFuzz consistently achieves the highest coverage, improving trap-handling coverage by 22%, 26%, 13%, 18%, and 15% over Cascade, RISCVDV, WhisperFuzz, HWFuzz, and FORCE-RISCV, respectively. These results answer **RQ2**.

**Table 8.** Trap handling coverage on eight RISC-V CPUs in 24 hours. TrapFuzz consistently achieves the highest coverage.

| CPU         | Cascade | RISCV-DV | WhisperFuzz | HWFuzz | FORCE-RISCV | TrapFuzz     |
|-------------|---------|----------|-------------|--------|-------------|--------------|
| NanoRV32    | 5892    | 5617     | 6211        | 6041   | 6193        | <b>7078</b>  |
| CVA6        | 7124    | 6862     | 7578        | 7382   | 7421        | <b>8711</b>  |
| VexRiscv    | 6537    | 6338     | 6975        | 6824   | 6739        | <b>7916</b>  |
| Rocket      | 7689    | 7392     | 8157        | 7885   | 8044        | <b>9462</b>  |
| Xiangshan   | 8921    | 8604     | 9522        | 9272   | 9438        | <b>10673</b> |
| Kronos      | 8210    | 7961     | 8803        | 8504   | 8658        | <b>10035</b> |
| BOOM        | 10198   | 9571     | 11206       | 10639  | 10947       | <b>12471</b> |
| srv32       | 5922    | 5893     | 6519        | 5951   | 6422        | <b>7229</b>  |
| Improvement | ↑22%    | ↑26%     | ↑13%        | ↑18%   | ↑15%        | -            |

The improvement comes from TrapFuzz’s explicit modeling of nested-trap semantics. Instruction generators mainly optimize instruction-level diversity, while fault injectors and FORCE-RISCV-based tests do not systematically explore legal nested-trap interleavings, delegation-dependent routing, or trap-return constraints. In contrast, TrapFuzz uses MTSM to construct legal nested-trap sequences and TSVS to prioritize high-impact trap states, enabling deeper and more diverse coverage of trap-handling paths.

## 6.3 Ablation Study

To evaluate the contributions of the Multi-Trap Sequence Model (MTSM) and the TSVS-guided fuzzing algorithm, we conducted an ablation study comparing three TrapFuzz variants: (1) **TrapFuzz**, the full version with both MTSM and TSVS-guided fuzzing enabled; (2) **TrapFuzz<sub>m</sub>**, a baseline that disables MTSM and instead generates trap injections uniformly at random, without enforcing architectural constraints such as privilege ordering, trap-entry legality, delegation rules, or return-state dependencies; and (3) **TrapFuzz<sub>f</sub>**, a variant that retains MTSM but removes TSVS-guided fuzzing, mutating MTSM-valid trap sequences at random. We measured both the number of recovery failures detected and the amount of trap-handling logic covered by each variant during a 24-hour testing period on all eight RISC-V CPUs.

As shown in Table 9, with the support of both MTSM and TSVS-guided fuzzing, TrapFuzz successfully detects all

**Table 9.** Comparison of TrapFuzz<sup>-m</sup>, TrapFuzz<sup>-f</sup>, and TrapFuzz in terms of bug detection and trap-handling coverage.

| CPU         | Number of Bugs         |                        |          | Trap-Handling Coverage |                        |              |
|-------------|------------------------|------------------------|----------|------------------------|------------------------|--------------|
|             | TrapFuzz <sup>-m</sup> | TrapFuzz <sup>-f</sup> | TrapFuzz | TrapFuzz <sup>-m</sup> | TrapFuzz <sup>-f</sup> | TrapFuzz     |
| NanoRV32    | 1                      | 2                      | 3        | 6320                   | 6554                   | <b>7078</b>  |
| CVA6        | 2                      | 3                      | 5        | 7700                   | 7992                   | <b>8711</b>  |
| VexRiscv    | 1                      | 3                      | 3        | 7132                   | 7399                   | <b>7916</b>  |
| Rocket      | 1                      | 2                      | 2        | 8309                   | 8681                   | <b>9462</b>  |
| Xiangshan   | 0                      | 1                      | 2        | 9703                   | 9882                   | <b>10673</b> |
| Kronos      | 0                      | 0                      | 1        | 8960                   | 9372                   | <b>10035</b> |
| BOOM        | 1                      | 2                      | 3        | 11361                  | 11490                  | <b>12471</b> |
| srv32       | 1                      | 1                      | 2        | 6607                   | 6918                   | <b>7229</b>  |
| Improvement | +200%                  | +50%                   | -        | +11%                   | +8%                    | -            |

21 recovery failures and achieves the highest trap-handling coverage within 24 hours. In contrast, TrapFuzz<sup>-m</sup> detects only 7 recovery failures, while TrapFuzz<sup>-f</sup> detects 14. The remaining failures require precise ordering and dependency relations among diverse traps, which are difficult to exercise without the MTSM or targeted volatility guidance. In terms of coverage, TrapFuzz<sup>-m</sup> covers 66,092 trap-handling branches across eight CPUs, while TrapFuzz<sup>-f</sup> covers 68,288. TrapFuzz covers 73,575 branches, improving coverage by 11% over TrapFuzz<sup>-m</sup> and 8% over TrapFuzz<sup>-f</sup>. These results demonstrate the effectiveness of both components: compared to TrapFuzz<sub>m</sub>, TrapFuzz shows that the MTSM enhances both recovery failures detection count and trap handling coverage. Similarly, compared to TrapFuzz<sub>f</sub>, the TSVS-guided fuzzing strategy effectively improves these two metrics. These statistics provide a comprehensive answer to **RQ3**.

## 6.4 Accuracy of the TrapFuzz

To evaluate the accuracy of TrapFuzz and assess potential **false negatives** and **false positives** that may be introduced by the bug reproduction and deduplication process, we first collected all recovery failures reported by three detectors, along with their corresponding reproduction sequences during the testing phase. We then manually analyzed them to identify true positives and false positives.

**Table 10.** Accuracy of TrapFuzz’s Detector.

|                  | Before Deduplication |           |          | After Deduplication |           |          |
|------------------|----------------------|-----------|----------|---------------------|-----------|----------|
|                  | Consistency          | Privilege | Liveness | Consistency         | Privilege | Liveness |
| Reported Number  | 39                   | 22        | 13       | 12                  | 6         | 3        |
| Unique True Bugs | 12                   | 6         | 3        | 12                  | 6         | 3        |
| False Positive   | 4                    | 1         | 1        | 0                   | 0         | 0        |

Before recovery-failure deduplication, TrapFuzz reported 74 raw anomalies in total, including 39 from the State-Consistency Detector, 22 from the Privilege-Flow Detector, and 13 from the Liveness Detector. Replay-based analysis shows that 68 raw reports correspond to 21 unique true recovery failures, while 6 reports are false positives. They arose from long-latency CSR update sequences in CVA6 and Xiangshan, where delayed mcycle and minstret updates temporarily caused reference-model mismatches, leading to misclassification of incorrect trap recovery. After applying TrapFuzz’s bug reproduction and deduplication process, these false positives were eliminated. In addition, we evaluated the accuracy

of the MTSM by manually validating its transition rules against the RISC-V privileged specification. We did not observe incorrect transition rules in the current MTSM, which is consistent with the absence of false positives after replay and deduplication. But some implementation-specific or context-dependent interactions may still fall outside the architectural rules captured by the model. Nevertheless, the current MTSM is sufficiently accurate to guide fuzzing and support reliable recovery-failure detection, as evidenced by TrapFuzz reproducing 44/47 historical recovery-failure issues in Section 6.1. Overall, TrapFuzz achieves satisfactory accuracy, which adequately answers RQ4.

## 7 Discussion

**More bug types support.** TrapFuzz currently targets recovery failures in multi-trap mechanisms by exercising nested exception-interrupt behaviors and checking architectural-state restoration after trap returns. Applied to eight widely used RISC-V CPUs, it uncovered 21 previously unknown recovery failures. Other bug classes remain outside its scope, including microarchitectural hazard bugs [25, 32] and speculation bugs [29]. Representative microarchitectural hazards include pipeline flush/squash ordering errors, CSR bypass hazards, and delayed-retirement inconsistencies, which may silently corrupt state when traps overlap with specific pipeline stages or CSR updates race with trap entry. TrapFuzz could be extended with lightweight microarchitectural monitors, such as reorder-buffer activity or CSR bypass timing, as additional feedback. However, these events are not architecturally visible and vary across implementations, making portable and accurate detection challenging. Scalable support for such bugs remains an important future direction.

**Support for proprietary extensions and vendor specific trap logic.** Some RISC-V implementations include custom interrupt controllers, platform-level exception logic, or vendor specific CSR extensions. Since TrapFuzz currently focuses on the standardized privileged architecture, such features may fall outside its modeling scope and reduce its ability to expose implementation-specific recovery failures. Supporting these extensions would require augmenting the MTSM with custom trap-entry rules, additional CSR semantics, and platform-specific interrupt-routing behavior. However, doing so demands fine-grained design knowledge and careful instrumentation across heterogeneous implementations, increasing both complexity and overhead. Maintaining generality while efficiently modeling vendor-specific trap logic remains an important direction for future work.

## 8 Related Work

**CPU Testing and Fuzzing.** Recent CPU testing and fuzzing techniques mainly target instruction execution, architectural-state consistency, and microarchitectural correctness. Differential and coverage-guided fuzzers such as DifuzzRTL [14],

RFUZZ [18], TheHuzz [15], and WhisperFuzz [1] expose functional, coverage, or timing inconsistencies in RTL implementations. Program-generation tools such as RISC-V-DV [3], FORCE-RISC-V [28], Cascade [37], INSTILLER [46], and MorFuzz [44] improve exploration through randomized generation, constrained testing, coverage feedback, or instruction morphing. These techniques effectively find instruction-level bugs, pipeline hazards, memory-ordering issues, timing vulnerabilities, and CSR inconsistencies. However, they mainly optimize instruction streams, microarchitectural coverage, or program diversity, without modeling ordered nested-trap recovery chains involving trap entry, delegation-dependent routing, interrupt preemption inside handlers, and mret/sret return-state restoration. TrapFuzz addresses this gap by constructing an MTSM for legal nested-trap generation and an MTSM-derived oracle for CSR consistency, privilege-flow legality, and forward progress.

**Formal and Semi-formal Verification.** Formal and semi-formal hardware-verification approaches can provide strong correctness guarantees for specified properties. Tools and frameworks such as riscv-formal [39], Kami [4], and UCLID5 [31] support formal reasoning about processor or hardware behaviors. Prior RISC-V verification efforts have used such tools to check instruction-level ISA semantics, bounded microarchitectural properties, memory-protection logic, or selected privilege-related properties [2]. In principle, formal methods can also verify nested-trap recovery correctness. However, doing so requires complete properties for trap-entry CSR effects, delegation decisions, interrupt-priority rules, nested preemption conditions, and mret/sret return-state restoration, together with implementation-specific observation interfaces. Writing and maintaining such multi-trap recovery properties across heterogeneous RISC-V CPUs requires substantial manual effort. TrapFuzz is complementary to formal approaches: instead of proving all recovery properties statically, it dynamically synthesizes architecturally legal nested-trap executions and checks MTSM-derived recovery invariants during RTL simulation.

## 9 Conclusion

In this paper, we present TrapFuzz, a framework for detecting recovery-failure issues in RISC-V multi-trap mechanisms. TrapFuzz models nested trap rules and uses state-guided exploration with a recovery oracle to expose recovery failures. We implemented and evaluated TrapFuzz on eight RISC-V CPUs, and it successfully found 21 new recovery failures.

## Acknowledgments

This research is sponsored by the NSFC Program (No.62525207, U2441238).

## References

- [1] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5377–5394.
- [2] Kevin Cheang, Cameron Rasmussen, Dayeol Lee, David W. Kohlbrenner, Krste Asanović, and Sanjit A. Seshia. 2022. Verifying RISC-V Physical Memory Protection. arXiv:2211.02179 [cs.CR]
- [3] CHIPS Alliance. 2026. RISC-V-DV: Random Instruction Generator for RISC-V Processor Verification. <https://github.com/chipsalliance/riscv-dv>. Accessed: August 20, 2026.
- [4] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 30. doi:10.1145/3110268
- [5] Matteo Colella. 2023. *Unveiling Security Vulnerabilities of RISC-V Microprocessors*. Master's thesis. Politecnico di Milano.
- [6] Enfang Cui, Tianzheng Li, and Qian Wei. 2023. RISC-V instruction set architecture extensions: A survey. *IEEE Access* 11 (2023), 24696–24711.
- [7] elvisfox. 2026. NanoRV32: A small RV32I[MC] core capable of running RTOS. <https://github.com/elvisfox/nanorv32>. Accessed: August 18, 2026.
- [8] Rocket Chip Generator. 2025. CHIPS Alliance. <https://github.com/chipsalliance/rocket-chip>. Accessed at October 29, 2025.
- [9] OpenHW Group. 2025. CVA6 RISC-V CPU. <https://github.com/openhwgroup/cva6/>. Accessed at October 29, 2025.
- [10] OpenXiangShan Group. 2025. Open-source high-performance RISC-V processor. <https://github.com/OpenXiangShan/XiangShan>. Accessed at October 29, 2025.
- [11] SpinalHDL Group. 2025. A FPGA friendly 32 bit RISC-V CPU implementation. <https://github.com/SpinalHDL/VexRiscv>. Accessed at October 29, 2025.
- [12] YosysHQ Group. 2025. PicoRV32 - A Size-Optimized RISC-V CPU. <https://github.com/YosysHQ/picorv32>. Accessed at October 29, 2025.
- [13] Qusay F Hassan and Assim Sagahyroon. 2025. RISC-V: A Comprehensive Overview of an Emerging ISA for the AI-IoT Era. *Advances in the Internet of Things* (2025), 244–284.
- [14] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1286–1303.
- [15] 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 *31st USENIX Security Symposium (USENIX Security 22)*. 3219–3236.
- [16] Mark Kuhne, Stavros Volos, and Shweta Shinde. 2025. Dorami: Privilege Separating Security Monitor on RISC-VTEEs. In *34th USENIX Security Symposium (USENIX Security 25)*. 1149–1166.
- [17] Kuoping Hsu. 2026. SRV32: Simple 3-stage pipeline RISC-V processor. <https://github.com/kuopinghsu/srv32>. Accessed: August 20, 2026.
- [18] Kevin Laeuffer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.
- [19] Zhiyu Li, Yuhao Huang, Longfeng Tian, Ruimin Zhu, Shanlin Xiao, and Zhiyi Yu. 2021. A low-power asynchronous RISC-V processor with propagated timing constraints method. *IEEE transactions on circuits and systems II: Express briefs* 68, 9 (2021), 3153–3157.
- [20] Per Lindgren, Pawel Dzialo, Henri Lunnikivi, and Johan Ericsson. 2023. ENEST-Efficient Interrupt Nesting for RISC-V based CPUs. In *2023 IEEE 2nd Industrial Electronics Society Annual On-Line Conference (ONCON)*. IEEE, 1–7.
- [21] American Fuzzy Lop. 2025. Performance Tips - AFL (american fuzzy lop) - Keep your test cases small. <https://afl-1.readthedocs.io/en/latest/tips.html>. Accessed at December 27, 2025.
- [22] Tao Lu. 2021. A Survey on RISC-V Security: Hardware and Architecture. *arXiv preprint arXiv:2107.04175* (2021).
- [23] Benjamin W Mezger, Douglas A Santos, Luigi Dilillo, and Douglas R Melo. 2023. Hardening a real-time operating system for a dependable risc-v system-on-chip. In *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 1–6.
- [24] Benjamin W Mezger, Douglas A Santos, Luigi Dilillo, Cesar A Zferino, and Douglas R Melo. 2022. A survey of the RISC-V architecture software support. *IEEE Access* 10 (2022), 51394–51411.
- [25] Hoda Naghibijouybari, Esmail Mohammadian Koruyeh, and Nael Abu-Ghazaleh. 2022. Microarchitectural attacks in heterogeneous systems: A survey. *Comput. Surveys* 55, 7 (2022), 1–40.
- [26] Yari Nowicki. 2023. Timing and Design Optimization of RISC-V on Flexible Technologies. Master's thesis, UHasselt. <http://hdl.handle.net/1942/41336>
- [27] Willian Analdo Nunes, Marcos Luiggi Lemos Sartori, Matheus Trevisan Moreira, Fernando Gehm Moraes, and Ney Laert Vilar Calazans. 2023. Validating an automated asynchronous synthesis environment with a challenging design: Risc-v. In *2023 36th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 1–6.
- [28] OpenHW Group. 2026. FORCE-RISCV: Instruction Sequence Generator for the RISC-V Instruction Set Architecture. <https://github.com/openhwgroup/force-riscv>. Accessed: August 20, 2026.
- [29] Donn B Parker. 2007. The dark side of computing: SRI international and the study of computer crime. *IEEE Annals of the History of Computing* 29, 1 (2007), 3–15.
- [30] Phantom1003. 2022. Incorrect tval for ecall/ebreak. <https://github.com/openhwgroup/cva6/issues/898>. Accessed at December 27, 2025.
- [31] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeuffer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. 2022. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. In *34th International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 13371)*. Springer, 538–551.
- [32] Joseph Ravichandran. 2023. *Discovering Novel Microarchitectural Security Vulnerabilities in Modern Processors*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [33] Isaac Richter, Jie Zhou, and John Criswell. 2024. DeTRAP: RISC-V return address protection with debug triggers. In *2024 IEEE Secure Development Conference (SecDev)*. IEEE, 166–177.
- [34] RISC-V International. 2021. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Document Version 20211203, Version 1.12. <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>
- [35] RISC-V-BOOM Project. 2026. BOOM: The Berkeley Out-of-Order Machine. <https://github.com/riscv-boom/riscv-boom>. Accessed: August 20, 2026.
- [36] RK Risikesh, Sharad Sinha, and Nanditha Rao. 2021. Variable bit-precision vector extension for RISC-V based Processors. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*. IEEE, 114–121.
- [37] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU fuzzing via intricate program generation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5341–5358.
- [38] SonalPinto. 2025. Kronos is a 3-stage in-order RISC-V core towards FPGA implementations. <https://github.com/SonalPinto/kronos>. Accessed at October 29, 2025.

- [39] Symbiotic EDA. 2026. riscv-formal: RISC-V Formal Verification Framework. <https://github.com/SymbioticEDA/riscv-formal>. Accessed: August 20, 2026.
- [40] Shibo Tang, Jiacheng Zhu, Yifei Gao, Jing Zhou, Dejun Mu, and Wei Hu. 2023. Verifying RISC-V Privilege Transition Integrity Through Symbolic Execution. In *2023 IEEE 32nd Asian Test Symposium (ATS)*. IEEE, 1–6.
- [41] Chuanning Wang, Chao Fang, Xiao Wu, Zhongfeng Wang, and Jun Lin. 2024. Speed: A Scalable RISC-V Vector Processor Enabling Efficient Multiprecision DNN Inference. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2024).
- [42] Wonicon. 2025. Multicore rocket failed to run bbl or pk due to unhandlable trap 3. <https://github.com/chipsalliance/rocket-chip/issues/549>. Accessed at December 27, 2025.
- [43] WorldofJARcraft. 2025. mstatus.FS does not indicate dirty correctly. <https://github.com/openhwgroup/cva6/issues/2934>. Accessed at December 27, 2025.
- [44] 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 *32nd USENIX Security Symposium (USENIX Security 23)*. 1307–1324.
- [45] Andrés Zambrano Bustos. 2024. *ReCon Implementation: A Load Pair-Tracking Mechanism to Lift Security Protections on a RISC-V Processor*. Master's thesis. NTNU.
- [46] Gen Zhang, Pengfei Wang, Tai Yue, Danjun Liu, Yubei Guo, and Kai Lu. 2024. INSTILLER: Toward efficient and realistic RTL fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 7 (2024), 2177–2190.
- [47] Yang Zhong, Haoran Wu, Yungang Bao, and Kan Shi. 2025. HWFuzz: An FPGA-Accelerated Fuzzing Framework for Efficient RISC-V Verification. RISC-V Summit Europe 2025 Poster/Abstract. Accessed: August 20, 2026. <https://riscv-europe.org/summit/2025/media/proceedings/2025-05-14-RISC-V-Summit-Europe-P2.2.04-ZHONG-abstract.pdf>
- [48] Yuzhi Zhou, Xi Jin, Tian Xiang, and Daolu Zha. 2020. Enhancing energy efficiency of RISC-V processor-based embedded graphics systems through frame buffer compression. *Microprocessors and Microsystems* 77 (2020), 103140.