

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

DRVFuzz: Data-Sensitive RISC-V CPU Fuzzing

Zehong Yu, *Tsinghua University;* Yuanliang Chen, *Renmin University of China;*
Zhen Yan, Xudong Zhang, Zhensheng Xian, and Yu Jiang, *Tsinghua University*

https://www.usenix.org/conference/usenixsecurity26/presentation/yu-zehong

This paper is included in the Proceedings of the
35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the
35th USENIX Security Symposium
is sponsored by



جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

DRVFuzz: Data-Sensitive RISC-V CPU Fuzzing

Zehong Yu
Tsinghua University

Yuanliang Chen*
Renmin University of China

Zhen Yan
Tsinghua University

Xudong Zhang
Tsinghua University

Zhensheng Xian
Tsinghua University

Yu Jiang*
Tsinghua University

Abstract

The rapid adoption of RISC-V across modern computing systems has made the security integrity of its implementations a paramount concern. Logic bugs in RISC-V cores can lead to critical failures, such as faulty privilege transitions and architectural state corruption. While hardware fuzzing has emerged as a powerful technique for automated bug discovery, existing frameworks remain largely data-agnostic. By prioritizing instruction sequence diversity while treating operands as incidental random values, these tools often fail to trigger guarded microarchitectural states that manifest only under precise, data-dependent conditions.

In this work, we present *DRVFuzz*, a data-sensitive fuzzing framework designed to expose hardware vulnerabilities by explicitly modeling and navigating the data-sensitive semantics. First, *DRVFuzz* introduces a sensitive data model (SDModel) that hierarchically codifies ISA semantics to synthesize tailored operands, including boundary values and exception triggers. Second, to effectively explore data-dependent paths, *DRVFuzz* employs transition-guided fuzzing, prioritizing testcases that trigger previously unseen state transitions as labeled by the SDModel. We evaluated *DRVFuzz* on six real-world RISC-V CPUs with varying microarchitectural complexity, uncovering 22 previously unknown bugs (19 new CVEs).

1 Introduction

The RISC-V architecture has gained significant traction across the global semiconductor industry, emerging as a prominent open-standard Instruction Set Architecture (ISA) for applications ranging from embedded IoT sensors to high-performance accelerators [13, 17, 29]. As RISC-V becomes a viable alternative to proprietary architectures, ensuring the security integrity of its implementations has become a paramount concern [2, 20]. Despite rigorous design efforts, a substantial number of logic bugs and security vulnerabilities have been uncovered in widely-used RISC-V cores [26, 31, 47].

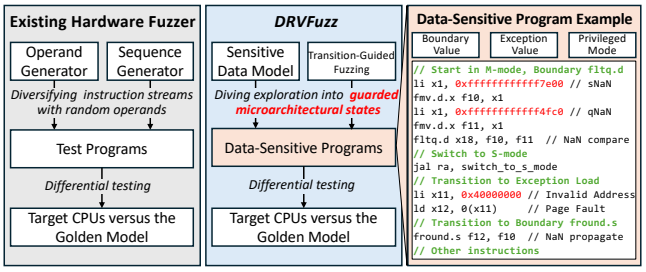


Figure 1: *DRVFuzz* leverages the sensitive data model and constructs transition-guided instruction sequences to explore guarded microarchitectural states. In contrast, existing hardware fuzzers overlook the importance of operand data.

These flaws, such as floating-point errors or faulty privilege transitions, often remain dormant and manifest only under rare conditions that are difficult to trigger [16, 26, 28].

Hardware fuzzing [8, 9, 27, 31, 34, 43, 47, 51] has emerged as an effective technique for the automated discovery of hardware vulnerabilities by generating instruction sequences and comparing Design Under Test (DUT) execution against a golden model, *e.g.*, Spike [41]. Existing tools primarily focus on sequence-level diversity and coverage-guided mutations. For example, Cascade [47], a state-of-the-art fuzzer, constructs intricate instruction streams to stress inter-instruction interactions and control-flow transitions. Similarly, DiveFuzz [23] enhances feedback granularity by incentivizing diverse writeback behaviors to improve observational variety.

While these approaches have shown effectiveness in uncovering certain bugs, they still lack *data-sensitive guidance*, as shown in Figure 1. That is, they treat operands as incidental and randomly generated values, leading to instruction sequences that frequently exercise common-case logic rather than data-dependent microarchitectrual states. The RISC-V ISA is inherently replete with data-sensitive semantics, where the activation of critical microarchitectural paths is tied to specific operand values [5, 35, 40, 54]. For instance, the data-sensitive program in Figure 1 illustrates that precise

*Yuanliang Chen and Yu Jiang are the corresponding authors.

bit-patterns can trigger guarded behaviors, including NaN propagation and page-fault exceptions. These paths remain dormant unless triggered by precise bit-patterns and configurations.

In this paper, we introduce *DRVFuzz*, a data-sensitive fuzzing framework for RISC-V CPUs. Our key insight is that certain latent hardware failures manifest only under fragile execution conditions, requiring specific operand values to trigger. Accordingly, *DRVFuzz* explicitly synthesizes tailored operands, *e.g.,* boundary values and exception triggers, to drive exploration into guarded microarchitectural states.

To achieve this, however, we have to deal with the following challenges: (1) **The first challenge is that modeling instruction data-sensitive semantic is non-trivial.** RISC-V instructions span heterogeneous semantics, from integer and floating-point operations to memory and I/O interactions, each with distinct input domains, *e.g.*, subnormals and alignment boundaries. Systematically identifying and modeling these diverse data-sensitive semantics across a vast instruction set is difficult. (2) **The second challenge is that exploring fragile execution conditions is difficult.** Even when sensitive operand values are identified, the search space remains vast because reaching these conditions often depends on how data interacts with instruction sequences and their data-flow constraints. Random generation strategies are therefore rarely to satisfy the state-dependent requirements necessary to reach and sustain fragile architectural behaviors.

To address the first challenge, *DRVFuzz* introduces a SDModel that automatically synthesizes data-sensitive operands, spanning arithmetic corner cases (*e.g.*, NaNs) and architectural exception triggers (*e.g.*, misalignments). Specifically, SDModel hierarchically codifies data-sensitive ISA semantics into class-specific templates derived from opcode metadata and core parameters, while preserving instruction-level precision via lightweight refinements. To address the second challenge, *DRVFuzz* employs transition-guided fuzzing to effectively steer execution into data-sensitive paths. *DRVFuzz* leverages SDModel to monitor the execution states and extract state transitions of each retired instruction. *DRVFuzz* then prioritizes testcases that introduce previously unseen transitions as guidance for continuously exploring guided microarchitectural behaviors. Finally, *DRVFuzz* records essential architectural context, including registers, memory, and exceptions, to enable precise vulnerability diagnosis.

We implemented *DRVFuzz* and evaluated it on six real-world RISC-V CPUs spanning diverse design complexities and ISA extensions: BOOM-V3 [6], BOOM-V4 [6], Rocket [19], CVA6 [21], Kronos [48], and Srv32 [33]. Our evaluation demonstrates that, compared to state-of-the-art fuzzers, Cascade [47] and DiveFuzz [23], *DRVFuzz* triggers more state transitions and outperforms them in microarchitectural coverage metrics. In total, *DRVFuzz* successfully uncovered 22 previously unknown bugs (19 new CVEs), including 3 in BOOM-V3, 2 in BOOM-V4, 3 in Rocket, 5 in CVA6, 3 in Kronos, and 6 in Srv32. These bugs range from FPU logic errors to speculative corruption, and can lead to serious consequences, such as denial-of-service and control-flow hijacking. In summary, we make three key contributions:

- We propose the SDModel that hierarchically codifies ISA semantics to synthesize data-sensitive operands, capturing boundary values and exception triggers.
- We introduce transition-guided fuzzing that prioritizes previously unseen transitions to continuously exercise data-sensitive microarchitectural states.
- We implement and evaluate *DRVFuzz* on six widely used RISC-V CPUs. We will open-source it[1] for practical usage. Currently, it has already detected 22 new bugs.

2 Background

2.1 RISC-V

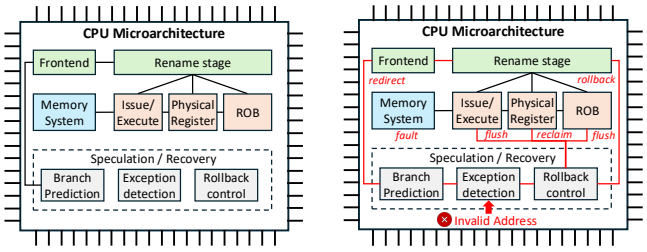
RISC-V Architecture. RISC-V is an open and modular ISA consisting of unprivileged and privileged architecture specifications. Implementations may target 32-bit or 64-bit variants and selectively include standard extensions, *e.g.*, M (integer multiplication and division), F (floating-point), and A (atomic operations). Beyond the unprivileged ISA, the privileged specification defines the execution modes, *i.e.*, machine mode (M), supervisor mode (S) and user mode (U), which form a hierarchical privilege model. M-mode provides the highest level of privilege and is responsible for low-level system initialization, trap handling, and direct control over hardware resources. S-mode supports operating-system functionality, including virtual memory management and access control, while U-mode executes application code under restricted privileges. The hardware's security and execution logic are orchestrated by a set of Control and Status Registers (CSRs), such as `mstatus` and `satp`, which reflect and configure foundational states including privilege transitions and trap delegation policies. Furthermore, to manage the architectural complexity, RISC-V UnifiedDB [15] provides an official and machine-readable repository that precisely codifies instruction encodings, operand constraints, and CSR semantics, maintaining strict alignment with the RISC-V specifications.

Data-Sensitive Semantics in RISC-V. Many RISC-V behaviors are inherently data-sensitive, where architectural control flow and state evolution are gated by specific operand values rather than opcodes alone [5, 54]. We refer to these as semantic data, where bit-patterns in addresses, immediates, or floating-point operands serve as implicit control signals. For instance, while standard floating-point addition proceeds through a hardware-optimized fast-path, specific operand patterns such as subnormal numbers or NaNs may bypass this logic and trigger a slow-path, often involving a pipeline flush

[1]The artifact of *DRVFuzz* is available at Zenodo [55]. We also release it at https://github.com/YzhDDDing/DRVFuzz.

to invoke microcoded handling routines [3, 38, 40]. In these scenarios, the data is not merely a calculation input but a determinant for the underlying hardware's execution mode.

Fragility of Data-Sensitive Paths. Compared to common-case execution, data-sensitive scenarios often activate a broader set of interacting CPU components. While typical instruction execution primarily exercises the steady-state pipeline, data-sensitive conditions simultaneously engage front-end control logic, speculative execution structures, and exception handling mechanisms [4, 25, 30]. For example, in out-of-order designs such as BOOM [6], a speculative long-latency instruction, *e.g.*, `fdiv` and `fsqrt`, can occupy entries in the reorder buffer (ROB) and tie up physical registers in the rename map for an extended period [28]. If an exceptional event arises during this time, for instance, a memory access to an unmapped address as shown in Figure 2, the processor's control logic must coordinate a multi-faceted recovery: flushing any incorrect speculative state, rolling back the register renaming to a known-good state, and canceling all in-flight instructions beyond the faulting point [11, 26]. Such intense cross-component interaction significantly increases the probability of subtle architectural inconsistencies, making data-sensitive paths structurally more error-prone than common-case execution flows.



(a) Common-case execution (b) Data-sensitive execution

Figure 2: Data-sensitive execution activates broader and more tightly coupled CPU modules than common-case execution. An invalid memory address triggers exception handling, speculative state flush, ROB cancellation, and rename rollback, exposing dense cross-component interactions.

2.2 Hardware Fuzzing

Hardware fuzzing has emerged as a key technique for the automated discovery of logic bugs and security vulnerabilities in Register Transfer Level (RTL) designs. Unlike formal verification, which can suffer from state-space explosion and depends on carefully crafted properties and assumptions, hardware fuzzing provides a scalable approach to stress-test complex microarchitectures by executing large volumes of generated instruction sequences [44].

A common workflow for hardware fuzzing is differential testing. In this setup, the same instruction sequences are executed on both the Design Under Test (DUT) and the software-based Instruction Set Simulator (ISS) served as a reference model, for example, Spike in RISC-V [41]. Any mismatch in the architectural state, such as register updates, memory writes, and trap causes, between the DUT and the ISS is flagged as a potential vulnerability or logic bug. To improve testing efficiency, existing hardware fuzzers typically prioritize instruction sequence diversity and coverage-guided mutations. Concretely, they mutate instructions under ISA constraints and leverage feedback signals, such as RTL coverage, to guide exploration toward newly covered behaviors. However, as aforementioned, many critical behaviors are guarded by specific operand values rather than opcode patterns alone. As a result, instruction diversity and generic feedback may still frequently exercise common-case logic, leaving data-dependent microarchitecture behaviors untested.

3 Motivation Example

Modern processors improve performance by incorporating a wide range of microarchitectural mechanisms, including deep pipelines, speculative execution, and sophisticated recovery logic [14, 24, 32, 36, 45]. While these components are essential for high throughput, their interactions are often data-dependent, significantly increasing design complexity and making it more challenging to preserve security invariants across the microarchitecture [1, 39, 49]. Precise exception handling provides a representative example. Although exceptions are architecturally defined as occurring at a precise instruction boundary, their implementation inevitably involves coordination among multiple components, such as the load-store unit (LSU), ROB, and pipeline recovery [11, 25, 45]. When these components are not carefully synchronized, faulting instructions may interact with transient or partially validated microarchitectural states, causing behaviors that violate architectural expectations such as precise exception ordering, rollback atomicity, or fault isolation [10].

A typical example in RISC-V designs is a BOOM-V4 precise-exception violation uncovered by *DRVFuzz*. As illustrated in Figure 3, the root cause lies in the improper coordination between the LSU alignment validation and ROB retirement logic. Specifically, the instruction sequence first loads `0x8FFFFFCC60` into `a8`, followed by two `sd` instructions using the same misaligned offset `+6`. Since `sd` requires an 8-byte aligned effective address, both Store A and Store B should raise a store address misaligned exception (`mcause`=6) and must not become visible under precise-exception semantics. However, BOOM-V4 exhibits inconsistent ordering between exception and retirement. For Store A, the misaligned exception is available by the time the store reaches the ROB head, so retirement is correctly suppressed and no commit trace is emitted. In contrast, for Store B, the alignment check is resolved late in the execution pipeline. Due to an implementation flaw, the ROB commit logic does not wait for this

late-arriving exception signal and erroneously marks the faulting store as "commit valid". The core subsequently registers the exception, flushes the pipeline, and vectors to the trap handler, but the false commit has already been logged. Consequently, the faulting store is incorrectly marked as committed before the trap is delivered, making the misaligned store appear architecturally visible even though it should have been suppressed under precise-exception semantics.

```
li a8, 0x8FFFFFCC60
# 1. Exception arrives early
# 2. Commit suppressed.
sd a10, 6(a8) # Misaligned Store A
# 1. Store reaches ROB head
# 2. Assert Commit Valid and emit Trace
# 3. Exception arrives and flush pipeline
sd a12, 6(a8) # Misaligned Store B
```

Figure 3: A precise-exception violation in BOOM-V4. Store A is handled correctly because the misaligned exception is reflected in the ROB before retirement. Store B triggers a race condition: the alignment verification is not synchronized with the commit decision. The ROB erroneously commits the instruction before the exception signal arrives.

RISC-V is now deployed from tiny IoT chips to high-performance CPUs, making the security of its implementations a foundational concern. Yet, given the growing complexity, latent bugs within RISC-V CPUs are inevitable, and even a single flaw can translate into real-world security failures, including secret leakage and privilege escalation [26, 28]. We can draw three important lessons from this case: (1) **Many security-relevant failures are fundamentally data-sensitive.** They are activated only when operand values steer the core into guarded and rarely exercised modes, *e.g.*, boundary values and exception triggers, which data-agnostic fuzzers often fail to explore. To address this, we propose *DRVFuzz* to explicitly model data-sensitive semantics and synthesize operands that drive exploration into these guarded microarchitectural states. (2) **Modeling such data sensitivity is inherently non-trivial.** The difficulty stems from the highly heterogeneous semantics of RISC-V instructions, which range from integer arithmetic and floating-point operations to memory and I/O interactions. Each category possesses distinct data domains, such as subnormal, or specific alignment boundary, varying on different execution contexts, such as privileged modes and rounding modes. To address this issue, *DRVFuzz* introduces a SDModel that hierarchically codifies ISA semantics into class-specific templates to automatically synthesize data-sensitive operands across diverse instruction classes. (3) **Sensitive data alone is insufficient as many bugs surface only in deep and fragile sequences.** In this case, triggering the BOOM-V4 precise-exception violation requires a specific combination of sensitive conditions, including a misaligned effective address and two misaligned stores that expose different timing relationships between LSU alignment validation and ROB retirement. Such combinations are inherently difficult to reach and sustain, because the underlying search space over sensitive conditions and valid instruction sequences is vast. To handle this issue, *DRVFuzz* employs transition-guided fuzzing to continuously steer execution into diverse data-sensitive paths, by prioritizing testcases that trigger previously unseen state transitions.

4 *DRVFuzz* Design

Design goal: A practical data-sensitive hardware fuzzing framework should have the following properties.

- *General: DRVFuzz* is designed to uncover bugs in a wide range of real-world RISC-V CPUs, spanning in-order microcontrollers, *e.g.*, Kronos [48] and Srv32 [33], to superscalar out-of-order processors with rich ISA extensions, *e.g.* BOOM [6]. The framework is largely independent of specific microarchitectural optimizations and can be applied across diverse core designs with minor adjustments.
- *Efficient: DRVFuzz* is able to constantly dive exploration into rarely exercised data-sensitive paths, and effectively detect bugs in real-world RISC-V CPUs within 24 hours.
- *Accurate: DRVFuzz* is designed to have satisfying precision and recall to avoid reporting false positives.

4.1 *DRVFuzz* Workflow

Figure 4 illustrates the workflow of *DRVFuzz*, which consists of two main phases. The first phase focuses on the construction of *Sensitive Data Model* (SDModel). Given the RISC-V UnifiedDB and a target CPU configuration, *DRVFuzz* parses opcode-level metadata and core-specific parameters. Based on this information, *DRVFuzz* derives a predicate set that captures boundary conditions and exception triggers, and hierarchically constructs data-synthesis templates. The resulting SDModel serves as an executable specification that supports both operand synthesis and execution state labeling.

The second phase is *Transition-Guided Fuzzing*. After SDModel construction, *DRVFuzz* enters a fuzzing loop that repeatedly executes the following steps: (1) Based on the SDModel, *DRVFuzz* selects and mutates a seed from the seed pool by selecting the applicable predicates and synthesizing sensitive operands via corresponding data-synthesis templates. (2) *DRVFuzz* executes the testcase on both the DUT and the ISS, collecting a compact execution trace that records opcodes, operands, relevant architectural context, and results at retirement. (3) *DRVFuzz* compares results between the DUT and the ISS. If a mismatch is detected, the testcase is reported as a bug candidate and preserved for reproduction. (4) If no mismatch is observed, *DRVFuzz* reuses SDModel to label the execution state of each instruction and extracts state transitions from the trace. (5) *DRVFuzz* filters extracted transitions

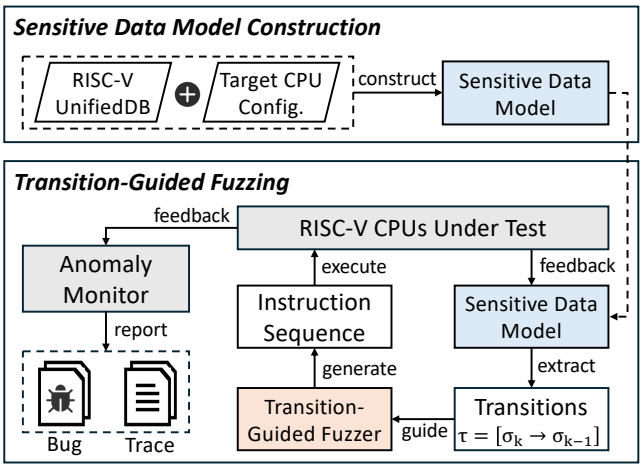


Figure 4: Workflow of *DRVFuzz*. It consists of two main phases: (1) Sensitive Data Model Construction for deriving data-sensitive predicates and operand-synthesis templates, and (2) Transition-Guided Fuzzing for generating data-sensitive instruction sequences and prioritizing previously unexplored execution-state transitions.

against the visited set. If the testcase contains any previously unseen transitions, it is marked as an *interesting seed* and added to the seed pool, prioritized for subsequent mutations. *DRVFuzz* then proceeds to the next iteration (from step 1 to step 5) until the testing process terminates.

4.2 SDModel Construction

RISC-V instructions expose a wide range of data-sensitive behaviors, whose activation depends on specific operand values and architectural context. To model these behaviors in a systematic and scalable manner, *DRVFuzz* introduces a SDModel. As illustrated in Figure 5, SDModel parses RISC-V UnifiedDB with target CPU configuration, and then hierarchically constructs the predicate set and data-sensitive synthesis templates that capture both boundary conditions and exception triggers across heterogeneous RISC-V instructions.

Preparation. *DRVFuzz* first parses two complementary sources, RISC-V UnifiedDB and target CPU configuration, before constructing SDModel. UnifiedDB, maintained by the RISC-V community, provides opcode-level metadata, including instruction classes, operand types and bit-widths, encoding fields, and ISA legality constraints. These metadata define the instruction semantics that SDModel needs to model and directly drive our class-specific template construction. Besides, SDModel must also respect core-specific applicability. Thus, *DRVFuzz* parses the target CPU configuration that specifies the target core's XLEN (RV32 or RV64), the supported ISA extensions (such as M, A, F, D, C, and Zicsr), whether multiple privilege levels are enabled (M/S/U), and the valid physical memory ranges that can be safely accessed by generated programs. *DRVFuzz* uses this configuration to prune irrelevant templates for unsupported extensions, and to parameterize memory-related and privilege-related sensitive data synthesis. Together, UnifiedDB ensures ISA-level completeness while CPU configuration anchors SDModel to supported features and executable environments.

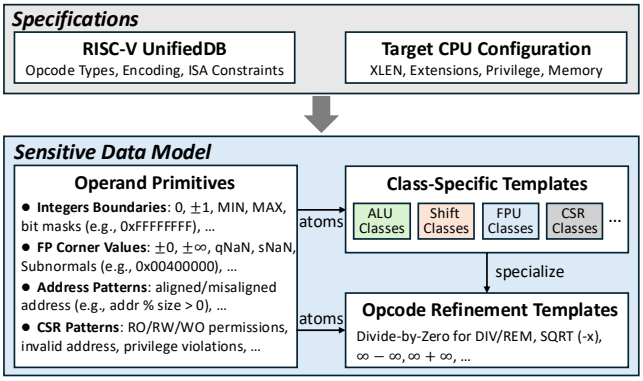


Figure 5: Overview of the Sensitive Data Model. It parses the RISC-V UnifiedDB and target CPU configurations to derive architectural constraints, and then hierarchically constructs data-synthesis templates.

SDModel Abstraction. SDModel treats data sensitivity as a set of predicates $\mathcal{P}$ over instructions, and uses these predicates to derive sensitive-operand synthesis. Formally, a predicate $p \in \mathcal{P}$ characterizes whether a guarded behavior is activated for opcode op under operands $\langle o_i \rangle$ and architectural context ctx:

$$p(op, \langle o_i \rangle, ctx) \in \{\texttt{true}, \texttt{false}\},$$

where $\langle o_i \rangle$ include register values, immediates, and/or effective addresses, and ctx captures the architectural environment such as current privilege level, relevant CSR settings, and memory translation configuration. Each predicate p is associated with a data-synthesis template that constructs operand families to satisfy p, We categorize data-sensitive operands into two fundamental dimensions:

- *Boundary Values:* These operands are designed to stress the functional units, such as ALU and FPU, by exercising representation and arithmetic corner cases. This category includes integer boundaries, such as `0`, $\pm$`1`, `MIN`/`MAX`, and bit-boundary masks, as well as IEEE 754 [46] corner values like $\pm$`0`, $\pm\infty$, and various `NaN` encodings. Besides, SDModel supports subnormal numbers, identified by a zero exponent field and a non-zero significand, *e.g.*, `0x00000001` in IEEE-754, for exercising slow-path normalization logic .
- *Exception Triggers:* These operands are crafted to intentionally violate ISA-defined preconditions, thereby forcing the core to enter guarded control paths such as legality checks, trap dispatch, and recovery. This category includes (1) memory-related triggers, such as misaligned effective

addresses and access patterns that exercise address translation and permission checks; (2) privilege-related triggers, such as CSR indices and access forms that are disallowed under the current privilege mode; and (3) operation-specific triggers that raise architecturally defined exceptions, such as invalid opcodes, illegal instruction encodings, and arithmetic exception conditions.

Hierarchical Template Construction. After defining the abstraction of SDModel, *DRVFuzz* constructs data-synthesis templates at scale via a three-layer hierarchy. (1) *Operand Primitives:* In this layer, SDModel maintains a repository of sensitive data across different primitive data types, such as integer boundaries, bit-boundary masks, and floating-point corner cases. (2) *Class-Specific Templates:* In this layer, SDModel groups opcodes into semantic classes and instantiates them by combining primitives with class-level constraints. UnifiedDB drives this instantiation by mapping each opcode to its class and exposing operand roles, bit-widths, and encoding fields, while CPU configuration further specializes templates with core parameters such as XLEN, supported extensions, privilege modes, and valid memory regions. (3) *Opcode Refinement Templates:* To preserve instruction-level precision, SDModel applies lightweight refinements to specific opcodes. For example, for `Div` class and `REM` class, SDModel augments divide-by-zero triggers and signed overflow corners. Similarly, for `SQRT` class, SDModel supplements negative values as invalid-operation triggers. This hierarchy enables scalable modeling across large opcode sets via reusable primitives and class templates, while maintaining opcode-specific behaviors through refinements.

Using SDModel in *DRVFuzz*. SDModel provides *DRVFuzz* with two complementary capabilities. (1) *Operand Synthesis.* During testcase generation, *DRVFuzz* selects a target predicate from the applicable predicate set $\mathcal{P}(op, ctx)$ and invokes the corresponding data-synthesis template in SDModel to generate data-sensitive operands as immediates or register values. (2) *Data-Sensitive Labeling.* After executing a testcase, *DRVFuzz* reuses SDModel to label each dynamic instruction instance by analyzing the execution feedback. Given the observed operands $\langle o_i \rangle$ and architectural context ctx, *DRVFuzz* evaluates $p(op, \langle o_i \rangle, ctx)$ to determine which data-sensitive conditions are satisfied during execution. By unifying operand synthesis and data-sensitive labeling, SDModel provides a semantic foundation for transition-guiding fuzzing toward fragile execution conditions.

4.3 Transition-Guided Fuzzing

Sensitive operands do not merely change results; they can configure how the core executes an instruction by activating guarded checks, alternative micro-paths, or recovery routines. The BOOM precise-exception violation in Figure 3 illustrates that the bug manifests only when a misaligned effective address and consecutive store operations drive the LSU alignment validation and ROB retirement logic into an uncommon ordering This motivates a key heuristic in *DRVFuzz*: a newly observed transition between data-sensitive modes is likely to correspond to entering previously unexplored guarded microarchitectural states. Therefore, testcases that induce new transitions should be treated as *interesting seeds* and prioritized for further mutation.

Transition Definition. To systematically quantify these behaviors, we formally define the execution state and state transitions. Let an instruction sequence be $I = \langle \imath_1, \imath_2, \ldots, \imath_n \rangle$. For each executed instruction $\imath_k$, its execution state σ_k is defined as a tuple $\sigma_k = \langle op_k, \mathcal{S}_k \rangle$. op_k represents the unique opcode identity, and $\mathcal{S}_k$ is the set of satisfied SDModel predicates derived from its operands and architectural context:

$$\mathcal{S}_k = \{ p \in \mathcal{P} \mid p(op_k, \langle o_i \rangle_k, ctx_k) = \texttt{true} \}$$

Under this definition, common-case instructions are represented as neutral states $\sigma = \langle op_k, \emptyset \rangle$. This ensures that neutral states remain distinct based on their operation type, *e.g.*, $\langle \texttt{ADD}, \emptyset \rangle$ versus $\langle \texttt{FENCE}, \emptyset \rangle$, preserving execution context.

A *State Transition*, denoted as τ_k, is defined as the directed pair of execution states between two consecutive instructions:

$$\tau_k = (\sigma_k \rightarrow \sigma_{k+1})$$

This transition captures the temporal evolution of data sensitivity. For example, consider two instructions in S-mode: $\sigma_k = \langle \texttt{ADD}, \emptyset \rangle$ and $\sigma_{k+1} = \langle \texttt{LW}, \{\texttt{Misaligned}, \texttt{PageFault}\} \rangle$. The resulting transition $\tau_k = (\sigma_k \rightarrow \sigma_{k+1})$ indicates that execution moves from a common-case datapath into a guarded exception handling and recovery logic. Two transitions are consider equal *iff* they share the same source and destination opcodes and the same SDModel predicate sets, i.e., $(op_k, \mathcal{S}_k) \rightarrow (op_{k+1}, \mathcal{S}_{k+1})$.

Transition-Guided Fuzzing. *DRVFuzz* uses state transitions τ_k as the fuzzing feedback: a previously unseen transition likely indicates that execution has entered new guarded microarchitectural logic. Accordingly, *DRVFuzz* treats testcases that yield new transitions as *interesting seeds* and prioritizes them for further fuzzing. Algorithm 1 illustrates the detailed procedure of transition-guided fuzzing. In each iteration, *DRVFuzz* selects a *seed* from the seed pool and mutates it into a new testcase tc based on the SDModel, similar to prior CPU fuzzers [9,53]: it either performs SDModel-guided operand mutation or applies lightweight random perturbations. Concretely, for SDModel-guided mutation, *DRVFuzz* samples a target opcode op and a sensitive predicate $p \in \mathcal{P}(op, ctx)$, and then instantiates operands via the corresponding template to satisfy p; alternatively, it randomly mutates opcodes and operands (Lines 5-6). Note that, *DRVFuzz* maintains operand-level data dependencies across instructions during testcase generation, so that synthesized sensitive values are propagated to subsequent uses, thereby sustaining fragile conditions. Then, *DRVFuzz* executes tc on both the DUT and ISS

Algorithm 1: Transition-Guided Fuzzing

Input: *SDModel*: Sensitive Data Model
DUT: CPU under test
ISS: Reference model
Output: $\mathcal{B}$: Discovered bugs

```
1  B ← ∅; SeedPool ← InitSeeds();
2  // Visited transitions
3  Visited ← ∅
4  while true do
5      seed ← SelectSeed(SeedPool);
6      tc ← Mutate(seed, SDModel);
7      // Execute and differential test
8      (Δ, Trace) ← ExecuteAndDiff(tc, DUT, ISS);
9      if Δ ≠ ∅ then
10         B ← B ∪ {Report(tc, Δ)};
11         SeedPool ← SeedPool ∪ {tc};
12         continue;
13     end
14     // Label execution states and extract transitions
15     Σ ← LabelStates(Trace, SDModel);
16     (T, N) ← ExtractTransitions(Σ, Visited);
17     // Contain new transitions
18     if N ≠ ∅ then
19         Visited ← Visited ∪ N;
20         SeedPool ← SeedPool ∪ {tc};
21     end
22 end
```

reference model for differential testing, reports any architectural mismatch as a bug candidate, and adds tc triggering the bug into seed pool (Lines 8-13). Otherwise, if no mismatch is observed, *DRVFuzz* parses the execution trace and uses SDModel to label each dynamic instruction instance ι_k with an execution state $\sigma_k = \langle op_k, \mathcal{S}_k \rangle$, where $\mathcal{S}_k$ is the set of satisfied SDModel predicates under the observed operands and context (Lines 15-16). Then, *DRVFuzz* constructs all transition instances $\mathcal{T}$ and filters them against the visited transition set *Visited* to isolate the set of previously unseen transitions $\mathcal{N}$. If $\mathcal{N}$ is non-empty, *DRVFuzz* updates *Visited* with N and preserves tc in the *SeedPool* for further usage (Lines 19-21).

Execution Trace Analysis. To enable transition extraction and bug diagnosis, *DRVFuzz* performs execution trace analysis after running each testcase. For each dynamic instruction instance ι_k, *DRVFuzz* collects (1) the decoded opcode and Program Counter (PC), (2) concrete operand values, including register, immediate and memory address, (3) architectural context relevant to SDModel predicates, including privilege mode and key CSR/MMU settings (*e.g.,* `mstatus`, `sstatus`, `satp`, `medeleg`, and PMP configurations), (4) architectural effects observed at retirement, including destination register update, memory write, and trap cause. These comprehensive trace statistics provide the necessary runtime visibility to accurately label execution states, and support differential oracles to detect inconsistencies between the DUT and the ISS.

Bug Diagnosis. When a testcase triggers a differential mismatch between the DUT and the ISS, *DRVFuzz* automatically constructs a diagnosis report to facilitate root-cause analysis. Upon detecting a divergence, *DRVFuzz* localizes the *mismatch point*, which is the earliest retired instruction at which the DUT and ISS differ in architectural observation. *DRVFuzz* then classifies each failure based on the divergence types, including *trap-related* mismatches (wrong trap cause/priority), *data-corruption* mismatches (register value mismatch), *memory-side* mismatches (unexpected store address/data), and other causes.

Bug Reproduce. Since the generated testcases are often long and noisy, *DRVFuzz* minimizes them into deterministic and compact reproducers. *DRVFuzz* first validates determinism by re-running the testcase and checking that the same mismatch consistently occurs. It then applies a delta-debugging reduction that removes instructions in a binary-search way and re-executes each reduced candidate on the DUT and ISS, keeping the deletion only if the failure persists, until no further reduction is possible. To avoid breaking fragile conditions, *DRVFuzz* preserves the essential data-flow slice that feeds the mismatch-triggering instructions, and retains any required setup instructions that establish the architectural context. To prevent redundant reporting, *DRVFuzz* computes a bug signature from the mismatch type, the mismatch-point instruction, and the mismatch behaviors. Failures sharing the same signature are identified as the same issue.

5 Implementation

We implemented *DRVFuzz* and evaluated it on six widely used open-source RISC-V CPUs. Table 1 summarizes the detailed information of these targets, including their supported ISA variants and repository versions. The selected CPUs cover a broad spectrum of real-world RISC-V designs. Rocket and two variants of BOOM, *i.e.*, BOOM-V3 and BOOM-V4, are Chisel-based cores from the Berkeley ecosystem. Rocket is a mature in-order core, whereas BOOM variants provide out-of-order execution, with BOOM-V4 introducing a refactored load-store unit and broader support for the Zba/Zbb/Zbs extensions. CVA6, maintained by the OpenHW Group, is a Linux-capable core that is actively used in both academic research and industrial projects. Kronos and Srv32 are RV32 CPUs that implement a complete instruction execution pipeline. Together, these CPUs differ substantially in XLEN, including both RV32 and RV64, as well as in ISA extensions and microarchitectural complexity, demonstrating that *DRVFuzz* is applicable across diverse RISC-V designs.

Figure 6 presents the three core components of *DRVFuzz*. The first part is the Configuration Constructor, responsible for parsing the RISC-V UnifiedDB and target CPU configurations to construct SDModel used for data-sensitive operand

Table 1: Detailed information of target RISC-V CPUs.

CPUs	Architecture	Version
Rocket	RV64IMAFDCB	Master, 960396f
BOOM-V3	RV64IMAFDC	Master, 7d1b075
BOOM-V4	RV64IMAFDCB	Master, 7d1b075
CVA6	RV64IMAFDC	Master, 100bb05
Kronos	RV32IMC/RV32EC	Master, 13678d4
Srv32	RV32IMC/RV32EMAC	Master, 3081aa9

synthesis. The second part is the Testcase Generator, which labels execution states, extracts state transitions, and employs transition-guided fuzzing to generate instruction sequences that explore guarded and fragile microarchitectural behaviors. The third part is the Inconsistency Detector, which executes generated testcases on the target CPU and a reference ISS to detect architectural mismatches and automatically diagnose and minimize failing testcases. Only the third part requires minor modifications when adapting to new CPUs. The rest of the section describes notable implementation details.

Integration with New RISC-V CPUs. The effort of adapting *DRVFuzz* to new RISC-V CPUs is small. When targeting a new CPU under test, developers only need to provide a small amount of CPU-specific information and interfaces. First, *DRVFuzz* requires a trace extraction interface that exposes architectural states needed for differential testing and execution-state labeling, including committed instructions, register updates, memory accesses, and relevant CSRs. Most open-source RISC-V CPUs already expose such information through commit logs, waveform signals, or RVFI-style traces, so adaptation typically involves a lightweight wrapper that maps existing signals into *DRVFuzz*'s unified trace format. Second, *DRVFuzz* relies on the target CPU's existing simulation or testbench infrastructure to execute generated testcases, which can usually be reused directly without modification. In addition, to support a new CPU, *DRVFuzz* requires users to provide a CPU configuration description that specifies supported ISA extensions, XLEN, privilege modes, and memory layout; this description is used to parameterize the Sensitive Data Model and does not require any code changes. All other components of *DRVFuzz* can be reused without modification.

6 Evaluation

In this section, we evaluated *DRVFuzz* to answer the following three research questions:

- **RQ1:** Is *DRVFuzz* effective in detecting vulnerabilities in the real-world RISC-V CPUs?
- **RQ2:** Can *DRVFuzz* achieve higher microarchitectural coverage compared to state-of-the-art methods?
- **RQ3:** How do the SDModel and Transition-Guided Fuzzing contribute to the effectiveness of *DRVFuzz*?

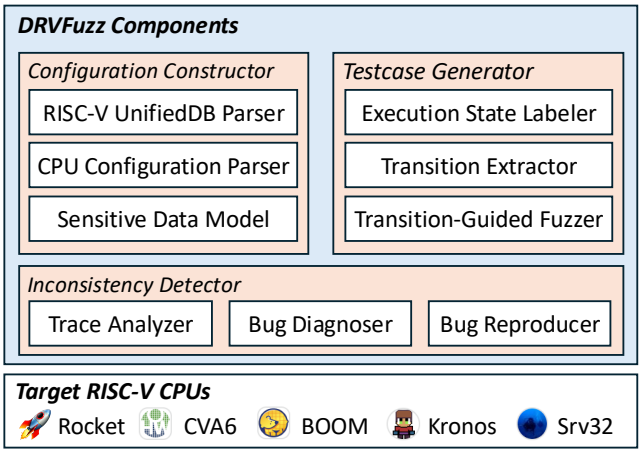


Figure 6: Core components of *DRVFuzz* implementation, contains three key parts: Configuration Constructor, Testcase Generator, and Inconsistency Detector.

6.1 Experiment setup

Subject. We evaluated *DRVFuzz* on six widely used RISC-V CPUs, as shown in Table 1. The selection of CPUs was guided by two considerations: (1) the chosen designs are representative, commonly adopted RISC-V cores in real deployments, allowing us to assess whether *DRVFuzz* can expose bugs with practical impact; and (2) the set intentionally spans diverse ISA profiles and microarchitectural styles (e.g., RV32/RV64 and multiple extension combinations), enabling a comprehensive evaluation of *DRVFuzz* across different pipeline organizations and implementation choices. Appendix A summarizes the supported RISC-V instruction extensions, covering major instruction categories from arithmetic and floating-point operations to privilege-state management.

Compared Tools. We compared *DRVFuzz* with two state-of-the-art hardware fuzzers: Cascade [47] and DiveFuzz [23]. Cascade represents a leading approach which constructs intricate instruction streams with entangled data flow and control. DiveFuzz focuses on feedback mechanism, which enhances fuzzing granularity via diverse register writeback behaviors. These two tools serve as robust baselines for evaluating *DRVFuzz* against established standards in both instruction sequence complexity and feedback-driven guidance.

Metrics and Settings. We evaluated each tool using three metrics: (1) the number of unique bugs detected, (2) microarchitectural coverage of the target CPUs, and (3) the average time speed-up for bug localization. These metrics are widely adopted in prior hardware fuzzing studies [8, 22, 47]. All tools were executed in the same environment on a machine with an Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz, 32 GiB DDR4 memory, running x86_64 Ubuntu Linux 20.04. We tested all RISC-V CPUs with their default configuration parameters. Each experiment was repeated 10 times under identical conditions, and we report the averaged results.

Table 2: 22 new bugs detected by *DRVFuzz* within 24 hours. *DRVFuzz* discovered all 22 bugs across six RISC-V CPU implementations, while other tools detect no more than 4.

#	Platform	Bug Type	The Root Cause Analysis	CVE
1	BOOM-V3	Decoding Error	`fcvt.l.s` does not raise illegal instruction exception for dyn rounding mode when frm is invalid.	CVE-2025-70451
2	BOOM-V3	Misprediction	Frontend prediction incorrectly redirects PC to an invalid address, corrupting `mepc` and causing a trap loop.	CVE-2025-70436
3	BOOM-V3	Exception Logic	`mtval` is not updated with instruction encoding on illegal instruction exceptions.	CVE-2025-70450
4	BOOM-V4	Misprediction	Frontend prediction incorrectly redirects PC to an invalid address, corrupting `mepc` and causing a trap loop.	CVE-2025-70483
5	BOOM-V4	Atomicity Violation	`LSU` fails to suppress misaligned `sd` side-effects after an address exception, causing post-trap trace leakage.	CVE-2025-70441
6	Rocket	Data Misattribute	Tracer incorrectly attributes the writeback of `divw` to the trapping `flw` instruction.	CVE-2025-70442
7	Rocket	Data Misattribute	Tracer logs the `remuw` writeback under the PC of the trapping instruction `fld`.	CVE-2025-70484
8	Rocket	Privilege Violation	Rocket fails to check CSR legality for `mtval2` and allows csrrs to execute without raising an exception.	CVE-2025-70443
9	CVA6	FPU Flag	CVA6 incorrectly treats normal finite operands as invalid, causing the `NV` flag to be set.	CVE-2025-70444
10	CVA6	FPU NaN Logic	`fltq.*` instructions bypass NaN handling and fall back to raw integer comparison.	CVE-2025-70495
11	CVA6	FPU NaN Logic	CVA6 incorrectly treats quiet NaNs operands as signaling NaNs in `fltq.h`, causing the `NV` flag to be set.	CVE-2025-70494
12	CVA6	FPU Arithmetic	`fleq.s` and `fleq.d` instructions incorrectly evaluate equal values as unequal instead of returning true.	CVE-2025-70487
13	CVA6	ISA Compliance	CVA6 does not enforce NaN-boxing for `fsqrt.s`, and directly uses the lower 32 bits for calculation.	CVE-2025-70487
14	Kronos	Privilege Violation	Kronos allows `csrrwi` to access the unsupported H-extension CSR `mtval2` without raising an exception.	CVE-2025-70452
15	Kronos	Privilege Violation	Kronos allows `csrrs` to access performance counters instead of raising an illegal instruction exception.	CVE-2025-70488
16	Kronos	Privilege Violation	Kronos does not raise illegal instruction on `csrrs` writes to read-only Machine CSRs.	CVE-2025-70454
17	Srv32	Privilege Violation	Srv32 allows `csrrsi` to write the read-only `cycle` CSR and fails to raise an illegal instruction exception.	CVE-2025-70856
18	Srv32	Privilege Violation	Srv32 allows `csrrc` to write the read-only `instret` CSR and fails to raise an illegal instruction exception.	CVE-2025-70445
19	Srv32	Privilege Violation	Srv32 treats `csrrci` as a write operation when `uimm = 0`, so a legal read of `time` is incorrectly trapped.	CVE-2025-70485
20	Srv32	Privilege Violation	Srv32 allows `csrrs` to write read-only Machine-mode CSRs without raising an illegal instruction exception.	CVE-2025-70857
21	Srv32	Wiring Error	The `.raddr` port is misconnected to `imem_addr_i` instead of `imem_raddr_i`, breaking the build process.	–
22	Srv32	Config Error	`coverage` is misspelled as `coverate`, so this option is not passed to Verilator, disabling instrumentation.	–

6.2 Bugs in Real-World RISC-V CPUs

We evaluated the bug detection effectiveness of *DRVFuzz* by applying it to six widely-used RISC-V CPUs: BOOM-V3, BOOM-V4, Rocket, CVA6, Kronos, and Srv32. For a comprehensive comparison, we also executed two state-of-the-art hardware fuzzers, Cascade [47] and DiveFuzz [23], under the same configurations. Each tool was granted a 24-hour testing budget per CPU. In total, *DRVFuzz* successfully uncovered 22 previously unknown bugs across the target cores, including 3 in BOOM-V3, 2 in BOOM-V4, 3 in Rocket, 5 in CVA6, 3 in Kronos, and 6 in Srv32. All discovered bugs have been reported to and confirmed by the respective open-source maintainers. The detailed information and root cause analysis on these new bugs is presented in Table 2.

As shown in Table 2, the detected bugs span a wide range of data-sensitive execution behaviors across six RISC-V CPUs. A large portion of the bugs (13 out of 22) are related to incorrect handling of data-dependent instruction semantics, such as FPU arithmetic and NaN processing errors (#9– #13), decoding and exception metadata propagation issues (#1, #3), and mispredicted control-flow interactions that corrupt data-bearing CSRs such as `mepc` (#2, #4). These failures indicate that instruction behavior is highly sensitive to operand values, rounding modes, and intermediate data representations, and that subtle data-dependent corner cases are often overlooked by existing fuzzing approaches. Eight bugs (#8, #14–#20) are caused by privilege violations in CSR access and update logic, where illegal or read-only CSRs are incorrectly accessed or modified depending on instruction operands or immediate values. Such violations demonstrate that data-dependent CSR semantics are not consistently enforced across different implementations. Bugs #21 and #22 are configuration-related issues, which were surfaced during necessary fuzzing steps when generating the simulator model and enabling coverage support. The remaining bugs (#5–#7) stem from data propagation, including pipeline leakage after exceptions and incorrect attribution of writeback data to trapping instructions. These bugs further show that incorrect data flow across pipeline stages or tooling configurations can silently undermine both correctness and observability.

Security impact of bugs. Excluding the configuration issues (#21 and #22), the remaining 20 CVE-assigned bugs expose four categories of security impact. First, control- and trap-state corruption bugs (#2 and #4) can poison privileged trap state such as `mepc`, causing trap return to repeatedly resume from an invalid target and leading to denial-of-service. Second, the precise-exception violation in #5 breaks fault atomicity by making a faulting store appear commit-visible before trap delivery, which can undermine exception recovery. Third, privilege- and CSR-isolation violations (#8 and #14–#20) allow illegal or read-only CSRs to be accessed or modified without the required exception, weakening architectural protection guarantees. Finally, FPU- and ISA-semantic violations (#1, #3, #6, #7, and #9–#13) can corrupt floating-point results, exception metadata, or trace attribution, potentially affecting control decisions and numerical correctness in security-sensitive workloads. Overall, these impacts show that the detected bugs are not merely functional mismatches; they can compromise availability, privilege isolation, precise exception handling, and architectural correctness.

Comparison with Existing Fuzzers: In our 24-hour experiments, Cascade and DiveFuzz identified only 3 (Bug #3,

Table 3: Bugs found by *DRVFuzz* and other state-of-the-art methods. Existing hardware fuzzers detect no more than 4 bugs, while *DRVFuzz* detects all 22 bugs.

Tools	Bug Number	Bugs ID #
DRVFuzz	22	#1 – #22
DiveFuzz	4	#3, #9, #21, #22
Cascade	3	#3, #21, #22

#21, #22) and 4 bugs (Bug #3, #9, #21, #22), respectively. As shown in Table 3, they failed to uncover the remaining 18 bugs (e.g., Bug #1, #16, and #20) mainly because these bugs reside in data-dependent microarchitectural logic that is triggered only under specific operand values and semantic stat es. Many missed cases involve subtle floating-point corner semantics (e.g., NaN handling, rounding-mode constraints) or operand-sensitive privilege/CSR behaviors, where incorrect checks or updates manifest only for particular immediates or CSR encodings. However, existing tools are largely data-agnostic: they emphasize instruction-sequence diversity but generate operands as incidental random bit-patterns, making it unlikely to activate such value-specific conditions. In contrast, *DRVFuzz* leverages SDModel to synthesize semantically meaningful operands and employs Transition-Guided Fuzzing to systematically explore interactions across sensitive-data states. Together, these two components enable *DRVFuzz* to uncover all 22 new bugs within 24 hours, demonstrating advantages in detecting data-sensitive hardware bugs, which adequately answers **RQ1**. Compared with other state-of-the-art CPU fuzzing technologies, *DRVFuzz* found all bugs that other methods found.

6.2.1 Case Study

We now present a representative case to illustrate the security severity of the vulnerabilities detected by *DRVFuzz*, and how *DRVFuzz* found them.

This case corresponds to Bug #2 and #4 in Table 2. We found that, in both the BOOM-V3 and BOOM-V4, a BTB (Branch Target Buffer) misprediction can become architecturally visible after two back-to-back exceptions: an initial load page fault followed by an illegal-instruction fault. In this situation, BOOM may redirect the fetch PC to an invalid BTB target, and the subsequent instruction-side trap records this mispredicted PC in `mepc`, causing `mret` to resume from an incorrect address.

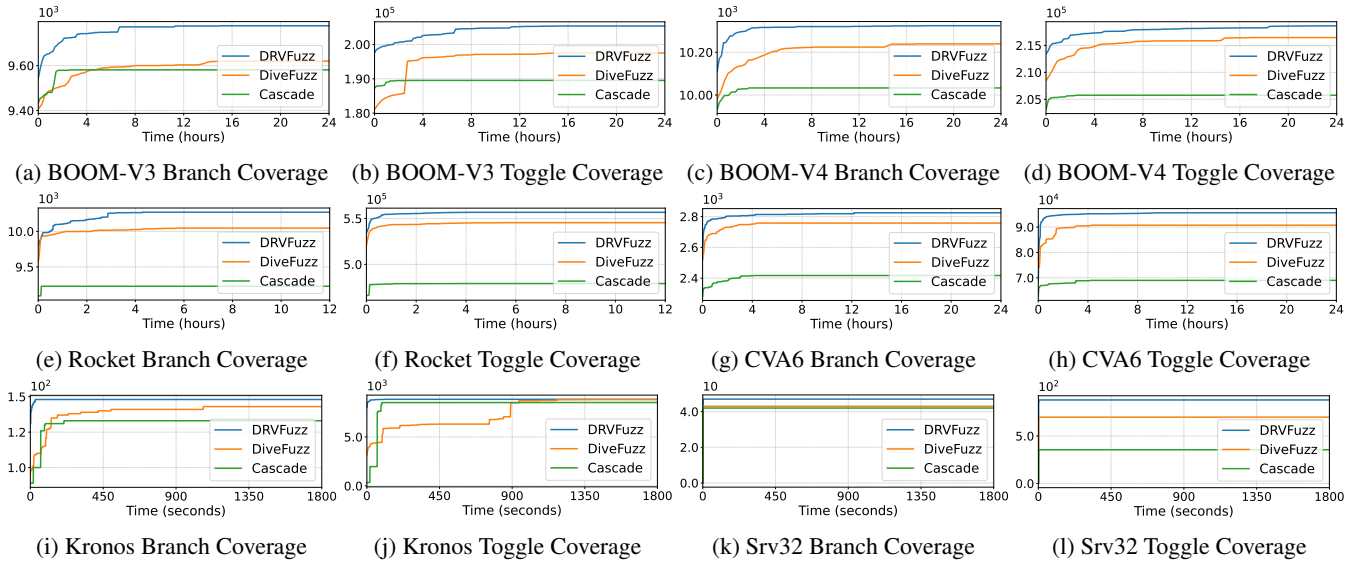
Figure 7 shows a minimized reproducer. The testcase first installs a machine-mode trap handler (`mtvec`), then switches to S-mode with Sv39 enabled (line 6). It intentionally loads from an unmapped virtual address to trigger a load page fault (line 10). Immediately after returning, it executes `fround.s`, which triggers an illegal-instruction fault in the BOOM (line 12). After this pair of consecutive exceptions, the frontend of

```
1  # mepc becomes the speculative BTB target.
2      la      t0, trap_handler
3      csrw    mtvec, t0
4  start:
5      # in S-mode with Sv39 enabled
6      jal     ra, switch_to_s_mode
7      # unmapped virtual memory
8      lui     a1, 0x40000000
9      # load page fault
10     c.ld    a2, 0(a1)
11     # illegal instruction fault in BOOM
12     fround.s f17, f16, dyn
13     # Mispredict to 0xffffffec8e7b600
14     fsub.s f16, f16, f17, rtz
15 trap_handler:
16     # Invalid PC addr 0xffffffec8e7b600
17     csrr    t1, mepc
18     addi    t1, t1, 4
19     csrw    mepc, t1
20     mret
```

Figure 7: Simplified instruction sequence for case study. After an initial load page fault (`c.ld`) and an invalid instruction falult `fround.s`, a wrong BTB prediction redirects the fetch PC to an invalid target (`0xffffffec8e7b600`), where instruction fetch raises an instruction-side fault. `mepc` is corrupted by this poisoned PC, leading to a denial-of-service via repeated traps.

BOOM consults the BTB and is redirected to a mispredicted target (`0xffffffec8e7b600`) instead of the architectural fall-through. The core then fetches from this invalid address and raises an instruction page fault, transferring control to `trap_handler`. Critically, the handler observes `mepc` pointing to the poisoned BTB target rather than the correct next PC in program order (line 17). With a common recovery pattern that advances `mepc` (e.g., `mepc+=4`), `mret` repeatedly returns to an incorrect address and re-enters the trap handler, resulting in a trap loop. The root cause is a squash-synchronization bug between the BTB and the exception-flush pipeline. When exceptions occur back-to-back, the execution latency allows a stale BTB target to bypass internal suppression and survive recovery, erroneously driving the next instruction fetch.

Security impact. This bug breaks the architectural guarantee that trap state is derived from the committed control flow. By poisoning `mepc` with a BTB-generated target, BOOM allows a microarchitectural prediction error to corrupt privileged architectural state, leading to reliable denial-of-service via repeated traps. Because privileged software relies on `mepc` for fault attribution and recovery, a poisoned `mepc` can mislead trap-handling logic and trigger incorrect recovery actions. More broadly, this bug provides a primitive for undermining control-flow integrity: If an adversary trains the BTB to target a valid executable address, `mret` would resume execution at an unintended location, effectively bypassing security isolation and hijacking the privileged control flow.



(a) BOOM-V3 Branch Coverage (b) BOOM-V3 Toggle Coverage (c) BOOM-V4 Branch Coverage (d) BOOM-V4 Toggle Coverage

(e) Rocket Branch Coverage (f) Rocket Toggle Coverage (g) CVA6 Branch Coverage (h) CVA6 Toggle Coverage

(i) Kronos Branch Coverage (j) Kronos Toggle Coverage (k) Srv32 Branch Coverage (l) Srv32 Toggle Coverage

Figure 8: Comparison of the coverage of branch and toggle for *DRVFuzz*, DiveFuzz, and Cascade over time. *DRVFuzz* consistently achieves the highest coverage on six RISC-V CPUs.

How *DRVFuzz* exposes this bug. The manifestation of this bug requires satisfying coupled data-sensitive constraints. Specifically, it is triggered when a page fault is immediately chained with an illegal-instruction trap, which reliably drives BOOM into the narrow recovery window where a stale BTB target can pollute `mepc`. Such a condition is difficult to reach through generic random operand generation, because it requires not only sensitive operand values, such as an unmapped virtual address, but also a fragile sequence of exception-triggering instructions. Given that Cascade and DiveFuzz lack explicit data-sensitive guidance, producing a single required sensitive operand is largely incidental, making such coupled conditions unlikely reachable. In contrast, *DRVFuzz* leverages SDModel to explicitly synthesize tailored operands for satisfying sensitive constraints, *e.g.*, page faults, while employing transition-guided fuzzing to identify and prioritize the rare transitions. This enables *DRVFuzz* to effectively and continuously explore fragile execution states and trigger latent bugs hidden in data-sensitive paths.

6.3 Microarchitectural Coverage

To evaluate *DRVFuzz*'s performance in terms of coverage, we compared Cascade, DiveFuzz, with *DRVFuzz* using commonly used coverage metrics across six RISC-V CPU implementations: (1) Branch coverage measures how many RTL control-flow branches (e.g., `if`/`else` and `case` outcomes) are exercised, reflecting the diversity of explored control paths. (2) Toggle coverage measures how many signal bits toggle, reflecting the activation of internal datapath and state logic. To track the trends of coverage growth, we record both the branch coverage and toggle coverage every minute over 24 hours, as shown in Figure 8. For Kronos and Srv32, which feature smaller design scales and exhibit rapid saturation, we truncate their plotted statistics to the first 30 minutes for clarity.

According to Figure 8, these figures indicate that all three fuzzers exhibit rapid coverage growth in the early phase across all six RISC-V CPUs, especially within the first 1–2 hours, where easily reachable control paths and datapath activities are quickly exercised. However, Cascade saturates early on most CPUs, showing little coverage improvement after the initial phase, while DiveFuzz achieves moderate gains but gradually converges after several hours. In contrast, *DRVFuzz* consistently achieves higher branch and toggle coverage across all evaluated CPUs. Although the coverage advantage of *DRVFuzz* is relatively small in the very early stage, it continues to explore new microarchitectural states over time and steadily widens the coverage gap compared to the other tools. This trend is particularly evident on more complex out-of-order designs such as BOOM-V3, BOOM-V4, Rocket, and CVA6, where *DRVFuzz* maintains sustained coverage growth even after other fuzzers have plateaued. For smaller cores such as Kronos and Srv32, all tools converge faster due to the limited microarchitectural complexity. Nevertheless, *DRVFuzz* still achieves the highest final branch and toggle coverage, indicating its effectiveness even under constrained state spaces. Overall, these results demonstrate that *DRVFuzz* explores a broader set of control-flow paths and internal datapath activities, leading to consistently higher microarchitectural coverage across diverse RISC-V implementations.

To validate that transition-guided fuzzing indeed drives execution into more diverse data-sensitive behaviors, we also measure the number of *unique transitions* exercised by each fuzzer. Recall that a transition $\tau_k = (\sigma_k \rightarrow \sigma_{k+1})$ connects

Table 4: Average unique transitions per 1000 instructions. *DRVFuzz* outperforms all baselines across six RISC-V CPUs.

Tools	DiveFuzz	Cascade	*DRVFuzz*	Improvement	
				vs DiveFuzz	vs Cascade
BOOM-V3	247	151	**341**	38.1%↑	125.8%↑
BOOM-V4	254	147	**350**	37.8%↑	138.1%↑
Rocket	246	172	**352**	43.1%↑	104.7%↑
CVA6	248	124	**348**	40.3%↑	180.6%↑
Kronos	159	75	**226**	42.1%↑	201.3%↑
Srv32	245	115	**349**	42.4%↑	203.5%↑

two consecutive execution states $\sigma = \langle op, \mathcal{S} \rangle$, where $\mathcal{S}$ is the set of satisfied SDModel predicates. We count unique transitions τ_k, and two transitions are considered duplicates *iff* they share the same source and destination opcodes and the same SDModel predicate sets, i.e., $(op_k, \mathcal{S}_k) \rightarrow (op_{k+1}, \mathcal{S}_{k+1})$; otherwise they are distinct. The average unique transitions per 1000 instructions for each tool on six RISC-V CPUs are shown in Table 4.

Overall, *DRVFuzz* consistently triggers the largest number of unique transitions across all targets, indicating that transition-guided fuzzing effectively drives execution to traverse a broader spectrum of data-sensitive behaviors. Compared with DiveFuzz, *DRVFuzz* achieves a stable improvement of 37.8%–43.1% on BOOM-V3/V4, Rocket, and CVA6, and similarly improves by 42.1% on Kronos and 42.4% on Srv32. The gap is even more pronounced when compared with Cascade, where *DRVFuzz* triggers 104.7%–203.5% more unique transitions, suggesting that Cascade tends to revisit similar state transitions due to its largely data-agnostic operand generation. These results demonstrate that explicitly modeling sensitive-data predicates and prioritizing unexplored τ_k helps *DRVFuzz* escape transition saturation and effectively explore many more transitions among diverse data-sensitive behaviors, which is essential for exposing data-dependent microarchitectural corner cases and detecting hidden bugs. These results adequately answer **RQ2**.

6.4 Ablation Study

To evaluate the specific contributions of the SDModel and the Transition-Guided Fuzzing strategy, we conducted an ablation study comparing three variants of our tool: (1) *DRVFuzz*, the full version with both the SDModel and transition-guided fuzzing enabled; (2) *DRVFuzz*$_{m^-}$, a baseline that disables the SDModel and instead generates operands using purely random bit-patterns; (3) *DRVFuzz*$_{g^-}$, a variant that retains the SDModel but replaces the transition-guided strategy, instead randomly mutating the instruction sequences identified by SDModel. and (4) *DRVFuzz*$_{\text{mode}}$, a mode-guided variant that retains the SDModel but prioritizes seeds based only on previously unseen data-sensitive modes, without considering

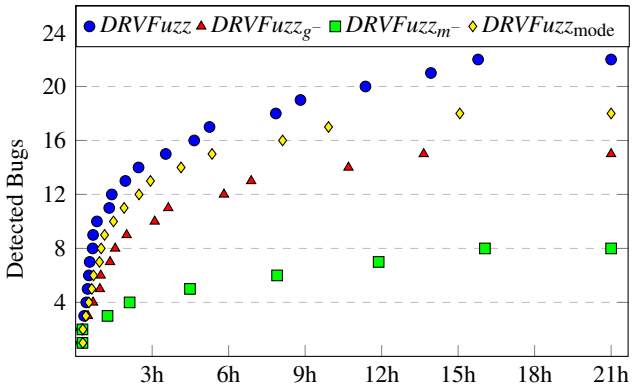


Figure 9: Number of bugs detected by *DRVFuzz*, *DRVFuzz*$_{g^-}$ disabled transition-guided fuzzing, *DRVFuzz*$_{m^-}$ disabled SDModel over time, and *DRVFuzz*$_{\text{mode}}$ using data-sensitive modes instead of transitions.

transitions across consecutive instructions. We measured the number of bugs detected by each variant over a 24-hour testing period on the six RISC-V CPUs. In addition, we recorded the detection time for each identified bug, as shown in Figure 9.

With the support of the SDModel and transition-guided fuzzing, *DRVFuzz* detects all 22 bugs within 24 hours. In contrast, *DRVFuzz*$_{m^-}$ detects only 8 bugs, *DRVFuzz*$_{g^-}$ detects 15 bugs, and *DRVFuzz*$_{\text{mode}}$ detects 18 bugs. The 7 bugs missed by both *DRVFuzz*$_{m^-}$ and *DRVFuzz*$_{g^-}$ (e.g., Bug #2 and #4, detailed in the case study) are triggered only when (i) operands satisfy specific data-sensitive constraints and (ii) execution reaches particular transitions between data-sensitive modes; such conditions are unlikely to be activated by purely random operand generation or unguided exploration. Compared with *DRVFuzz*$_{\text{mode}}$, *DRVFuzz* further detects 4 additional bugs. This shows that reaching individual data-sensitive modes is insufficient for some bugs, because transition-guided fuzzing captures ordered interactions across consecutive modes, which are critical for triggering fragile multi-instruction behaviors. As shown in Figure 9, *DRVFuzz* also detects bugs more efficiently, averaging 43 min per bug. This discovery speed is 2.80× faster than *DRVFuzz*$_{m^-}$ (120 min), 1.27× faster than *DRVFuzz*$_{g^-}$ (55 min), and 1.16× faster than *DRVFuzz*$_{\text{mode}}$ (50 min). Overall, these results confirm the contribution of both components. Compared to *DRVFuzz*$_{m^-}$, *DRVFuzz* demonstrates that the SDModel improves both the number of bugs found and the discovery speed. Compared to *DRVFuzz*$_{g^-}$, the transition-guided strategy further boosts these two metrics by effectively exploring transitions across execution states. More importantly, compared to *DRVFuzz*$_{\text{mode}}$, *DRVFuzz* shows that transition-guided fuzzing provides stronger guidance than mode-based feedback by preserving the ordering relation between consecutive data-sensitive modes. Together, they provide a comprehensive answer to **RQ3**.

7 Discussion

Scope and Extensibility of SDModel. While SDModel enables *DRVFuzz* to effectively exercise data-sensitive paths, it is not a complete semantic specification of all data-dependent behaviors in RISC-V CPUs. First, SDModel focuses on ISA-defined sensitive conditions, such as boundary values and exception triggers. It does not codify core-specific internal policies that are not defined by the ISA, *e.g.*, predictor update, replacement heuristics, and implementation-dependent fast paths. However, SDModel can be extended with microarchitecture-aware predicates and operand templates. For example, to more directly stress predictor-related behaviors, predicates can encode targeted BTB training patterns (e.g., saturating or aliasing predictor entries) before a fault/return sequence, and templates synthesize the corresponding branch streams and fault triggers. Because these extensions reuse the same predicate-template interface, they integrate seamlessly with transition-guided fuzzing.

Second, SDModel currently supports a broad set of RISC-V extensions on both RV32 and RV64, *e.g.*, I/M/F/D/C, Zicsr, Zifencei, Zfh, Zaamo, Zalrsc, covering the majority of semantics of real-world RISC-V CPUs (integer/floating-point arithmetic, atomic operations, bit manipulation, compressed encodings, and privilege control flows). However, certain extensions, such as the V(Vector) and K(Cryptography) extensions, are not yet fully integrated into the SDModel. Integrating these extensions necessitates hierarchically codifying their unique semantics into predicates and operand synthesis templates. For example, this involves modeling predicates for element-wise boundary conditions and mask-register dependencies for the vector extension. We plan to support these extensions in our future work.

More Bug Types Support. *DRVFuzz* currently uses ISA-level differential testing, which is effective for functional, CSR/privilege, and exception-handling bugs, but it may miss issues that are architecturally silent, such as side-channel effects. A straightforward extension for *DRVFuzz* is to add complementary oracles: (1) Side-effect checks (cache footprints, squash windows, privilege-check dominance) to capture transient behaviors, and (2) Trace-consistency checks to validate precise-exception ordering and trap CSR integrity (`mepc`/`mtval`/`mcause`) against committed control flow. Designing scalable detectors that make these checks practical and precise can be explored in future research.

8 Related Work

Mutation-guided CPU Fuzzing. To explore the CPU bug space efficiently, prior works employ mutation-guided fuzzing driven by hardware-specific coverage. Early tools like DIFUZZRTL [27] track register-state and control-signal coverage, while RFUZZ [34] and TheHuzz [31] focus on structural signals such as multiplexer switching. Recent works have introduced more sophisticated feedback metrics. WhisperFuzz [7] targets timing channels by incorporating execution latency and state-transition coverage Geier [18] employs differential analysis across parallel executions to capture subtle microarchitectural leakage signals. Trippel [51] applies traditional software coverage metrics, such as basic-block and path coverage, to drive hardware testing. DiveFuzz [23] enhances feedback granularity by incentivizing diverse instruction writeback behaviors. However, these coverage metrics are data-agnostic. By prioritizing structural or control-flow coverage over operand semantics, they remain insensitive to microarchitectural behaviors that are only activated under specific data-dependent conditions.

Generation-based CPU Fuzzing. Beyond mutation, research has evolved toward generating structured test programs to improve reachability and instruction interaction. Cascade [47] entangles data and control flow to ensure instruction reachability at a program-level granularity. RISCover [50] performs post-silicon differential fuzzing by generating and executing unprivileged RISC-V instruction sequences to identify user-exploitable architectural vulnerabilities. To explore deep architectural states, HyPFuzz [12] and SymbFuzz [37] integrate formal verification and symbolic execution to solve complex path constraints. More recently, ChatFuzz [42] and GenHuzz [52] leverage Large Language Models (LLMs) to synthesize semantically rich instruction sequences. However, they primarily focus on instruction-level reachability rather than the data-sensitivity inherent in the ISA. Consequently, they often fail to exercise data-dependent execution paths.

Main Difference. Different from the above work, *DRVFuzz* focuses on detecting latent bugs that manifest under fragile execution conditions and depend on tailored operand values. To this end, *DRVFuzz* introduces a SDModel that explicitly models data-sensitive semantics across heterogeneous ISA classes, capturing boundary values and exception triggers under relevant architectural contexts, *e.g.*, privilege, rounding, and alignment constraints. To effectively and continuously explore data-dependent microarchitectural paths, *DRVFuzz* employs transition-guided fuzzing that uses the SDModel to label execution states and prioritizes testcases inducing previously unseen state transitions.

9 Conclusion

In this paper, we present *DRVFuzz*, a data-sensitive fuzzing framework for detecting RISC-V CPU bugs that manifest under guarded microarchitectural states. *DRVFuzz* first introduce SDModel to explicitly synthesize tailored operands, and then employs transition-guided fuzzing to steer exploration into data-sensitive paths. We evaluated *DRVFuzz* on six RISC-V CPUs, and it successfully found 22 previously unknown bugs (19 new CVEs). Our coverage experiments further show that *DRVFuzz* consistently achieves higher microarchitectural coverage than state-of-the-art fuzzers.

Acknowledgments

We sincerely thank the reviewers for their valuable comments on this paper. This research is sponsored by the NSFC Program (No.62525207, U2441238).

Ethical Considerations

In conducting our research, we carefully considered the ethical implications of identifying and reporting vulnerabilities within RISC-V CPU implementations. The primary stakeholders include RISC-V core maintainers, downstream SoC/system integrators, OS and hypervisor developers/operators, the hardware-security research community, and end users of systems built on these processors. Our work has clear defensive value for these groups by enabling earlier discovery and remediation of CPU vulnerabilities, but it also has dual-use potential and may impose patching and validation costs.

All experiments were conducted in controlled local RTL/simulation environments using private resources. We did not perform fuzzing on live production hardware services, shared cloud-based CPU instances, or third-party operational infrastructure. All discovered issues were reported through responsible coordinated disclosure. Except for the two configuration issues, the remaining 20 bugs were assigned CVE identifiers and handled following the formal CVE process and standard vulnerability-disclosure practices. We do not release exploit-ready artifacts or deployment-specific attack instructions. With these safeguards in place, we believe that the benefits of publication outweigh the remaining risks, and that this work serves the public interest by improving the safety and accountability of RISC-V processor implementations.

Open Science

We fully support the open science policy. In alignment with this policy, we will openly share all relevant research artifacts associated with our work, including datasets, scripts, binaries, and source code [2]. We acknowledge the importance of this initiative in fostering transparency and collaboration within the research community and are willing to participate in the artifact evaluation process as required.

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[2] Jens Anders, Pablo Andreu, Bernd Becker, Steffen Becker, Riccardo Cantoro, Nikolaos I Deligiannis, Nourhan Elhamawy, Tobias Faller, Carles Hernandez, Nele Mentens, et al. A survey of recent developments in testability, safety and security of risc-v processors. In *2023 IEEE European Test Symposium (ETS)*, pages 1–10. IEEE, 2023.

[3] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1369–1382, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Jean-Loup Baer. *Microprocessor architecture: from simple pipelines to chip multiprocessors*. Cambridge University Press, 2009.

[5] Kristin Barber, Moein Ghaniyoun, Yinqian Zhang, and Radu Teodorescu. A pre-silicon approach to discovering microarchitectural vulnerabilities in security critical applications. *IEEE Computer Architecture Letters*, 21(1):9–12, 2022.

[6] RISC-V BOOM. The berkeley out-of-order riscv processor. https://github.com/riscv-boom/riscv-boom, 2026. Accessed at January 14, 2026.

[7] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. WhisperFuzz:White-Box fuzzing for detecting and locating timing vulnerabilities in processors. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5377–5394, 2024.

[8] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 529–534. IEEE, 2021.

[9] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12, 2023.

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[2] *DRVFuzz*: https://github.com/YzhDDDing/DRVFuzz.

[11] Christopher Patrick Celio. *A highly productive implementation of an out-of-order processor generator*. PhD thesis, University of California, Berkeley, 2017.

[12] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hypfuzz: formal-assisted processor fuzzing. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[13] Yuan-Hu Cheng, Li-Bo Huang, Yi-Jun Cui, Sheng Ma, Yong-Wen Wang, and Bing-Cai Sui. Rv16: An ultra-low-cost embedded risc-v processor core. *Journal of Computer Science and Technology*, 37(6):1307–1319, 2022.

[14] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[15] RISC-V Community. Risc-v unifieddb deployment artifacts. https://riscv-software-src.github.io/riscv-unified-db/, 2026. Accessed at January 14, 2026.

[16] Cpider. Load address misaligned exception can cause the meltdown attack on boom v3. https://github.com/riscv-boom/riscv-boom/issues/698, 2026. Accessed at January 14, 2026.

[17] Enfang Cui, Tianzheng Li, and Qian Wei. Risc-v instruction set architecture extensions: A survey. *IEEE Access*, 11:24696–24711, 2023.

[18] Gideon Geier, Pariya Hajipour, and Jan Reineke. Coverage-guided pre-silicon fuzzing of open-source processors based on leakage contracts. *arXiv preprint arXiv:2511.08443*, 2025.

[19] Rocket Chip Generator. Chips alliance. https://github.com/chipsalliance/rocket-chip, 2026. Accessed at January 14, 2026.

[20] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. A security risc: microarchitectural attacks on hardware risc-v cpus. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2321–2338. IEEE, 2023.

[21] OpenHW Group. Cva6 risc-v cpu. https://github.com/openhwgroup/cva6/, 2026. Accessed at January 14, 2026.

[22] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.

[23] Zihui Guo, Miaomiao Yuan, Yanqi Yang, Liwei Chen, Gang Shi, and Dan Meng. Divefuzz: Enhancing cpu fuzzing via diverse instruction construction. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 1964–1978, New York, NY, USA, 2025. Association for Computing Machinery.

[24] Joel Hestness, Stephen W Keckler, and David A Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 150–160. IEEE, 2014.

[25] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *International Conference on Computer Aided Verification*, pages 521–537. Springer, 2000.

[26] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1473–1487, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[28] Tobias Jauch, Alex Wezel, Mohammad R. Fadiheh, Philipp Schmitz, Sayak Ray, Jason M. Fung, Christopher W. Fletcher, Dominik Stoffel, and Wolfgang Kunz. Secure-by-construction design methodology for cpus: Implementing secure speculation on the rtl. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[29] Stavros Kalapothas, Manolis Galetakis, Georgios Flamis, Fotis Plessas, and Paris Kitsos. A survey on risc-v-based machine learning ecosystem. *Information*, 14(2):64, 2023.

[30] Timothy Kam, Michael Kishinevsky, Jordi Cortadella, and Marc Galceran-Oms. Correct-by-construction microarchitectural pipelining. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 434–441. IEEE, 2008.

[31] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.

[32] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. Microarchitectural mechanisms to exploit value structure in simt architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 130–141, 2013.

[33] kuopinghsu. Srv32: Simple 3-stage pipeline risc-v processor. https://github.com/kuopinghsu/srv32, 2026. Accessed at January 14, 2026.

[34] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[35] Tao Lu. A survey on risc-v security: Hardware and architecture. *arXiv preprint arXiv:2107.04175*, 2021.

[36] Yoshio Masubuchi, Satoshi Hoshina, Tomofumi Shimada, B Hirayama, and Nobuhiro Kato. Fault recovery mechanism for multiprocessor servers. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 184–193. IEEE, 1997.

[37] Samit Shahnawaz Miftah, Amisha Srivastava, Hyunmin Kim, Shiyi Wei, and Kanad Basu. Symbfuzz: Symbolic execution guided hardware fuzzing. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, page 1477–1490, New York, NY, USA, 2025. Association for Computing Machinery.

[38] Gideon Mohr, Marco Guarnieri, and Jan Reineke. Synthesizing hardware-software leakage contracts for risc-v open-source processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[39] Hamza Omar and Omer Khan. Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 111–122. IEEE, 2020.

[40] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1451–1468, 2021.

[41] Spike risc-v isa simulator. Risc-v software. https://github.com/riscv-software-src/riscv-isa-sim, 2026. Accessed at January 14, 2026.

[42] Mohamadreza Rostami, Marco Chilese, Shaza Zeitouni, Rahul Kande, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. Beyond random inputs: A novel ml-based hardware fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.

[43] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. The fuzz odyssey: A survey on hardware fuzzing frameworks for hardware design verification. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, pages 192–197, 2024.

[44] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. The fuzz odyssey: A survey on hardware fuzzing frameworks for hardware design verification. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, GLSVLSI '24, page 192–197, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Jun Sawada and Warren A Hunt Jr. Processor verification with precise exceptions and speculative execution. In *International Conference on Computer Aided Verification*, pages 135–146. Springer, 1998.

[46] IEEE Computer Society. Ieee standard for floating-point arithmetic. https://standards.ieee.org/ieee/754/6210/, 2026. Accessed at January 14, 2026.

[47] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: CPU fuzzing via intricate program generation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5341–5358, 2024.

[48] SonalPinto. Kronos is a 3-stage in-order risc-v core towards fpga implementations. https://github.com/SonalPinto/kronos, 2026. Accessed at January 14, 2026.

[49] Chao Su and Qingkai Zeng. Survey of cpu cache-based side-channel attacks: Systematic analysis, security models, and countermeasures. *Security and Communication Networks*, 2021(1):5559552, 2021.

[50] Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. 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, page 3326–3340, New York, NY, USA, 2025. Association for Computing Machinery.

[51] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254, 2022.

[52] Lichao Wu, Mohamadreza Rostami, Huimin Li, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. *GenHuzz: an efficient generative hardware fuzzer*. USENIX Association, USA, 2025.

[53] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1307–1324, 2023.

[54] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. *Cryptology ePrint Archive*, 2018.

[55] Zehong Yu. Drvfuzz. https://doi.org/10.5281/zenodo.20343558, 2026.

A Instruction Categories

The instruction categories in *DRVFuzz* are dynamically derived from the RISC-V UnifiedDB [15] opcode metadata. This approach ensures that *DRVFuzz* remains synchronized with the evolving RISC-V ecosystem, maintaining a comprehensive alignment with the latest ISA specifications and extensions. Specifically, the supported instruction categories are organized as follows:

- **Base and Integer Arithmetic**: Includes the base integer set (I) and its 64-bit variants. It covers standard ALU operations, multiplications and divisions (M), and atomic memory operations (A) for both 32-bit and 64-bit architectures.
- **Bit-Manipulation Extensions**: Full support for the Zba, Zbb, Zbc, and Zbs extensions, which are essential for modern high-performance computational workloads.
- **Floating-Point Suites**: Comprehensive coverage of single-precision (F), double-precision (D), and quad-precision (Q) floating-point instructions, as well as Zfa extension for advanced floating-point operations.
- **Control Flow and Memory**: Includes direct jumps (JAL), indirect jumps (JALR), conditional branches (BRANCH), compressed instructions (C), and cache-block management extensions, including Zicbom, Zicboz, and Zicbop.
- **System and Privilege States:** This category encompasses instructions for managing deep architectural states, including reading and writing Control and Status Registers (Zicsr), executing transitions between Machine, Supervisor, and User modes (PPFSM, DWNPRV), and handling complex trap logic such as exception delegation (MEDELEG), trap vectoring (TVECFSM), and privilege preservation (EPCFSM).
- **Synchronization and Fences:** DRVFuzz supports rigorous testing of memory ordering and instruction-stream synchronization via Zifencei and standard FENCE instructions.