2026 Design, Automation & Test in Europe Conference (DATE 2026)

SimFuzz: Similarity-guided Block-level Mutation for RISC-V Processor Fuzzing

Hao Lyu, Jingzheng Wu✉, Xiang Ling✉, Yicheng Zhong, Zhiyuan Li, Tianyue Luo
University of Chinese Academy of Sciences
Institute of Software Chinese Academy of Sciences

***Abstract*—The Instruction Set Architecture (ISA) defines processor operations and serves as the interface between hardware and software. As an open ISA, RISC-V lowers the barriers to processor design and encourages widespread adoption, but also exposes processors to security risks such as functional bugs. Processor fuzzing is a powerful technique for automatically detecting these bugs. However, existing fuzzing methods suffer from two main limitations. First, their emphasis on redundant test case generation causes them to overlook cross-processor corner cases. Second, they rely too heavily on coverage guidance. Current coverage metrics are biased and inefficient, and become ineffective once coverage growth plateaus.**

To overcome these limitations, we propose SimFuzz, a fuzzing framework that constructs a high-quality seed corpus from historical bug-triggering inputs and employs similarity-guided, block-level mutation to efficiently explore the processor input space. By introducing instruction similarity, SimFuzz expands the input space around seeds while preserving control-flow structure, enabling deeper exploration without relying on coverage feedback. We evaluate SimFuzz on three widely used open-source RISC-V processors: Rocket, BOOM, and XiangShan, and discover 17 bugs in total, including 14 previously unknown issues, 7 of which have been assigned CVE identifiers. These bugs affect the decode and memory units, cause instruction and data errors, and can lead to kernel instability or system crashes. Experimental results show that SimFuzz achieves up to 73.22% multiplexer coverage on the high-quality seed corpus. Our findings highlight critical security bugs in mainstream RISC-V processors and offer actionable insights for improving functional verification.

***Index Terms*—Processor Verification, Security, Fuzzing**

I. Introduction

The ISA serves as the fundamental bridge between hardware and software, defining the complete set of operations that a processor can interpret and execute [4]. As an open and freely available ISA, RISC-V lowers barriers to processor design and has driven widespread adoption, with millions of processors now deployed [25]. At the same time, RISC-V processors face increasingly severe security challenges. One example is GhostWrite [6], a critical bug in T-Head XuanTie [31] RISC-V processors. It allows attackers to bypass privilege enforcement, gain arbitrary memory access, and control peripheral devices. The root cause lies in the XuanTie processor's support for a non-standard instruction, which violates the ISA specification. Such implementation-specific bugs typically stem from design flaws at the processor's Register Transfer Level (RTL) and ultimately manifest as functional bugs. These bugs directly affect software correctness and can occur even in the absence of adversarial inputs. Unlike software, processors cannot be patched once fabricated [5]. Because the RTL's functionality and timing behavior remain fixed after tape-out, comprehensive RTL-level verification is essential to ensure correctness and enhance security before fabrication.

Modern processors are extremely complex, making exhaustive exploration of their state space infeasible. As a result, processor fuzzing [1], [7], [8], [10]–[12], [16], [18], [30], [32] has emerged as an effective and scalable method for uncovering RTL-level bugs. Current fuzzers typically rely on coverage metrics to track the proxy of the processor state. For example, multiplexer (mux) coverage [18] focuses on multiplexer control signals. During fuzzing, these coverage metrics guide mutation, enabling exploration of a broader range of processor states through mutated seeds.

While promising, coverage-guided processor fuzzers face two major limitations. First, by generating large numbers of redundant test cases, they often fail to capture cross-processor corner cases. For instance, PathFuzz [32] uses real-world programs (e.g., SPEC CPU2006) to expand its seed corpus and fuzz the XiangShan [23] processor. Even with a large number of seeds, it still failed to uncover certain corner-case bugs, such as bug B7 (Table III) identified in our study. Second, fuzzers rely too heavily on coverage guidance. ProcessorFuzzer [11] and DifuzzRTL [7] design new coverage metrics to guide fuzzing, but the choice of metrics often reflects human bias, limiting verification effectiveness [29]. Selecting too few metrics results in insufficient verification, while tracking too many increases overhead and reduces efficiency. Moreover, when coverage plateaus [20], fuzzers lose meaningful guidance, severely restricting exploration. Meanwhile, increased coverage does not necessarily translate to effective bug discovery [15].

Because processor states are driven by inputs, broader state exploration requires a large set of high-quality test cases that can exercise diverse states. We therefore formulate processor verification as the problem of exploring the processor's input space. To this end, we propose SimFuzz. First, SimFuzz builds a high-quality seed corpus by collecting bug-triggering test cases from real-world processors, ensuring that exploration starts in regions of the input space more likely to expose bugs. Second, it preserves the control-flow structure of these seeds during mutation, maintaining semantic validity. Finally, SimFuzz introduces instruction similarity as a guidance mechanism. Unlike coverage metrics, similarity captures fine-grained variations between seeds, enabling more effective input space exploration.

We evaluate SimFuzz on three widely used open-source

978-3-9826741-1-7/DATE26/© 2026 EDAA

RISC-V processors: Rocket [2], BOOM [24], and XiangShan [23]. SimFuzz achieves 73.22% mux coverage on the constructed seed corpus. Compared with seeds generated by Csmith [3] and Cascade [30], SimFuzz slightly outperforms PathFuzz [32] in coverage. More importantly, SimFuzz discovered 17 bugs in total, including 14 previously unknown issues, 7 of which have been assigned CVE identifiers. These bugs affect the decode and memory units, cause misdecoding and data errors, and can lead to kernel instability or system crashes.

In summary, this paper makes three key contributions:

- We construct a real-world, high-quality seed corpus from bug-triggering test cases across three open-source RISC-V processors: Rocket, BOOM and XiangShan.
- We present SimFuzz[1], a new processor fuzzing framework that replaces coverage-guided guidance with similarity-guided block-level mutation.
- Our evaluation shows that SimFuzz achieves higher coverage and discovers 14 previously unknown bugs, 7 of which have been assigned CVE identifiers.

II. Background

A. RISC-V

RISC-V is an open ISA designed for modularity and extensibility [26], [27]. A processor can implement only the minimal base integer instruction set or extend it with additional capabilities. To ensure ecosystem consistency, RISC-V defines a standard general-purpose configuration called the G extension set, which includes Integer, Multiplication/Division, Atomic, and Floating-Point (Single and Double precision) extensions.

The G extension specifies six instruction formats [26]: Register-Register (R-type), Register-Immediate (I-type), Store (S-type), Conditional Branch (B-type), Upper Immediate (U-type), and Unconditional Jump (J-type). Table I summarizes these formats for 32-bit instructions.

TABLE I: RISC-V Instruction Format

31:25	24:20	19:15	14:12	11:7	6:0	Type
funct7	rs2	rs1	funct3	rd	opcode	R
imm		rs1	funct3	rd	opcode	I
imm	rs2	rs1	funct3	imm	opcode	S
imm	rs2	rs1	funct3	imm	opcode	B
imm				rd	opcode	U
imm				rd	opcode	J

The semantics of the instruction determine the state of the processor. In RISC-V instructions, the `opcode` and `funct` fields jointly determine semantics. The `opcode` specifies the instruction format and basic functionality, while the `funct` field refines precise operation executed by the processor. In practice, the processor's internal state space is not directly observable from the input space, making it difficult to reason about state-space coverage explicitly. However, processor state transitions are fundamentally driven by instruction semantics: the processor interprets each instruction according to its semantic definition and updates both its architectural and microarchitectural states. Therefore, even though the exact internal states are inaccessible, the semantic properties of instructions can serve as a reliable proxy for predicting their effects on processor states.

[1]https://github.com/has2lab/SimFuzz

B. Processor Fuzzing

The rapid growth of open-source RISC-V processors has made processor fuzzing an increasingly important verification technique. Fuzzing systematically explores processor states by mutating seeds and observing deviations from expected behavior. RFUZZ [18] was the first hardware fuzzing framework, using mux coverage to verify RTL correctness. Building on this foundation, subsequent research has explored a variety of input generation and mutation techniques.

Generative processor fuzzing requires the creation of a large number of test cases. Cascade [30] introduces an asymmetric ISA pre-simulation mechanism that intertwines control and data flows during input generation. However, its seeds often cluster close together in the input space, limiting diversity and slowing coverage growth. ChatFuzz [9] and GenHuzz [19] apply reinforcement learning to generate test cases. While promising, these approaches incur high computational and time costs and struggle to explore rare but critical states. In summary, although generative fuzzing can produce many seeds and achieve high coverage, it introduces substantial redundancy and remains ineffective at exposing certain corner cases.

Coverage-guided mutation fuzzing modifies existing seeds at the instruction level to expand the input space and uncover corner-case processor states. DifuzzRTL [7], TheHuzz [10], and MorFuzz [8] apply per-instruction mutations guided by coverage metrics. However, these techniques often lose the semantic structure of the original seeds, disrupting control flow and producing invalid or less effective test cases. PathFuzz [32] preprocesses seeds by converting memory from a linear to a footprint layout and then uses LibAFL [14] to guide mutation with coverage metrics. As a software fuzzing library, LibAFL lacks hardware awareness. Overall, these coverage-guided mutation approaches rely heavily on coverage metrics and overlook the semantic structure of seeds, limiting their ability to fully exploit the information embedded in existing test cases.

III. SimFuzz

SimFuzz is a processor fuzzing framework that explores the input space using a similarity-guided mutation strategy, rather than relying on coverage feedback. As shown in Fig. 1. SimFuzz begins by constructing a high-quality seed corpus (Section III-A) from test cases that previously triggered real bugs. During mutation (Section III-B), it preserves each seed's control-flow structure while expanding the input space based on instruction similarity. Finally, the mutated seeds are evaluated through differential testing (Section III-C) to expose inconsistencies between the processor under test and ISA simulators.

A. High-quality Seed Corpus Construction

The quality of the seed corpus strongly affects fuzzing effectiveness [17]. High-quality seeds guide the fuzzer toward critical execution paths more quickly, increasing the likelihood of early bug discovery. Unlike earlier fuzzers [11], [12], [18]

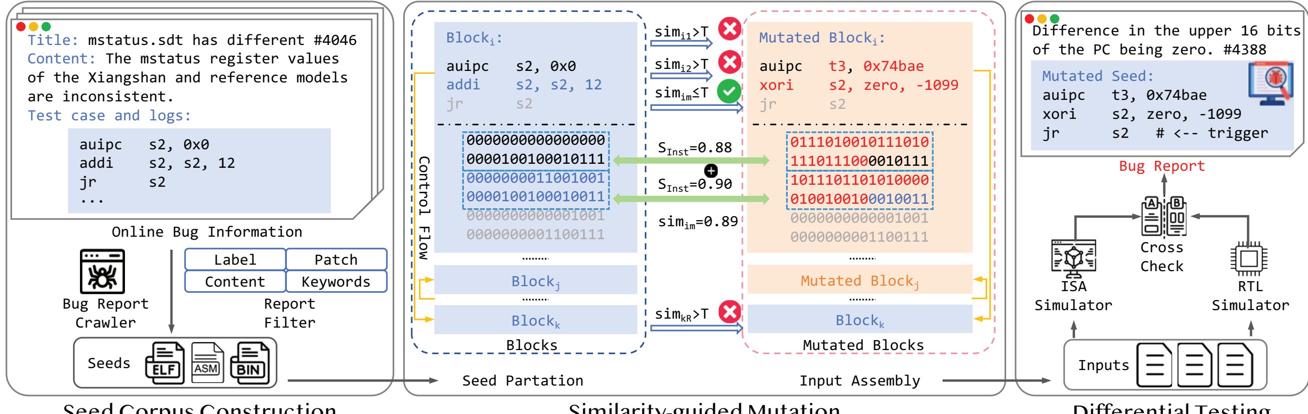


Fig. 1: Overview of SimFuzz workflow.

that used empty corpora or focused on artificially generated seeds [9], [19], [30], SimFuzz builds its corpus directly from real-world bug-triggering test cases. The key insight is that processor bugs often arise from misinterpretations of the ISA specification or subtle design deviations. By reusing test cases that have already triggered real bugs, SimFuzz ensures that exploration starts from meaningful points in the input space rather than random points.

To build its seed corpus, SimFuzz implements an online bug-report crawler and filter targeting three major open-source RISC-V processors: Rocket [2], BOOM [24], and Xiang-Shan [13]. Bug information is gathered from GitHub issues and CVE entries. GitHub reports are automatically filtered based on tags, patches, titles, and descriptions, then categorized by available testing resources: some contain complete executable test cases, others include partial code snippets requiring completion, and some provide only textual descriptions, necessitating manual analysis and test case construction. Similarly, CVE entries from the past decade are analyzed to identify relevant bugs and reconstruct test cases when possible. All resulting test cases, whether executable, completed, or manually created, are consolidated into a real-world seed corpus, which forms the foundation of SimFuzz.

These seeds not only enable high-quality fuzzing but also highlight error-prone instructions and functional units in RISC-V processors, providing valuable guidance for both researchers and vendors.

B. *Similarity-guided Mutation*

All seeds in the seed corpus are initial anchor in processor input space exploration. To expand the input space while preserving semantic structure of seeds, SimFuzz introduces similarity-guided block-level mutation.

1) Block-level Mutation: A block is defined as a continuous sequence of instructions with a single exit point, such as jumps, branches, or system calls, which alters the control flow. The exit of a block is a Control Transfer Instruction (CTI), whose position determines the block boundary. Formally, for a given seed S, the seed is segmented as N blocks: $S = \{B_1, B_2, ..., B_N\}$. Each block B_i contains a fixed number M_i of instructions ($Inst$), such that for each $i \in N$, $B_i = \{Inst_1, Inst_2, ..., Inst_{M_i-1}, CTI\}$. CTIs include target addresses, and modifying them can directly affect the processor's control flow. Changes to CTIs may cause test programs to crash or deviate from the intended execution path, limiting deep exploration of the instruction stream. Therefore, block mutation deliberately avoids altering CTIs, preserving the seed's original control flow as much as possible.

During block mutation, SimFuzz selects several instructions within the block for individual instruction-level mutation. The mutated instructions replace their original counterparts and are reinserted into the block, producing a new, mutated version. After mutation, the similarity between the mutated block and the original block is computed based on instruction-level similarity, as defined in Algorithm 1.

Algorithm 1 Block Similarity Calculation Function

1: **function** BLOCKSIMILARITY(B_1, B_2)
2: $\quad sim \leftarrow 0.0$; $size \leftarrow \min(|B_1|, |B_2|)$
3: $\quad$ **for** $i = 1$ to $size$ **do**
4: $\quad\quad sim \leftarrow sim + \mathrm{S}(B_1[i], B_2[i]) \times (B_1[i] \neq B_2[i])$
5: $\quad$ **end for**
6: $\quad$ **return** $sim/size$
7: **end function**

2) Similarity-guided: The core idea is to mutate blocks while maintaining control flow. Algorithm 2 outlines the full similarity-guided block mutation process. A mutation is accepted if the similarity between a mutated block and its original remains above a threshold T, allowing the input space to expand. If the similarity does not meet T after multiple attempts, the block is retained without further changes. This approach ensures that mutations remain semantically meaningful while introducing diversity. Block similarity is determined by the similarity of the instructions it contains.

Instruction-level semantic similarity, modeled across execution units, reflects the similarity of the processor state transitions induced by the instructions. Based on this observation, we design an instruction similarity metric that approximates

Algorithm 2 Similarity-guided Mutation Algorithm

Require: Seed corpus $\mathcal{S} = \{S_1, S_2, \dots, S_K\}$, T
1: $\mathcal{S}^* \leftarrow \emptyset$
2: **for** each seed $S \in \mathcal{S}$ **do**
3: $\{B_1, B_2, \dots, B_N\} \leftarrow$ SegmentSeed(S)
4: **for** $i = 1$ to N **do**
5: **for** $j = 1$ to R **do**
6: $B_i^* \leftarrow$ **BlockMutation**(B_i)
7: $sim \leftarrow$ **BlockSimilarity**(B_i, B_i^*)
8: **if** $sim < T$ **then**
9: $B_i \leftarrow B_i^*, break$
10: **end if**
11: **end for**
12: **end for**
13: $S' \leftarrow$ ReassembleSeed($\{B_1, B_2, \dots, B_N\}$)
14: $\mathcal{S}^* \leftarrow \mathcal{S}^* \cup \{S'\}$
15: **end for**
16: **return** $\mathcal{S}^*$

processor state-space similarity by analyzing instruction semantics at multiple granularities. Given two RISC-V instructions, Instruction A (I_a) and Instruction B (I_b), the similarity function decomposes instruction similarity into four components: instruction type, opcode, sub-semantic, and field-level similarity.

$$S(I_a, I_b) = w_1 S_{\text{tp}}(I_a, I_b) + w_2 S_{\text{op}}(I_a, I_b) + w_3 S_{\text{sm}}(I_a, I_b) + w_4 S_{\text{F}}(I_a, I_b) \tag{1}$$

a) Type similarity.: Instruction type similarity measures the similarity between the encoding layouts of two instructions. In RISC-V, instruction types (e.g., R, I, S) define the syntactic structure of instructions, including the number and positions of source registers, destination registers, and immediate fields. Instructions sharing the same type are assigned the highest type similarity, as they follow identical encoding layouts. For instructions of different types, similarity is determined based on the structural overlap of their encoding layouts.

b) Opcode similarity.: Opcode similarity captures the coarse-grained functional relatedness between instructions. The opcode primarily defines the high-level operation class of an instruction, such as arithmetic computation, memory access, control flow, or system operations. Instructions with identical opcodes are assigned the highest opcode similarity. Opcodes within the same functional category receive partial similarity. Instructions with unrelated opcodes are assigned low similarity.

c) Sub-semantic similarity.: While opcode similarity captures coarse functional behavior, it cannot distinguish fine-grained semantic differences among instructions. To address this, we introduce sub-semantic similarity, which models instruction semantics at a finer granularity by explicitly considering processor execution units. Instructions are first grouped by their primary execution units. Instructions dispatched to the same execution unit are assigned higher sub-semantic similarity, while instructions targeting different units receive lower similarity. Within the same execution unit, sub-semantic similarity further differentiates instructions based on operational characteristics, such as arithmetic operation type, comparison semantics, or data movement patterns.

d) Filed similarity.: Finally, field-level similarity measures the similarity of instruction operands and encoding fields (e.g., registers and immediates), capturing variations that may influence data dependencies and microarchitectural behavior without changing the instruction's core semantics.

The field similarity is computed using the normalized Hamming distance between the corresponding instruction fields:

$$S_{\text{F}}(I_a, I_b) = \sum_{i \in \mathcal{F}} w_i \left(1 - H(f_i^{(a)}, f_i^{(b)})\right) \tag{2}$$

In (2):

- $\mathcal{F}$ denotes the set of considered instruction fields, such as `func3`, `func7`, `func2`, and `operands`.
- $f_i^{(a)}$ and $f_i^{(b)}$ are the values of field i in instructions I_a and I_b, respectively.
- $H(x, y)$ denotes the normalized Hamming distance between two bit-vectors x and y, defined as:

$$H(x, y) = \frac{\text{HammingDist}(x, y)}{\text{bitlength}(x)}$$

- w_i is the weight assigned to field i, subject to:

$$\sum_{i \in \mathcal{F}} w_i = 1$$

The overall similarity (1) is a weighted combination of these measures, capturing both syntactic structure and semantic intent. This fine-grained metric enables effective clustering and mutation of instructions, producing valid and diverse test cases.

C. Differential Testing

After mutation, blocks are reassembled into complete test cases and executed on both RTL processor simulators and ISA reference simulators. Discrepancies in their behavior indicate potential bugs. Using this differential testing framework, SimFuzz is able to reveal inconsistencies in Rocket, BOOM, and XiangShan, uncovering previously unknown bugs.

IV. Evaluation

We implemented SimFuzz for the RISC-V GC instruction set, which is supported by all general-purpose processor. The evaluation focuses on two aspects: coverage (Section IV-A) and bug discovery (Section IV-B).

A. Coverage

1) Setting: Coverage is measured using the xfuzz testing framework [21], which supports multiple coverage metrics: `line`, `mux`, `toggle`, and `branch`. We focus on mux and toggle coverage, as these metrics are specifically designed for hardware and evaluated by other works.

The seed corpus includes 42 real-world, historical bug-triggering test cases (Historical) that we collected, along with test cases generated by the random C program generator Csmith [3], used by the BOOM project, and test cases produced by Cascade [30], a state-of-the-art processor fuzzing test case generation framework.

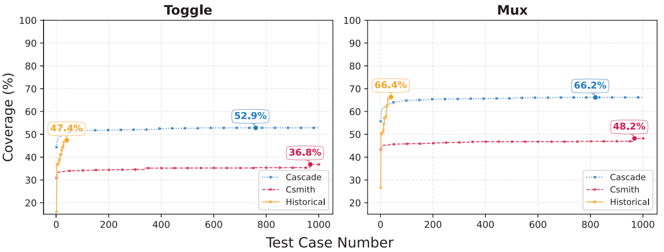
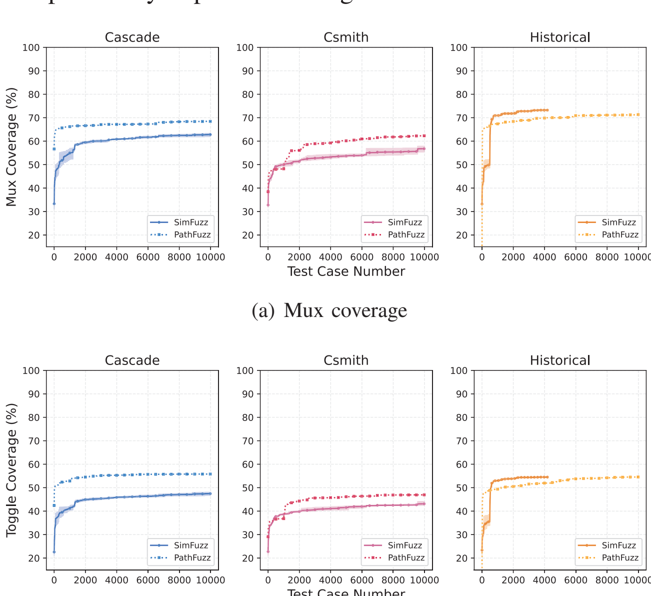


Fig. 2: Toggol and mux coverage across different seed corpora without fuzzing. The Historical seed corpus achieve a maximum mux coverage of 66.4%.

2) Corpus evaluation: We use Csmith and Cascade to generate 1,000 seeds each, along with 42 Historical seeds. Fig. 2 shows the total coverage across the three corpora. The Historical seeds achieved a maximum mux coverage of 66.4%, comparable to Cascade (66.2%) and significantly higher than Csmith (48.2%). Despite its smaller size, the Historical seeds produced strong results, highlighting the value of test cases that have previously exposed real bugs for seed construction.



(a) Mux coverage

(b) Toggle coverage

Fig. 3: SimFuzz vs. PathFuzz coverage. (a) Mux: SimFuzz peaks at 73.22% (historical). (b) Toggle: SimFuzz peaks at 54.51% (historical).

3) Design evaluation: We design an input space similarity matrix to guide the mutation process and assess whether direct exploration of the processor input space can effectively cover processor states. In our experiments, we set the similarity threshold to $T = 0.5$ based on preliminary tests. Csmith and Cascade were used to generate 1,000 seeds each. For SimFuzz, each seed from the Csmith and Cascade corpora was mutated 10 times, producing a total of 10,000 test cases. Historical seeds were mutated 100 times each, resulting in 4,200 test cases. For comparison, we evaluate SimFuzz against the state-of-the-art processor fuzzer PathFuzz [32]. PathFuzz first flattens a seed's memory access footprint into a linear sequence and then

TABLE II: Comparison of SimFuzz with PathFuzz across different seed corpora.

Seed Corpus	Coverage	SimFuzz(%)	PathFuzz(%)	Diff. (%)
Cascade	Mux	62.80 ±0.69	68.44	-5.64
	Toggle	47.46 ±0.65	55.81	-8.35
Csmith	Mux	56.78 ±1.24	62.31	-5.53
	Toggle	43.12 ±0.81	46.95	-3.83
Historic	Mux	73.22 ±0.27	71.33	+1.89
	Toggle	54.51 ±0.13	54.57	-0.06

* ± values represent half of the confidence interval range.
‡ Historic seeds: SimFuzz (4,200 tests), others (10,000 tests).

applies coverage-guided mutations. We ran PathFuzz for 10,000 executions per seed type.

Fig. 3 shows the mux and toggle coverage achieved by SimFuzz and the coverage-guided fuzzer PathFuzz after the same number of executions. Unlike PathFuzz, SimFuzz does not rely on coverage feedback, yet it achieves higher coverage on the Historical seeds. Detailed numerical results are presented in Table II. These results indicate that SimFuzz achieves coverage comparable to PathFuzz across most corpora and even slightly surpasses it on Historical seeds, demonstrating that similarity-guided exploration can effectively reach diverse processor states without relying on coverage feedback.

To further evaluate the effectiveness of the similarity-guided block-level mutation strategy, we compared SimFuzz with its variant SimFuzz*, in which similarity guidance is disabled. As shown in Fig. 4, SimFuzz consistently outperforms SimFuzz* on the Historical seeds. However, the overall coverage gap between the two remains moderate. A similar trend is observed for PathFuzz: once coverage reaches a certain level, it gradually saturates with little further improvement.

This saturation phenomenon can be attributed to two main factors. First, some processor states are inherently difficult to trigger. Second, given the limited input space and mutation strategies, most easily reachable states may already have been explored, while the remaining states often require more complex semantic perturbations or intricate inter-instruction dependencies to be triggered. Moreover, this observation suggests that coverage alone may not sufficiently characterize the extent of processor state exploration. A systematic investigation of the validity and effectiveness of coverage metrics for processor fuzzing remains an important direction for future work.

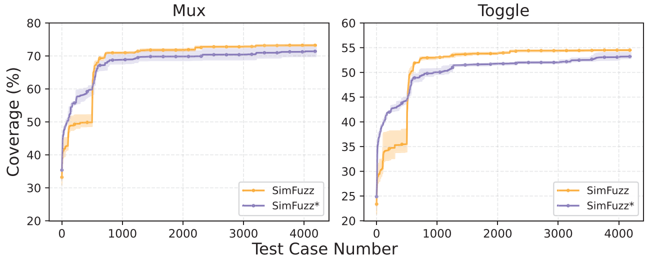


Fig. 4: SimFuzz vs. SimFuzz* on mux and toggle coverage.

TABLE III: A list of bugs discovered by SimFuzz. The "From" column indicates the seed origin: S denotes a seed from the constructed corpus, B denotes a bug discovered by SimFuzz.

Processor	Afftected Unit	From	New	Confirmed	Fixed
BOOM	**B1:** Floating	S1	✗	✗	✗
XiangShan	**B7:** Decode	S5	✓	✓	✓
	B11: Backend	NA	✓	✓	✓
	B14: Decode	B14	✓	✓	✓
	B16: Load	NA	✓	✓	✓
NEMU	**B2:** Floating	S4	✗	✗	✓
	B3: Control	S1	✓	✓	✓
	B4: Control	S6	✓	✓	✓
	B5: Decode	NA	✗	✗	✓
	B6: Memory	NA	✓	✓	✓
	B8: Decode	S5	✓	✓	✓
	B9: Control	S7	✓	✓	✓
	B10: Control	B10	✓	✓	✓
	B12: Decode	NA	✓	✓	✓
	B13: Decode	B11	✓	✓	✓
	B15: Decode	B14	✓	✓	✓
	B17: Memory	S8	✓	✓	✓

B. Bug Discovery

SimFuzz discovers a total of 17 bugs on real processors and related reference models, including 3 previously known bugs and 14 newly identified ones. All newly found bugs have been reported to the community, confirmed, and fixed.

The seed corpus consist of 42 binary test cases from 37 bug reports of three open-source processors. Rocket [2] and BOOM [24] are early open-source RISC-V processors with over a decade of development, while XiangShan [23] is a more recent processor introduced within the last five years. These three processors are commonly used as targets in verification studies [7], [8], [32]. Rocket and BOOM rely on the Spike [28] simulator as a reference model, whereas XiangShan uses NEMU [22]. The correctness of the reference model is critical, as errors in the model can mask real processor bugs. Notably, some bugs are observed in both XiangShan and NEMU, emphasizing the importance of reference model accuracy. Table III summarizes all identified bugs. Since all bugs are discovered prior to processor fabrication, no exploits currently exist that can be executed in real-world scenarios under normal user privileges. The following section provides detailed descriptions of several RTL-level bugs.

Bug B7, B8. These two bugs involve the AES encryption instruction `aes64ks1i` [26] in the XiangShan processor and NEMU simulator. The `rnum` field specifies the substitution round of the SBox [26] and is 5 bits wide (range `0-31`), but the specification allows only `0-0xA`. Both XiangShan and NEMU fail to enforce this restriction, permitting out-of-range values (e.g., `rnum = 0x15`) to execute normally. Notably, this bug is previously observed in Rocket and has reappeared in XiangShan, highlighting the difficulty of detecting certain corner cases in the RISC-V specification. SimFuzz leverages cross-processor bug data to cover such corner cases effectively.

Bug B11. This bug originates from a bug in XiangShan's back-end control logic. When a `jalr` instruction performs an indirect jump to an invalid physical address, the processor fails to trigger exception handling correctly. Specifically, if the computed target address exceeds the 48-bit physical address range, an instruction access fault should occur. However, the back-end control logic does not treat this as a valid pipeline redirection event. The valid signal generation only accounts for branch mispredictions, preventing timely pipeline flushing and proper exception handling.

Bug B13, B14. According to the RISC-V privileged specification [27], the indirect register selectors `siselect` and `vsiselect` in S-mode and VS-mode should allow addresses `0x0-0xFFF`. In XiangShan, hardware restricts this range to `0x0-0x1FF`, while NEMU imposes no limit. This discrepancy renders critical features, such as the advanced interrupt architecture, unusable on the processor, representing a severe implementation bug.

Bug B16. A bug in XiangShan's Load Unit causes incorrect values in the `mtval` register during exceptions. When a load instruction (e.g., `fld`) triggers both an address misalignment and a page table translation miss, `mtval` records an unrelated address rather than the original faulting memory access. Since the OS exception handler relies on `mtval` to handle page faults or permission violations, this error can cause incorrect fault handling and potential failure to service user-level programs.

C. Other Findings

SimFuzz revealed that processors from different open-source projects contain bugs stemming from the same underlying issues. Moreover, flaws related to known bugs persisting even in newer versions. This finding suggests that under the same ISA, different processors may share identical or similar bugs. Therefore, during processor design and verification, particular attention should be paid to known bugs to prevent the recurrence of similar errors. Expanding such corpora with additional historical bug-triggering seeds could further improve coverage and reduce verification time. We plan to enlarge the seed corpus by collecting seeds from more processors to enhance processor security and further reduce verification time.

V. Conclusion

We present SimFuzz, a novel processor fuzzing approach. Unlike traditional coverage-guided strategies, SimFuzz constructs a high-quality seed corpus using historical bug-triggering test cases from real-world processors. It then employs a similarity metric to explore the input space while preserving the control-flow structure of these seeds during mutation, enabling deeper coverage of processor states. Experimental results show that SimFuzz achieves higher coverage than coverage-guided methods, discovers 14 previously unknown bugs, and obtains 7 CVE identifiers, demonstrating both its effectiveness and practical significance. Furthermore, our results indicate that identical or similar bugs can arise across processors sharing the same architecture, highlighting the importance of incorporating real-world bug samples into processor verification.

VI. Acknowledgment

The authors would like to thank all anonymous reviewers for their valuable comments and suggestions, and the maintainers of OpenXiangShan community for their support in understanding and fixing bugs. Additionally, we acknowledge all open-source works (including but not limited to [7], [8], [18], [30], [32]) related to processor fuzzing. This paper is supported by the National Key R&D Program of China (2024YFB4506200) and the YuanTu Large Research Infrastructure.

References

[1] 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.

[2] Chips Alliance. Rocket Chip Generator. https://github.com/chipsalliance/rocket-chip.

[3] Csmith. Csmith, a random generator of c programs. https://github.com/csmith-project/csmith.

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

[5] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights into Software-Exploitable hardware bugs. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 213–230, Santa Clara, CA, August 2019. USENIX Association.

[6] Fabian Thomas et al. RISCVuzz: Discovering architectural CPU vulnerabilities via differential hardware fuzzing. https://ghostwriteattack.com/.

[7] Jaewon Hur et al. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[8] Jinyan Xu et al. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1307–1324, Anaheim, CA, August 2023. USENIX Association.

[9] Mohamadreza Rostami et al. Beyond random inputs: A novel ml-based hardware fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.

[10] Rahul Kande et al. 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, Boston, MA, August 2022. USENIX Association.

[11] Sadullah Canakci et al. 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.

[12] Timothy Trippel et al. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254, Boston, MA, August 2022. USENIX Association.

[13] Yinan Xu et al. Towards developing high performance risc-v processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022.

[14] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, CCS '22. ACM, November 2022.

[15] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[16] Shaoqian Jin, Yulin Li, Liwei Chen, and Gang Shi. SSFuzz: Generating syntactic and semantic seeds for risc-v processors. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, GLSVLSI '24, page 421–426, New York, NY, USA, 2024. ACM.

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. ACM.

[18] 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, 2018.

[19] Wu Lichao, Rostami Mohamadreza, Li Huimin, Rajendran Jeyavijayan, and Sadeghi Ahmad-Reza. GenHuzz: An efficient generative hardware fuzzer. In *34rd USENIX Security Symposium (USENIX Security 25)*. USENIX Association, August 2025.

[20] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. Reachable Coverage: Estimating saturation in fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 371–383, 2023.

[21] OpenXiangShan. Fuzzing general-purpose hardware designs with software fuzzers. https://github.com/OpenXiangShan/xfuzz.

[22] OpenXiangShan. NEMU. https://github.com/OpenXiangShan/NEMU.

[23] OpenXiangShan. XiangShan: An Open-Source High-Performance RISC-V Processor. https://xiangshan.cc/en/.

[24] RISC-V BOOM. SonicBOOM: The berkeley out-of-order machine. https://github.com/riscv-boom/riscv-boom.

[25] RISC-V International. RISC-V International Achieves Milestone with Ratification of 40 Specifications in Two Years. https://riscv.org/riscv-news/2024/04/, 2024.

[26] RISC-V International. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. RISC-V International, April 2024. Version 20240411.

[27] RISC-V International. The RISC-V Instruction Set Manual Volume II: Privileged ISA. RISC-V International, April 2024. Version 20240411.

[28] RISC-V Software. Spike: RISC-V ISA simulator. https://github.com/riscv-software-src/riscv-isa-sim.

[29] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. Exploring coverage metrics in hardware fuzzing: A comprehensive analysis. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, GLSVLSI '24, page 240–245, New York, NY, USA, 2024. ACM.

[30] 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, Philadelphia, PA, August 2024. USENIX Association.

[31] T-Head. Openc910 core. https://github.com/XUANTIE-RV/openc910.

[32] Yinan Xu, Sa Wang, Dan Tang, Ninghui Sun, and Yungang Bao. PathFuzz: Broadening fuzzing horizons with footprint memory for cpus. In *2024 61st ACM/IEEE Design Automation Conference (DAC'24)*, New York, NY, USA, 2024. ACM.