PathFuzz: Broadening Fuzzing Horizons with Footprint Memory for CPUs

Yinan Xu*
State Key Lab of Processors
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
xuyinan@ict.ac.cn

Sa Wang*
State Key Lab of Processors
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
wangsa@ict.ac.cn

Dan Tang
Beijing Institute of Open Source Chip
Beijing, China
tangdan@bosc.ac.cn

Ninghui Sun*
State Key Lab of Processors
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
snh@ict.ac.cn

Yungang Bao*
State Key Lab of Processors
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
baoyg@ict.ac.cn

ABSTRACT

Coverage metrics have been widely adopted to quantify the completeness of hardware verification. Recently, coverage-guided fuzzing has emerged as a popular method for automatically creating test inputs toward higher verification coverage reach. However, we observe that its effectiveness on CPUs is hindered by limited sources of seed corpus and efficiency of mutations. To broaden the fuzzing horizons, this paper proposes the PathFuzz framework incorporating an efficient input format for fuzzing CPUs, the footprint memory, with seed corpus from real-world large-scale programs. Experiments demonstrate that using PathFuzz reaches over 95% verification coverage with four long-standing bugs newly identified in two well-known open-source CPU designs.

CCS CONCEPTS

• **Hardware** → *Simulation and emulation*; *Coverage metrics.*

KEYWORDS

Fuzzing, Hardware Verification, Test Generation

ACM Reference Format:
Yinan Xu, Sa Wang, Dan Tang, Ninghui Sun, and Yungang Bao. 2024. PathFuzz: Broadening Fuzzing Horizons with Footprint Memory for CPUs. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655911

*Also with University of Chinese Academy of Sciences.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655911

1 INTRODUCTION

In the post-Moore's Law era, agile hardware design has garnered substantial attention and supported various CPU chips being designed and taped out. While the emerging design tools expedite the design phase and enable more complicated designs, verification remains the most time-consuming and labor-intensive stage. To quantify the verification progress, quantitative coverage metrics are commonly adopted to prune the verification space and used as an indicator for pre-silicon verification completeness [10, 15].

Recently, coverage-guided mutational fuzzing (CGF) has emerged as a popular test generation method for effectively increasing the verification coverage [3, 9, 14]. Hardware fuzzers reuse software fuzzers and follow similar workflows based on genetic algorithms. By mutating the corpus with the coverage guidance, CGF has the potential to identify unexposed functional flaws in CPUs [6–8].

While CGF is promising because of its mutation-based exploitation capability, its exploration capability when applied to CPU functional verification is limited. Due to the complicated grammar and semantic definitions of an instruction set architecture (ISA), CPUs require sophisticated instruction sequences to reach a specific privilege status. However, general fuzzers lack such domain-specific information and thus fall short in effective state exploration [2, 4, 18].

The exploration capability of CGF has two major impact factors, i.e., seed corpus and mutations. While the former corresponds to the start points of fuzzing, the latter determines how far the fuzzer could reach by continuously creating and scheduling the mutated inputs. Prior work proposes techniques to overcome the limitations, such as scenario-driven instruction templates [18], grammar- and semantic-aware mutations [7], and distance-based scheduling policies [3].

While we acknowledge their contributions in bringing in domain-specific knowledge for fuzzing CPUs, this paper focuses on an orthogonal approach to addressing the exploration issues, i.e., how to design the CPU fuzzer (in addition to the fuzzing algorithms) to expand sources of seed corpus and enable more efficient mutations. Such fundamental optimizations would benefit prior fuzzers as well as broaden the future research scope of CPU CGF.

One fundamental obstacle we identify is the input format for fuzzing CPUs. Prior CPU fuzzers use instruction sequences or memory images as the input, interpreted as a *linear memory* with contents organized according to the address order. However, this format restricts both sources of seed corpus and mutation efficiency due to its misalignment with the CPU execution. First, the input contents may have useless holes that never touched during the CPU execution but enlarge the corpus size and consume the mutation space. Second, the CPU may access memory regions absent in the input address-value pairs and not captured by the mutational fuzzer but filled with zero or random values [7].

In this paper, we propose PathFuzz, a novel CPU fuzzer aiming at broadening fuzzing horizons and enabling more sources of seed corpus and more efficient mutations. One of its key differences from prior fuzzers is the adoption of *footprint memory*. In contrast to the linear memory that tracks the input contents by addresses, the footprint memory chronologically records contents in an ordered sequence in which they are accessed during the CPU execution. This alignment with CPU dynamic execution allows PathFuzz to directly fuzz the CPU execution paths. Specifically, the format filters out useless contents, thereby accommodating diverse seeds and concentrating mutations on the most effective bytes. Additionally, this approach enhances the fuzzer's capability to handle out-of-bounds CPU memory requests by filling in input contents with mutated bytes.

Overall, our main contributions are as follows.

- We propose the novel PathFuzz fuzzer with footprint memory and expand CPU fuzzing horizons.
- To the best of our knowledge, we are the first to incorporate real-world, large-scale programs as seed corpus for fuzzing CPUs and showcase their effectiveness.
- We demonstrate the efficacy of PathFuzz by comparing the coverage increase when fuzzing with linear and footprint memory (1.28×), showcasing how a broader seed corpus could benefit the coverage reach (95.3%), and newly discovering 4 long-existing bugs in two well-known open-source RISC-V CPUs as well as tens of issues in a high-performance RISC-V CPU.

2 BACKGROUND AND MOTIVATION

CGF is an automated process of continuously creating tests to identify vulnerabilities in the design-under-test (DUT). With its growing adoption in the software community, researchers have delved into its potential for CPUs in recent years.

The CGF workflow for CPUs is as follows. Given the seed corpus, the fuzzer continuously mutates the corpus based on coverage feedback and sends mutated inputs to DUT for execution. The DUT behaviors are compared against the behaviors of a golden reference model (REF) during execution. If a mutated input leads to a behavioral mismatch, a bug is potentially detected. Otherwise, this input is added to the corpus for the next fuzzing iteration or dropped depending on whether it hits an uncovered coverage point.

Much of the existing research on CPU CGF harnesses the exploitation capability of fuzzing algorithms primarily for detecting security vulnerabilities. Given the start points (seed corpus) and searching directions (coverage guidance), the fuzzer automatically exploits the state space (applying mutations) and searches for vulnerable targets. Security-directed CGF is promising due to its capability of hitting unexpected states based on fine-tuned seed generation [8, 18], coverage guidance [7, 8, 14], mutations [2], and scheduling policies [6].

Figure 1: Coverage curve during fuzzing (mutator: havoc).

In this paper, we shift the emphasis to another crucial application of CGF: CPU functional verification. This task requires a complete and in-depth examination of CPU functionalities to verify their correctness. Therefore, in addition to the promising exploitation capability on identifying bugs, we also seek the exploration capability of CPU CGF, quantified by the coverage increase during fuzzing.

Figure 1 shows the CPU coverage curve during a ~10-hour fuzzing process with one million mutated inputs. The three sources of seed corpus S#1–S#3 provide the baseline coverage at 56.0%, 44.2%, and 72.6%, respectively. However, the coverage increase, i.e., the difference between final coverage reach and baseline coverage, is only 12.6%, 15.7%, and 9.9%, respectively. This limited coverage increase indicates limited exploration capabilities of the current CPU CGF workflow, where coverage-guided mutations do exploit some new states, but seed corpus seemingly dominates the state exploration.

The results motivate us to decouple CGF research into two orthogonal approaches. On the one hand, we should improve the fuzzing algorithms for better exploitation and exploration, such as mutations and scheduling policies. Extensive prior research has been dedicated to this area. On the other hand, there is a neglected but critical necessity to advance fuzzing infrastructures to broaden their scope, supporting a more diverse seed corpus and a wider range of fuzzing algorithms. We believe that integrating both approaches will lead to a more comprehensive CGF framework. This framework is expected to automate and enhance the functional verification of CPUs, yielding more thorough and efficient verification processes.

From the perspective of infrastructures, we investigate inherent limitations that constrain fuzzing horizons.

Constraints on seeds. Existing CPU CGF seeds are from three sources, including hand-written directed tests (such as S#1: riscv-tests and S#2: riscv-arch-test), instruction sequence generators (ISGs, such as S#3: riscv-dv), and formal tools [4]. While generating tests with formal tools usually requires design-specific knowledge, the other two tests can be easily obtained from the open-source community and have been widely adopted in most CPU projects. However, there is still one popular and high-quality type of CPU

tests not being leveraged by existing fuzzers, i.e., real-world complex programs that may require a large address space. This requirement not only risks crashing the CGF process but also compromises the efficiency of mutational fuzzers, which struggle to identify critical mutation targets within such extensive inputs. Therefore, it is necessary to improve the corpus design for supporting real-world CGF seeds.

Constraints on mutations. Mutation policies generally decide where and how the mutation applies. However, we notice that a large portion of input contents in seeds may be unused and thus hinder the effectiveness of mutations. For seeds S#1–S#3, an average of only 18.9%, 22.0%, and 6.2% of the input bytes are accessed by the DUT during execution. These fractions correlate to the coverage increase during CGF, as the lowest fraction at S#3 and the highest fraction at S#2 correspondingly contribute to the lowest and the highest coverage increase. Therefore, it is necessary to improve the CPU inputs' affinity for CGF mutations.

By looking into these constraints, we propose in this paper to review the corpus design and input format for CPU CGF. As prior work majorly focuses on algorithmic optimizations, they have been using similar input formats for CPUs, i.e., the *linear memory*. This format records contents by address–data pairs [7] or simply puts data bytes and instruction sequences in the address order [8, 18].

We observe that, despite the simplicity and general applicability of linear memory for CPUs, the correspondence between address and data results in underutilization and overutilization. Underutilization happens when some inputs are unused, causing a large seed corpus and impeding the mutation efficiency. Conversely, overutilization arises when the CPU attempts to access non-existent addresses in the inputs, where prior fuzzers could only inject random values [7] and fail to take advantage of coverage-guided mutations.

Underutilization and overutilization present significant challenges in optimizing fuzzing processes, highlighting the need for efficient utilization of input data in CPU fuzzing. Therefore, this paper proposes a novel CPU fuzzer, PathFuzz, with an input format of *footprint memory*, which avoids the static correspondence between address and data and interprets the input bytes dynamically during CPU execution.

3 DESIGN

PathFuzz pursues fuzzing the CPU execution paths via the footprint memory with expanded horizons in seeds and mutations. Figure 2 presents an overview of the PathFuzz workflow with three major phases: seed corpus generation, coverage-guided fuzzing, and co-simulation.

Recalling the motivation of advancing infrastructures to broaden the fuzzing scope for CPUs, we design the PathFuzz framework with two primary goals. First, it adopts the novel footprint memory (Section 3.1) to enable more efficient exploration of CPU execution paths by richer seeds and better format affinity for mutations. Second, while incorporating the new input format, it is built with the state-of-the-art modular software fuzzer LibAFL [5] and RISC-V CPU co-simulation framework DiffTest [19]. These open-source and reusable tools will promote PathFuzz as a shared and extensible platform for future CPU CGF research (Section 3.2).

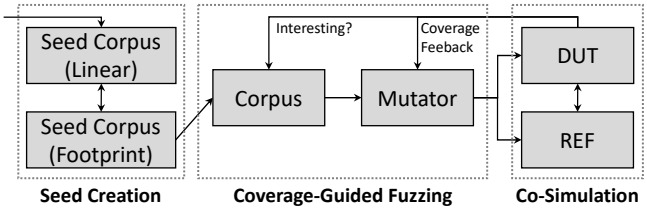

Figure 2: Overview of the proposed PathFuzz workflow.

Based on PathFuzz, we demonstrate how to further leverage RISC-V architectural checkpoints [19] to obtain CGF seeds from real-world, large-scale programs (Section 3.3).

3.1 Footprint Memory

As widely used by existing CPU fuzzers, the linear memory contains bytes indexed within a single contiguous address space and accessed by the CPU using addresses. To alleviate the underutilization and overutilization issues, we propose the footprint memory with bytes arranged in chronological order of CPU execution and memory accesses.

We explain the definition of footprint memory with the conversion from a linear memory. Given linear inputs as the initial image for the CPU's memory, the CPU generally fetches and executes instructions from the memory. The instructions may also access the memory via explicit load/store operations or implicit TLB accesses. During the simulation, the bytes being first-time accessed for every unique address are chronologically recorded. After that, these ordered bytes are concatenated to derive the converted footprint memory. Thus, the footprint format is simply the memory footprints of a CPU aligned with the CPU execution paths.

Conversely, it becomes straightforward to leverage footprint inputs for CPU execution. When an address is first-time accessed with a request size of N, we read the footprint memory for N bytes and return them as the response data.

Overall, the insight of using footprint memory for CGF is fuzzing the CPU execution paths directly via the CPU memory footprints instead of the whole memory image. It alleviates the underutilization and overutilization issues by eliminating useless portions at generation time and interpreting the contents dynamically during CPU execution.

3.2 Fuzzing CPUs via Footprint Memory

Though the footprint memory is promising in fuzzing the CPU execution paths, leveraging it for fuzzing CPUs requires some adaptions to the existing CGF workflow.

First, the seed corpus may be accumulated during a continuous fuzzing process and reused by various fuzzing iterations on multiple designs. However, the interpretation of footprint seeds relies on the specific CPU behaviors. Therefore, to maintain consistent semantic meanings for the test inputs, PathFuzz has a format conversion stage before and after each fuzzing iteration. The seeds are stored in the linear format but used by CGF in the footprint format.

Second, to integrate CPU designs into software fuzzers, PathFuzz designs standardized coverage collection interfaces between

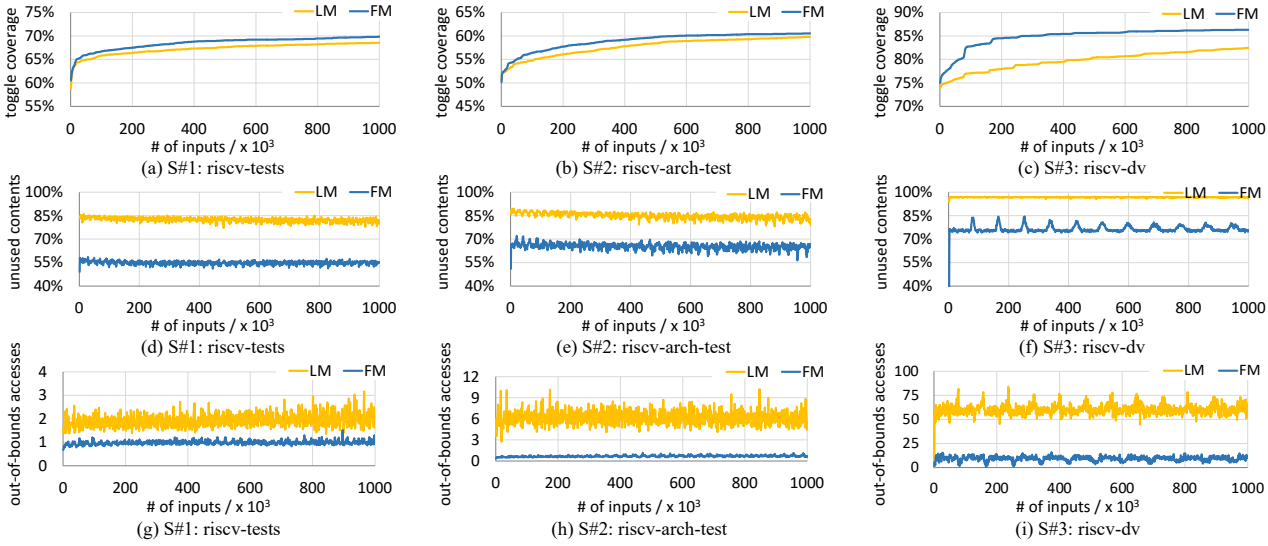

Figure 3: Coverage, fraction of unused input contents, and number of out-of-bounds accesses per input during fuzzing.

RTL-simulation and the fuzzer, supporting three types of CPU coverage metrics. Similar to prior work, PathFuzz accepts 1) hardware instrumentation for collecting structural and functional coverage, 2) software instrumentation on the simulator-generated C++ model, and 3) coverage feedback from the golden REF, which is a good indicator of how many ISA-level behaviors have been reached.

Third, the prior co-simulation methodology is incompatible with the footprint memory. For example, DiffTest copies the DUT's memory image to REF at reset to ensure they receive identical inputs with the same memory contents at any memory address. However, the semantic meaning of a footprint memory depends on the DUT's microarchitectural behaviors, i.e., the correspondence between input bytes and addresses can only be determined during CPU execution. Therefore, PathFuzz proposes an on-demand memory synchronization mechanism between the DUT and the REF. Since the REF does not speculatively execute any instruction, and its execution always falls behind the DUT, PathFuzz delays the memory copying from the initialization phase to when the input bytes are first-time accessed by the DUT. This copy-on-demand mechanism also improves minor performance by avoid copying untouched inputs to the REF.

3.3 Enriching Seed Corpus

To enrich the CGF corpus, PathFuzz selects arbitrary program phases during CPU execution and captures memory footprints as seeds corresponding to the execution paths.

Incorporating real-world, large-scale programs as seeds in CGF is challenging due to time and space constraints. First, these programs typically have long execution times, which conflicts with the rapid iteration cycle required for effective coverage feedback and mutations. Additionally, the substantial linear memory space these programs use hinders the effectiveness and efficiency of CGF.

PathFuzz proposes leveraging the RISC-V architectural checkpoints [19] with footprint memory to overcome time and space restrictions. Checkpoints are snapshots of program states at specific moments, enabling the instant resumption of program execution from these exact points. These linear checkpoints are then transformed into the footprint memory, thereby reducing their size to be used as seeds.

Table 1: Evaluation setup.

Feature	Configuration
DUT, REF	Rocket, Spike
Target Coverage	toggle
Seed Corpus, #Cases	S#1: riscv-tests, 140 (LM, FM) S#2: riscv-arch-test, 257 (LM, FM) S#3: riscv-dv, 1150 (LM, FM) S#4: force-riscv, 969 (FM) S#5: SPEC CPU2006, 55 (FM) S#6: SPEC CPU2006, 1090 (FM)

4 EVALUATION

In this section, we evaluate the proposed PathFuzz CPU fuzzer. Our results are highlighted as follows.

- Exp.#1: Footprint memory (FM) facilitates better coverage increases, up to 1.28× over linear memory (LM).
- Exp.#2: Incorporating real-world seeds could reach a higher final coverage, e.g., 95.3% with SPEC CPU2006.
- Exp.#3: PathFuzz finds 4 long-standing bugs in Rocket and Spike, along with additional bugs in other CPUs.

Setup. As shown in Table 1, we majorly fuzz the Rocket RISC-V CPU [1], a well-known and time-honored design, with Spike [13], a widely-adopted RISC-V golden REF for co-simulation. Since both CPUs have been extensively used by open-source projects and prior fuzzing research, it is challenging to find bugs in their designs.

The seed corpus used for evaluation also comes from open-source projects, including hand-written directed tests S#1 and S#2, ISGs S#3

Table 2: Uncovered bugs and the detection probability, detection time, and number of executed mutated inputs.

Bug ID	S#1 (LM/FM)	S#2 (LM/FM)	S#3 (LM)	S#3 (FM)	S#4 (FM)	S#5 (FM)
B#1 `281e5c8`	0 (0.00%)	0 (0.00%)	6 (37.50%) 0h-17m-43s, 13947	11 (68.75%) 0h-3m-50s, 1417	0 (0.00%)	16 (100.00%) 0h-0m-12s, 68
B#2 `7533edb`	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	11 (68.75%) 0h-2m-52s, 1543	0 (0.00%)
B#3 `0320cfb`	0 (0.00%)	0 (0.00%)	0 (0.00%)	4 (25.00%) 0h-4m-9s, 1998	15 (93.75%) 0h-3m-15s, 2512	15 (93.75%) 0h-1m-35s, 1932
B#4 `93aad1d`	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	5 (31.25%) 0h-14m-26s, 19624	1 (6.25%) 1h-26m-49s, 91527

and S#4, and real-world CPU benchmarks S#5 and S#6. They have been widely adopted by the RISC-V community for CPU verification. Since S#4–S#6 binaries are too large in LM for fuzzing, they are used only in FM. While S#5 contains only one test case for each SPEC CPU2006 benchmark, S#6 is a superset of S#5.

Unless otherwise specified, each CGF process runs for one million mutated inputs. We repeat each configuration 16 times, averaging the coverage data for analysis. Given page limits, we present solely the CGF results with toggle coverage feedback from Rocket. However, our principal findings are also applicable to a broader range of coverage metrics, additional CPU models, and varied seeds.

4.1 Exp.#1: Footprint Memory

PathFuzz adopts FM to fuzz CPU execution paths, expecting better mutation affinity and more coverage increases.

Enhanced coverage increases in less time. As illustrated in Figure 3(a)–(c), seeds S#1–S#3 are converted to FM and used as CGF seeds, with coverage increases of 13.8%, 16.5%, and 12.7% over the seeds' coverage. These increases are 1.10×, 1.05×, and 1.28× compared with LM, demonstrating better format affinity of FM for CGF. Additionally, since fuzzing is a time-bounded process, we assess the relative efficiency of FM in achieving the same coverage as LM. Specifically, the speedups are 2.79×, 1.96×, and 11.90×, underscoring the superior efficiency of FM compared with LM.

Underutilization and overutilization. We now quantify these two indicators of mutation inefficiency to explain why FM facilitates the state exploration of CGF. The results are shown in Figure 3(d)–(i). Specifically, both unused inputs and out-of-bounds memory requests are significantly reduced in FM. Using S#3 as a representative example, we notice that an average of 96.8% of mutated LM input bytes are useless, and the mutated cases lead to an average of 60.3 out-of-bounds CPU memory requests. By contrast, using FM effectively curtails the numbers to 76.3% and 9.3, respectively, thus alleviating the underutilization and overutilization.

4.2 Exp.#2: Broadening the Seed Corpus

PathFuzz, for the first time, accepts real-world programs as CGF seeds by the footprint memory. We now showcase how these expanded sources of seeds boost the coverage reach in CPU CGF.

SPEC CPU2006 is a popular large-scale CPU benchmark stressing the CPU microarchitecture. After one million mutated inputs with seeds S#5 and S#6, the fuzzer reaches the final coverage at 93.1% and 95.3%, respectively. These results suggest that, by broadening seeds, PathFuzz could quickly meet the criteria of 90% coverage that a prior formal-assisted fuzzer struggles to achieve [4].

4.3 Exp.#3: Finding Bugs

While evaluating PathFuzz, we discover three bugs (B#1–B#3) in Rocket and one (B#4) in Spike. As shown in Table 2, these bugs are detected across various fuzzing seeds with different probabilities and detection times.

Effectiveness. While bugs B#1–B#4 have existed in open-source projects since March 2017, December 2021, December 2021, and December 2014, respectively, they remained undetected in extensive prior fuzzing studies that employ these projects as benchmarks [7, 8, 18]. In contrast, PathFuzz identifies them in under 15 minutes in the best case. Additionally, we leverage PathFuzz to verify the other two designs supported by DiffTest. The fuzzer identifies tens of unique issues in the in-order NutShell CPU [17] as well as the out-of-order high-performance XiangShan CPU [19]. We are working with their vendors to confirm the actual bugs.

Effectiveness of footprint memory. Compared with S#3 (LM), using S#3 (FM) exclusively uncovers B#3 and detects B#1 more rapidly and more likely.

Effectiveness of seed selection. One notable result is that using S#4 and S#5 exclusively detects B#2 and B#4 as well as identifies B#1 and B#3 much faster. We delve into further details to understand why this occurs.

B#2 is due to the missing `mconfigptr` control and status register (CSR) in Rocket. It would be challenging to mutate an arbitrary instruction to read this specific CSR among the 4096 CSRs. However, while `mconfigptr` has an address of `0xF15`, the first instruction of S#4 is a read to the `mhartid` CSR at `0xF14`. Thus, this address adjacency gives S#4 a unique advantage in identifying B#2. Similarly, S#4 starts executing ISG-generated instructions by loading their start address into `s7` followed by a `jr s7` instruction. This pattern fortunately increases the possibility of manifesting B#4 that depends on a cross-boundary instruction fetch.

Besides, manifesting B#1 requires a page-table entry (PTE) with an out-of-bounds physical page number (PPN), causing an incorrect page-fault exception raised by Rocket rather than an access-fault exception. Since S#3 enables virtual memory with valid PTEs, fuzzing with S#3 seeds highly likely detects B#1. Hopefully, SPEC CPU2006

benchmarks traverse a larger memory region with more PTEs involved in CPU execution. Thus, S#5 requires far fewer mutated inputs than S#3 to identify bug B#1 with a higher probability.

To summarize, we have seen a strong correlation between bug detection and seed selection of CGF, underscoring the importance of expanding sources of seed corpus for fuzzing CPUs. Together with the coverage results, we provide clear evidence of the motivation and design of PathFuzz, i.e., to broaden fuzzing horizons towards better exploration capabilities of CGF for CPU verification.

5 RELATED AND FUTURE WORK

Fuzzing is a sophisticated process with lots of impact factors on its outcomes. Extensive prior research has covered a wide range of topics, such as formal assistance [4], coverage feedback [7, 8, 14], CPU-directed mutations [2, 18], and FPGA acceleration [9, 12]. We believe PathFuzz complements most of them and capitalizes on their advancements.

One similar approach to fuzzing CPU execution paths is proposed by [2, 18]. They design an instruction morpher and a translation buffer to mutate instruction footprints at the CPU decoder and fuzz the instructions going to be executed. Compared with them, PathFuzz proposes a generic footprint format capturing all CPU memory footprints. It supports broader fuzzing capabilities for non-instruction footprints, such as PTEs required for manifesting B#1.

PathFuzz illuminates one promising future research path of the strategic seed selection for improving coverage and detecting bugs [11]. For example, while S#5 achieves the final coverage at 93.1% after one million inputs and 14 hours, S#6 reaches the same coverage in a mere 12.5 minutes with fewer than nine thousand mutated inputs, and this remarkable 67× speedup is attributed by just selecting additional sequences from SPEC CPU2006 benchmarks. Another future work of PathFuzz, as an open-source CGF framework for efficient CPU verification, is to accelerate its execution by improving the RTL-simulation speed [9] or leveraging AI methods for accurate predictions of the execution results [16].

6 CONCLUSION

In this paper, we presents PathFuzz, a novel fuzzer for CPU verification. PathFuzz adopts the footprint memory for expanding fuzzing horizons with broader seeds and more efficient mutations. We incorporate real-world large-scale programs as the seed corpus and showcase their effectiveness in improving verification coverage and discovering functional bugs. PathFuzz is demonstrated to reach 95% verification coverage with 4 long-standing bugs being newly discovered. PathFuzz has been open-sourced[1] and is expected to be a reusable platform for future CPU fuzzing research.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (Grant No. 62090022, 62090023, and 62172388), the Strategic Priority Research Program of Chinese Academy of Sciences under grant number XDA0320000 and XDA0320300, and Youth Innovation Promotion Association of Chinese Academy of Sciences (2020105).

[1] Available at https://github.com/OpenXiangShan/xfuzz.

REFERENCES

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[2] Niklas Bruns, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2022. Efficient Cross-Level Processor Verification Using Coverage-Guided Fuzzing. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22).* Association for Computing Machinery, New York, NY, USA, 97–103.

[3] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. DirectFuzz: Automated Test Generation for RTL Designs Using Directed Graybox Fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE Press, 529–534.

[4] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. HyPFuzz: Formal-Assisted Processor Fuzzing. In *32nd USENIX Security Symposium.* 1361–1378.

[5] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22).* Association for Computing Machinery, 1051–1065.

[6] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. SoCFuzzer: SoC Vulnerability Detection using Cost Function enabled Fuzz Testing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE).* 1–6.

[7] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP).* 1286–1303.

[8] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium.* 3219–3236.

[9] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 1–8.

[10] Daniel Lewin, Dean Lorenz, and Shmuel Ur. 1996. A Methodology for Processor Implementation Verification. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD '96).* Springer, 126–142.

[11] Rongjian Liang, Nathaniel Pinckney, Yuji Chai, Haoxin Ren, and Brucek Khailany. 2023. Late Breaking Results: Test Selection For RTL Coverage By Unsupervised Learning From Fast Functional Simulation. In *2023 60th ACM/IEEE Design Automation Conference (DAC).* 1–2.

[12] Hany Ragab, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer. In *Fourth Workshop on the Security of Software/Hardware Interfaces.*

[13] riscv-software-src. [n. d.]. *Spike, a RISC-V ISA Simulator.* https://github.com/riscv-software-src/riscv-isa-sim

[14] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *31st USENIX Security Symposium (USENIX Security 22).* USENIX Association, 3237–3254.

[15] Shmuel Ur and Yaov Yadin. 1999. Micro Architecture Coverage Directed Generation of Test Programs. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99).* Association for Computing Machinery, 175–180.

[16] Shobha Vasudevan, Wenjie (Joe) Jiang, David Bieber, Rishabh Singh, hamid shojaei, C. Richard Ho, and Charles Sutton. 2021. Learning Semantic Representations to Verify Hardware Designs. In *Advances in Neural Information Processing Systems*, Vol. 34. Curran Associates, Inc., 23491–23504.

[17] Huaqiang Wang, Zifei Zhang, Yue Jin, Linjuan Zhang, and Kaifan Wang. 2020. NutShell: A Linux-Compatible RISC-V Processor Designed by Undergraduates. In *RISC-V Global Forum 2020* (Virtual Event). RISC-V International.

[18] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation. In *32nd USENIX Security Symposium.* 1307–1324.

[19] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1178–1199.