HWFuzz: An FPGA-Accelerated Fuzzing Framework for Efficient RISC-V Verification

Yang Zhong[1,2], Haoran Wu[3]*, Yungang Bao[1,2] and Kan Shi[1,2]

[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[3]University of Cambridge

Abstract

We introduce HWFuzz, a high-performance fuzzing-based verification framework that automatically detects potential vulnerabilities in RISC-V processors. The framework tackles the challenge of lengthy verification cycles required to achieve hard-to-reach coverage points, particularly in complex processors. Our solution enhances verification efficiency through two key components. First, we implement a hardware fuzzer that rapidly generates large-scale RISC-V instruction sequences. Instead of simply using random instructions, the fuzzing-based instructions are specifically designed to achieve better coverage while exploring hard-to-reach states during RISC-V processor verification. Second, we develop an end-to-end verification framework that iteratively generates coverage-directed stimuli using the hardware fuzzer, applies them to the design-under-test (DUT), collects coverage data, and identifies uncovered states for further fuzzing iterations. Through FPGA acceleration, our framework achieves 2.03× *speedup against conventional software approaches.*

Introduction

Fuzzing has traditionally served as software testing tools [1, 2, 3] that identify security vulnerabilities and bugs by generating large volumes of random and often invalid input data to trigger unexpected behavior. Recently, this approach has been adapted to verify hardware circuits [4], with modifications to enhance verification effectiveness. For example, several fuzzing methods have emerged specifically for processor verification. To our knowledge, RFuzz[5] was the first tool to apply fuzzing methods to RTL functional verification, but only limited to small-scale design. DifuzzRTL[6] brought improved fuzzing performance and more efficient coverage exploration, introducing a cycle-accurate, scalable coverage metric designed specifically for RTL design. Other research such as TheHuzz [7] and Cascade [8] have since developed innovative methods to optimize instruction generation and enhance overall efficiency.

However, existing hardware fuzzing techniques still require the fuzzer to run in software, creating a performance bottleneck that results in slower stimulus generation and a longer verification period.

To address the above issue, we propose HWFuzz, an FPGA-accelerated fuzzing framework for RISC-V processor verification. Our work makes three novel contributions: First, we developed a synthesizable and highly configurable hardware fuzzer IP that enables rapid generation of RISC-V coverage-directed instruction stimuli. Second, instead of simply using random instructions, we enhanced the fuzzing algorithm to improve stimulus quality, maximizing executable instructions and achieving better coverage while exploring corner cases. Third, we fully automated the hardware verification process. The system iteratively collects coverage data from the DUT to enhance stimulus quality while automatically detecting potential vulnerabilities by comparing the execution results from both the DUT and its software reference model based on the ENCORE framework with FPGA acceleration.

*Haoran Wu finish this work during their internship at Institute of Computing Technology, Chinese Academy of Sciences.

The HWFuzz Framework

Overview The overall workflow of the HWFuzz framework is shown in Figure 1. We take advantage of the Zynq UltraScale+ architecture that consists of Programmable Logic (PL) and Processing System (PS) on the same FPGA to implement and accelerate the overall verification process. Specifically, PL is used to deploy the hardware fuzzer IP and the RISC-V processor DUT. The corresponding software model of the DUT is running simultaneously on the PS. We reuse the differential checking capability from ENCORE to automatically compare the execution results from DUT and the reference model dynamically.

HWFuzz can automatically instrument fine-grained synthesizable coverpoints into the DUT during compilation. This allows coverage information to be collected directly from the FPGA during runtime, which feeds back to the fuzzer to improve stimulus generation.

Coverage-Directed Generation The hardware fuzzer IP operates in two modes: random and mutation. The random mode selects instructions purely at random, while the mutation mode modifies the previously generated stimuli in the random mode by adjusting their operands and context. During each fuzzing iteration, the system stores the generated stimuli and their corresponding DUT coverage data collected in the hardware in a corpus. The mutation

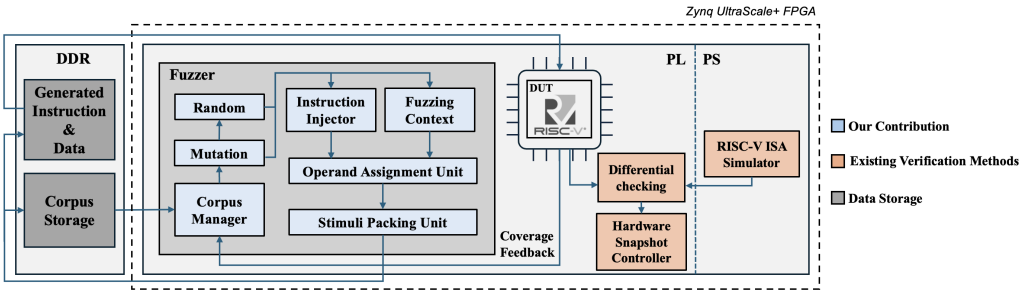

Figure 1: *The HWFuzz architecture.*

mode then selectively modifies the most promising stimuli—those with the highest probability of maximizing coverage. This targeted approach increases the chances of generating instructions that expand DUT coverage and enhance verification effectiveness.

Stimuli Packing The fuzzer generates two types of stimuli: instructions and data. Both are stored in a designated DDR region that serves as the testing environment for DUT verification. While data generation follows a simple random approach—creating random data values that Load/Store instructions can interact with—instruction generation involves a more sophisticated, multi-stage process, as shown in Figure 1.

In the initial stage, Random and Mutation generate a set of instructions with opcodes only. Next, the system automatically creates a context for these instructions by adding address-related information. Meanwhile, the instruction injector works in parallel to insert auxiliary instructions that provide register values, improving verification effectiveness. Finally, a processing unit assigns operands to each instruction based on its generated context.

Stimuli Constraints Beyond performance issues, previous fuzzing methods face another key limitation that many generated instructions cannot be fully executed. Control flow instructions comprise a significant portion of generated instructions, particularly when instruction set extensions are not fully enabled. Without proper jump range restrictions, many of these instructions miss execution opportunities, as Figure 2 illustrates. An unrestricted jump range fails to improve coverage. While Cascade addressed this through a software-based approach of reconstructing basic blocks, we opted for implementing hardware constraints on control flow instruction jump ranges. This method significantly increases the proportion of instructions that can be executed.

Evaluation and Results

We implemented HWFuzz on a Fidus Sidewinder board equipped with an AMD Zynq UltraScale+ XCZU19EG FPGA. We used Rocket Core—a 64-bit RISC-V single-issue processor as the DUT.

We configure HWFuzz to support RV64I and RV64G extensions, whereas the software fuzzer supports

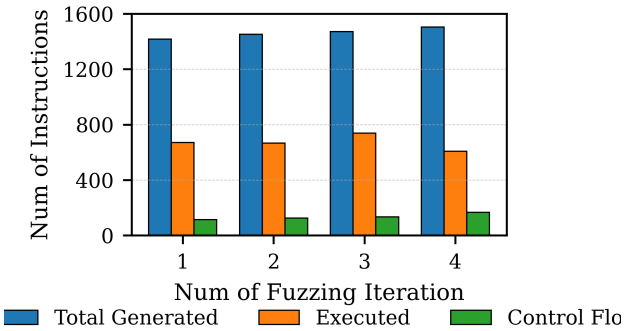

Figure 2: *The comparison of instruction counts for three instruction types*

RV64G ISA. As seen in Figure 3, HWFuzz with G extension achieves 2.03× coverage growth over software fuzzer after 100 fuzzing iterations, while also demonstrating a more significant increase in coverage over time. Meanwhile, HWFuzz with only the I extension reaches more than half the coverage points of the software fuzzer with the G extension.

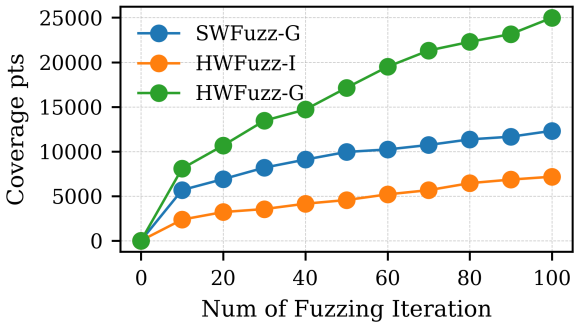

Figure 3: *Coverage comparison*

Table 1 shows the resource utilization. Compared to an instrumented RISC-V core, the fuzzer IP incurs a small area overhead.

Table 1: *Resource Usages*

Resource	Fuzzer	Infra	DUT
LUTs	68377 (13.08%)	8217 (1.57%)	209865 (40.15%)
Block RAMs	192.5 (19.56%)	325 (33.03%)	20 (2.03%)
Registers	92407 (8.84%)	12749 (1.22%)	118683 (11.35%)

References

[1] Michał Zalewski. *American Fuzzy Lop - Whitepaper*. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2025-04-12. 2016.

[2] Ella Bounimova, Patrice Godefroid, and David Molnar. "Billions and billions of constraints: Whitebox fuzz testing in production". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 122–131. DOI: 10.1109/ICSE.2013.6606558.

[3] A. Geralis. *Libfuzzer*. 2023. URL: https://github.com/planetis-m/libfuzzer (visited on 11/18/2023).

[4] 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. Clearwater, FL, USA: Association for Computing Machinery, 2024, pp. 192–197. ISBN: 9798400706059. DOI: 10.1145/3649476.3658697. URL: https://doi.org/10.1145/3649476.3658697.

[5] Kevin Laeufer et al. "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, pp. 1–8. DOI: 10.1145/3240765.3240842.

[6] Jaewon Hur et al. "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1286–1303. DOI: 10.1109/SP40001.2021.00103.

[7] 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)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3219–3236. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/kande.

[8] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. "Cascade: CPU Fuzzing via Intricate Program Generation". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5341–5358. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/solt.