# ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance

Sadullah Canakci[1], Chathura Rajapaksha[1], Leila Delshadtehrani[1], Anoop Nataraja[2],
Michael Bedford Taylor[2], Manuel Egele[1], and Ajay Joshi[1]
[1]Department of ECE, Boston University
[2]Department of ECE, University of Washington
{scanakci, chath, delshad, megele, joshi}@bu.edu, mysanoop@uw.edu, {prof.taylor}@gmail.com

2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 979-8-3503-0062-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/HOST55118.2023.10133714

***Abstract*—As the complexity of modern processors has increased over the years, developing effective verification strategies to identify bugs prior to manufacturing has become critical. Inspired by software fuzzing, a technique commonly used for software testing, multiple recent works use hardware fuzzing for the verification of Register-Transfer Level (RTL) designs. However, these works suffer from several limitations such as lack of support for widely-used Hardware Description Languages (HDLs) and misleading coverage-signals that misidentify "interesting" inputs. Towards overcoming these shortcomings, we present ProcessorFuzz, a processor fuzzer that guides the fuzzer with a novel *CSR-transition coverage* metric. ProcessorFuzz monitors the transitions in Control and Status Registers (CSRs) as CSRs are in charge of controlling and holding the state of the processor. Therefore, transitions in CSRs indicate a new processor state, and guiding the fuzzer based on this feedback enables ProcessorFuzz to explore new processor states. We evaluated ProcessorFuzz with three real-world open-source processors – Rocket, BOOM, and BlackParrot. ProcessorFuzz triggered a set of ground-truth bugs 1.23× faster (on average) than DIFUZZRTL. Moreover, our experiments exposed 8 new bugs across the three RISC-V cores and one new bug in a reference model. All nine bugs were confirmed by the developers of the corresponding projects.**

***Index Terms*—greybox fuzzing, processor, coverage, RTL verification, RISC-V**

## I. Introduction

As the complexity of processor designs has continuously grown over the years, verification has become one of the most challenging tasks in processor manufacturing. The state-space of a complex processor is extremely large, while the processor vendors have limited time and resources for verification. An exhaustive verification (i.e., testing each and every scenario) is an unrealistic goal to achieve, and therefore, a high-quality verification methodology is essential to discover bugs before fabrication. A timely, pre-silicon bug discovery can circumvent potentially millions-of-dollars of losses [25]. Otherwise, undiscovered bugs can manifest as severe functional and security holes in both proprietary and open-source processors [11], [13], [26], [28], [32], [58], [61].

Dynamic verification techniques [4], [14], [16], [47], [55], [59] are commonly used as part of the processor verification process. Dynamic verification involves simulating a Design Under Test (DUT) with a test input and analyzing the behavior of the DUT during or after simulation to identify bugs. Recent works [24], [33], [57] demonstrate that Coverage-based Greybox Fuzzing (CGF), a widely-used software testing technique, can be adapted as a dynamic verification technique to identify bugs in a processor design if certain differences between hardware and software are addressed.

Prior works on processor fuzzing mainly focus on addressing **two major challenges.** First, code coverage metrics used for fuzzing software programs (basic block, branch coverage, etc.) are not well-suited for fuzzing hardware [24], [55]. Second, a bug in a processor design does not result in an observable anomaly (i.e., crash) during testing as opposed to many software programs which can indicate the presence of bugs by throwing memory violation errors or raising exceptions.

**To address the first challenge**, researchers have introduced a variety of coverage metrics [24], [33], [35], [45] such as multiplexer toggle coverage and register coverage that are tailored for hardware. In the context of a processor, the processor is effectively a complex Finite State Machine (FSM) that consists of a large number of states. Exploring different states in 'processor FSM' is the key to identifying bugs in the processor. Therefore, hardware-specific coverage metrics mainly aim to guide the fuzzer towards different uncovered 'processor FSM' states. These metrics take the hardware intrinsic (e.g., wire connections) into account rather than merely the code structure of the hardware. For instance, DIFUZZRTL [24], a state-of-the-art processor fuzzer, introduces *register coverage* metric where the goal is to monitor value changes in registers that control multiplexer selection signals. The intuition is that a particular value in these registers represents a unique state in the 'processor FSM' and guiding the fuzzer based on this feedback explores additional FSM states.

DIFUZZRTL's register coverage metric improves on prior works [1], [33], [44] in terms of scalability, efficiency, and precision. However, we make a key observation that the register coverage can be a highly misleading metric for a processor fuzzer. Specifically, we find that DIFUZZRTL monitors many datapath registers which have minimal control over the current FSM state of the processor. The coverage increase resulting from the datapath registers does not provide meaningful information related to the current FSM state of the processor. This results in a scenario where inputs that affect datapath register coverage are incorrectly being classified as 'interesting' inputs, which in turn leads to wasted fuzzing time.

**To address the second challenge**, existing processor fuzzers [20], [24], [29], [34] adapt differential testing from the software

domain to the hardware domain. Differential testing in software compares outputs of multiple programs that have the same functional behavior and checks for inconsistencies. In the hardware domain, the results of an Register Transfer Level (RTL) simulator are compared with those of an Instruction Set Architecture (ISA) simulator. An RTL simulator is used to simulate the execution of an instruction stream on the detailed microarchitecture implementation of the processor. The ISA simulator is used to simulate the functional behavior of the processor design and used as a reference model. A difference in the execution output of RTL simulation and ISA simulation indicates a potential bug in the processor.

In this work, we present ProcessorFuzz, a processor fuzzer that implements two novel features. First, ProcessorFuzz uses a new coverage metric called *CSR-transition coverage* to effectively guide processor fuzzing towards exploring unique processor states. Specifically, it monitors transitions in Control and Status Registers (CSRs) that form the core of the architecture specifications. Our intuition is that certain CSRs dictated by ISA readily expose the current 'processor FSM' state (e.g., current privilege mode, the event that caused floating point mode exception), and thus the transitions in these CSRs signify a new 'processor FSM' state.

ProcessorFuzz's second feature is that it uses ISA simulation to rapidly determine if a test input is interesting. Prior works rely on RTL simulation for the same goal, which is time-consuming. In fact, this problem gets compounded if the coverage guidance is misleading and results in the execution of repetitive test inputs. ISA simulation is significantly faster than RTL simulation[1]. Hence, ProcessorFuzz can efficiently eliminate repetitive test inputs and focus on as many qualitatively distinct test input patterns as possible to expose bugs faster.

We evaluate ProcessorFuzz using a variety of widely-used open-source RISC-V based processors [2], [9], [48] designed in different HDLs (i.e., Chisel and SystemVerilog). These processors vary in microarchitectural implementations such as their pipeline depths, execution type (i.e., in-order and out-of-order execution), etc. We compare the bug-finding effectiveness of ProcessorFuzz against the state-of-the-art register coverage guided DIFUZZRTL. On average, for the bugs found by DIFUZZRTL, ProcessorFuzz triggers bugs 1.23× faster than DIFUZZRTL. In addition, ProcessorFuzz revealed 8 new bugs in widely-used open-source processors and one new bug in a reference model.

In summary, we make the following contributions:

- We propose ProcessorFuzz, a new processor fuzzing mechanism. ProcessorFuzz uses a novel *CSR-transition coverage (CTC)* metric, to effectively guide processor fuzzing towards interesting processor states.
- We propose to use the ISA simulator as part of a coverage feedback mechanism to rapidly identify interesting test inputs, thereby accelerating the bug-finding process.
- We demonstrate the practicality of ProcessorFuzz using 3 different open-sourced RISC-V processors and present eight new bugs identified in those three different processor designs and one new bug in a reference model.

[1]As a reference point, ISA simulation is 79× faster than RTL simulation for the open-source RISC-V based BOOM [9] processor.

## II. Background and Motivation

In this section, we first briefly explain coverage-based greybox fuzzing (CGF) for software. Next, we provide a brief background of how CGF is adapted as a hardware fuzzing method (specifically for processor fuzzing) and present the motivation of our work.

### A. Coverage-based Greybox Fuzzing

Fuzzing has gained broad adoption in the software community due to its effectiveness in bug discovery, scalability, and practicality [18], [19], [41]. Fuzzing is the process of repeatedly running a Program Under Test (PUT) with a large number of random inputs to discover bugs in software. One of the widely-used fuzzing variants is CGF which utilizes the coverage feedback collected from the PUT at runtime. In each run of the PUT, CGF records coverage (e.g., basic block coverage, edge coverage, etc.) to determine if the input is 'interesting', i.e., whether it leads to increased coverage. If so, CGF applies a set of mutations to the 'interesting' input to generate new inputs which are then fed to the PUT in the next fuzzing rounds. Here, the intuition is that generating new inputs from coverage increasing ones would cover even more unexplored code. CGF instruments the code of the program (either statically or dynamically) with the necessary book-keeping logic to record coverage during the program execution.

### B. Adapting CGF for Processor Fuzzing

Recent works [24], [33], [57] show that CGF can be adapted as a dynamic verification method for hardware including processors. In this section, we briefly explain two important aspects when adapting CGF to processor fuzzing.

**Hardware Execution.** In the case of CGF for software, the fuzzing target is a software program that can be directly executed on a host machine with a test input after compilation. However, hardware (e.g., a processor) is not directly executable on the host machine. A hardware design is implemented with an RTL abstraction and simulated with an RTL simulator to evaluate a test input. The RTL design is usually expressed with an HDL (e.g., Verilog, VHDL).

**Bug Detection.** Most software fuzzers focus on bugs that manifest as memory safety violations such as segmentation faults. These types of bugs are relatively easy to detect because they cause an observable anomaly (i.e., crash) in program behavior. However, fuzzing to find semantic bugs (e.g., logic errors) is harder than discovering memory violations because defining semantic violations is a highly domain-specific task. For these types of bugs, researchers proposed differential testing [5], [40], [42], [50] that compares the output of multiple programs with the same functionality and checks for inconsistent behaviors. This approach is used by processor fuzzers [20], [24], [34] where the processor fuzzer provides the same input to both the RTL simulator and the reference model. Here, the reference model is an ISA simulator that mimics the behavior of all the ISA-level operations. The hardware fuzzer extracts the final

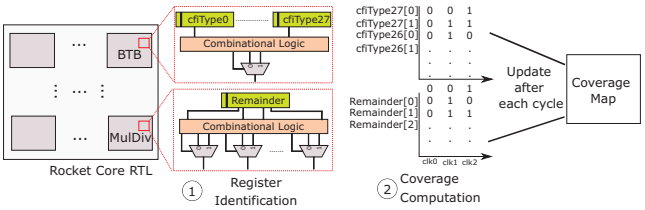
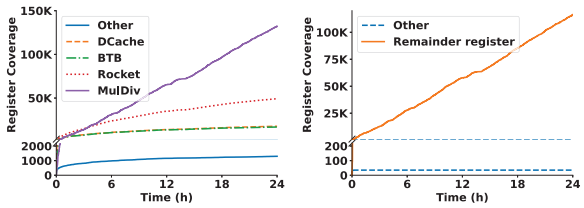



Fig. 1: Overview of DIFUZZRTL's coverage feedback strategy.




(a) Rocket core (b) MulDiv module

Fig. 2: DIFUZZRTL's register coverage breakup for Rocket.

memory states and architectural register values from both the ISA simulator and the RTL simulator for the same input and cross-checks the traces. Any mismatch is considered a potential bug in the processor and is marked for further investigation by the verification engineer.

### C. DIFUZZRTL's Register Coverage

DIFUZZRTL [24] adapts CGF to capture FSM state transitions during RTL simulation. The strategy follows a two-stage approach as depicted in Figure 1. In stage ①, it performs static analysis to identify a small set of registers in each RTL module and instruments the RTL with necessary hardware logic to record register coverage at simulation time. At a high level, DIFUZZRTL monitors a register if its value is directly or indirectly used to control a multiplexer selection signal. DIFUZZRTL creates a circuit graph of the RTL design where nodes and edges of this graph represent circuit elements (e.g., multiplexers, wires, ports, registers) and connections, respectively. Then, it recursively performs a backward data-flow analysis for each multiplexer's selection signal and identifies any register in the traversed path. In stage ②, DIFUZZRTL monitors value changes in the identified registers during the RTL simulation. For each clock cycle, DIFUZZRTL hashes all the values in the identified registers into a coverage map to represent the current FSM state. If a new hash value is observed, DIFUZZRTL increases register coverage to signify that the current test is interesting for further mutations.

DIFUZZRTL's register coverage improves prior work [33], [35] in terms of scalability, efficiency, and precision. However, using register coverage metric for hardware fuzzing can be highly misleading. At a high level, we observe that a subset of registers leads to misleading coverage increase, and therefore, misguides the hardware fuzzer. We provide more details using an example (illustrated in Figure 1) from the open-source RISC-V-based Rocket Core [2]. In the multiplication unit of Rocket Core, there is a 130-bit `remainder` register in the MulDiv module that indirectly controls 98 mux selection signals. Therefore, DIFUZZRTL identifies this register to monitor during fuzzing[2]. The change in the value of `remainder` results in an increase in coverage. In Figure 2, we demonstrate the coverage increase resulted from the `remainder` register during a 24-hour fuzzing session. First, in Figure 2a, we depict the coverage progress of different modules in the Rocket core. Clearly, the MulDiv module (multiplication unit of Rocket core) dominates the module-wise register coverage. 62% of overall register coverage results from the MulDiv module at the end of 24-hours. Figure 2b further shows the contribution of the `remainder` register to the coverage increase in the MulDiv module. Compared to all other registers in the MulDiv module, `remainder` register is clearly major factor that causes increase in register coverage.

Broadly, as pointed out with the above example, DIFUZZRTL monitors and uses coverage information from registers even if they are mostly involved in datapath-related operations and have minimal control over the current FSM state of the hardware. Unfortunately, data-path registers (e.g., `remainder`) increase search space significantly, yet the coverage increase resulting from data-path registers indeed does not provide meaningful information to the fuzzer related to the current hardware state. Therefore, it is not interesting to keep an input for further mutations if it increases coverage based on data-path registers. In our work, we present a new coverage metric that aims to tackle this problem.

[2]DIFUZZRTL applies some optimizations to reduce search space. As one of their optimizations, it is able to track only a subset of bits of a register and therefore; ultimately tracks 98 bits of the `remainder` register.

## III. ProcessorFuzz

### A. Design Overview

We illustrate the design overview of ProcessorFuzz in Figure 3. In stage **(1)**, ProcessorFuzz is provided with an empty seed corpus. It populates the seed corpus by generating a set of random test inputs in the form of assembly programs that conforms to the target ISA. Next, ProcessorFuzz chooses a test input from the seed corpus in stage **(2)** and subsequently applies a set of mutations (such as removing instructions, appending instructions, or replacing instructions) on the chosen input in stage **(3)**. For these three stages, ProcessorFuzz uses the same methods applied by a prior work [24]. In stage **(4)**, ProcessorFuzz runs an ISA simulator with one of the mutated inputs and generates an extended ISA trace log that includes the value of CSRs for each executed instruction. The Transition Unit (TU) receives the ISA trace log, extracts the transitions that occur in the CSRs, and cross-checks each transition against the Transition Map (TM) in stage **(5)**. The TM is initially empty and populated with unique CSR transitions during the fuzzing session. If the observed transition is not present in the TM, it is classified as a unique transition and added to the TM. In case the current test input triggers at least one new transition, the input is deemed interesting and added to the seed corpus for further mutations. Otherwise, the input is discarded. In stage **(6)**, ProcessorFuzz runs the RTL simulation of the target processor with only the interesting mutated input. The RTL simulation also generates an extended RTL trace log similar to

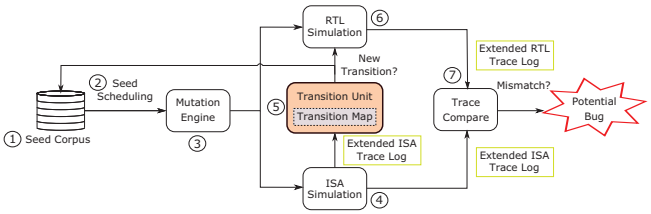



Fig. 3: ProcessorFuzz Design.

| # | PC | Instruction | [ Privileged | Unprivileged ] |
|---|---|---|---|---|
| 1 | 0x045c | sret | [8000000a00006000,00,0f,b100, | 0,00 ] |
| 2 | 0x283c | sraiw s5, s0, 6 | [8000000a00006020,00,0f,b100, | 0,00 ] |
| 3 | 0x2840 | fdiv.s fs11, ft0, fa7 | [8000000a00006020,00,0f,b100, | 0,00 ] |
| 4 | 0x2844 | fence iorw,iorw | [8000000a00006020,00,0f,b100, | 0,03 ] |
| 5 | 0x2848 | fsqrt.s ft0, ft5 | [8000000a00006020,00,0f,b100, | 0,03 ] |

Fig. 4: Extended trace log generated by the ISA simulator. The values (in hexadecimal) of a subset of CSRs in Table I are included within the square brackets in the given order; `mstatus`, `mcause`, `scause`, `medeleg`, `frm`, and `fflags`. Transitions are color coded; red and blue for `mstatus` and `fflags` CSR transitions, respectively.

the extended ISA trace log. The ISA trace log and the RTL trace log are compared in stage (**7**). Any mismatch between the logs signifies a potential bug that needs to be confirmed by a verification engineer usually by manual inspection.

### B. Feedback from the ISA Simulation

One design feature of ProcessorFuzz is that it relies on the ISA simulation to determine if a test input is interesting as opposed to prior works that rely on the RTL simulation. We use the ISA simulator to capture the CSR transitions for two main reasons. First, ISA simulators are generally much faster in executing a given program in comparison to executing that program on a processor using the RTL simulation. For instance, we observed that the RISC-V Spike ISA simulator [53] is, on average 79× faster than the RTL simulation of the RISC-V BOOM processor. This speedup provides a considerable advantage as ProcessorFuzz can then quickly identify if a test input is interesting without performing the slow RTL simulation. Eliminating inputs with similar characteristics helps ProcessorFuzz achieve faster bug discovery times as shown in Section IV. Indeed, ProcessorFuzz discovered all the bugs found by the existing processor fuzzer (i.e., DIFUZZRTL).

Second is the reduced effort needed to instrument the simulator. A simulator needs to be instrumented to generate an extended trace log with the selected CSRs. An ISA simulator can be easily instrumented by extending the already available trace logic with the selected CSRs. The same instrumented ISA simulator can be used to fuzz any processor design as long as it has been designed for the same ISA target. In contrast, instrumenting RTL designs for tracking the coverage metrics requires extensive effort. Moreover, instrumentation in one HDL does not readily translate to other HDLs. Additionally, as shown in Section IV, ProcessorFuzz incurs limited instrumentation overhead during fuzzing (only 1% in ISA simulator) as opposed to prior works [30] that instrument processor RTL and result in higher runtime overheads (e.g., 71% overhead in TheHuzz [30] and 97% overhead in RFUZZ [33]).

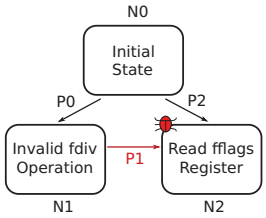



Fig. 5: Abstract state diagram for triggering Bug 2 in Table IV.

### C. CSR-transition Coverage

*1) Description of the Metric:* As described in Section II-C, DIFUZZRTL's register coverage technique monitors many datapath registers (e.g., `remainder` register) to determine the current FSM state, which leads to large state space. To test the processor with as many qualitatively distinct input patterns as possible, we propose a novel CSR-transition coverage metric.

CSRs are system registers in an ISA specification. These registers are used to control (e.g., delegated exceptions) or hold information (e.g., state of the floating-point unit) about the current architectural state of the processor. Our intuition for using CSRs is as follows. A processor is a complex FSM where CSRs have direct control over the current processor state. Architectural state of the processor (held in the register file and status registers) represents the state of a program running in the processor. A value change in a CSR often signifies an architectural state change such as a value change in a CSR that stores exception code or privilege level. Therefore, ProcessorFuzz aims to realize the current state of the processor by monitoring transitions in CSRs to guide the fuzzer towards interesting processor states.

CSR transitions can be extracted from either the ISA simulator or the RTL simulation of the processor design. ProcessorFuzz uses the ISA simulator to capture CSR transitions. Specifically, ProcessorFuzz monitors the CSR values resulting from the execution of the previous and current instructions and checks if they differ. If so, ProcessorFuzz uses the transition to determine if the input is interesting as detailed in the following subsections. We provide a concrete example using the extended ISA trace log shown in Figure 4 to illustrate how ProcessorFuzz identifies a CSR transition in the ISA trace log. After execution of the *sret*, the CSR value changes which can be seen by comparing the entries in line 1 and 2 of the 'Privileged' column. Specifically, we observe a CSR-transition in `mstatus` CSR as highlighted in red in Figure 4.

*2) Why Transitions Instead of Values?:* DIFUZZRTL determines the current processor state based on the register coverage as detailed in Section II-C. For each newly covered FSM state, DIFUZZRTL's register coverage only stores the current state of the processor and does not consider the previous state. Unfortunately, this design choice can lead to important test inputs being discarded by the fuzzer and the fuzzer can potentially miss out on the discovery of a bug. We illustrate this in Figure 5. The figure represents a subset of the abstract states associated with a real-world bug (Bug 2 in Table IV) that we

identified in a RISC-V processor. The processor starts out in the N0 state. The bug triggers in the N2 state only if the previous state is N1. During a coverage-guided fuzzing session, if both N1 (through P0 transition) and N2 (through P2 transition) are covered individually, there will not be a coverage increase for the denoted P1 state transition. Hence, the unique P1 transition is not particularly driven towards. Thus, the fuzzing session fails to trigger the bug. Contrarily, by monitoring transitions, we can detect P1 as a new transition even though N1 and N2 states are already covered. Overall, we monitor new transitions in CSRs rather than just identifying unique CSR values to improve the sensitivity of the feedback metric. Indeed, our rationale is similar to widely-used software fuzzers' [19], [38] rationale that monitor edges in a program instead of basic blocks.

*3) CSR Selection Criteria:* An ISA specification usually specifies a large number of CSRs[3]. Monitoring all available CSRs for transitions can mislead the fuzzer (as we show in Section IV) because not all CSRs provide distinctive information regarding the current processor state. As an example, consider `instret` CSR that holds the total number of retired instructions. Monitoring the `instret` results in a scenario where each committed instruction by the processor results in a CSR transition. Effectively, ProcessorFuzz would identify any test input as interesting since the `instret` causes a transition after each committed instruction. However, a test would rarely result in a bug due to a change in committed instruction count.

To aid ProcessorFuzz in determining qualitatively different inputs, we introduce the following two criteria when selecting the CSRs that ProcessorFuzz monitors for transitions. First, we select CSRs that contain status information about the processor (criteria C1). These CSRs are important because they directly reveal the current status of the processor. As an example, we select a CSR that stores the cause for an exception taken by the processor (e.g., `mcause`). If a test case results in an exception, ProcessorFuzz analyzes the cause and differentiates it from another test case that has a different exception reason (e.g., misaligned load/store attempt or access faults due to unauthorized privilege mode). Second, we select any CSR that is used to set a certain configuration in the processor (criteria C2). Here, we aim to realize if the processor behaves as expected under different configurations. For instance, the value of `medeleg` can be changed to determine which traps can be delegated to lower privilege levels (e.g., the load access fault handled in supervisor mode instead of machine mode). This way, ProcessorFuzz aims to realize if processor designs can perform correctly under different configurations (e.g., different exception delegations) for a particular processor status (e.g., an exception). Table I lists all the CSRs in the RISC-V ISA that we used for identifying transitions in the current implementation of ProcessorFuzz based on the aforementioned two criteria, i.e., C1 and C2. We also provide all the CSRs that we excluded (e.g., `instret`) along with details why they are not considered as part of ProcessorFuzz's current design in Table II.

Apart from these two criteria, CSR selection can be further limited depending on the desired scope of verification. For example, if we only want to verify the functionality of the floating-point unit in the processor, only floating-point CSRs can be monitored to identify transitions. We quantitatively demonstrate this capability of ProcessorFuzz in Section IV-B.

[3]As a reference point, RISC-V ISA defines up to 4096 CSRs.

### D. Transition Unit

As shown in Figure 3, the TU takes an extended ISA trace log as input and communicates with the TM to output whether the trace log contains any new transitions. We describe the complete workflow of the TU in this section.

**Filtering Transitions.** As a first step, the TU extracts all CSR transitions in the trace log based on the description in Section III-C. Then, ProcessorFuzz applies a filter to remove unnecessary transitions. We observe that not all CSR transitions represent interesting architectural state changes that are relevant for testing processors. For instance, a test program running on the target processor can write to a CSR that contains processor status, e.g. `mstatus` CSR in RISC-V ISA. This could get identified as a new CSR transition. If the write operation is legal, the processor continues the execution of the program and eventually overwrites the CSR with the updated status. Overall, the type of transitions that occur from writes to status CSRs do not affect the architectural state of the processor. Thus, ProcessorFuzz filters out transitions that occur from explicit writes to status CSRs.

**Grouping Transitions.** Next, the TU groups the transitions to reduce the state space. ProcessorFuzz provides the flexibility to customize the CSR-transition coverage metric to be suitable for verifying different Architectural Units (AUs) individually. Specifically, ProcessorFuzz allows a designer to group CSR transitions of AUs, thereby considering them as independent events. Grouping transitions improves the exploration of CSR transitions within each group. As a result, the fuzzer is able to generate tests targeted towards individual AUs and verify them thoroughly. This is a useful feature for a verification engineer as AUs in a processor can be individually verified as an initial step of verification. For example, privileged and unprivileged architectures in a RISC-V processor can be verified individually by grouping transitions as shown in Figure 4. Identifying and fixing the bugs in each AU before fuzzing the processor as a whole can reduce the overall verification effort.

**Transition Map** ProcessorFuzz maintains a transition map to store CSR-transitions. Each transition is stored in the map as a tuple: $(I_m, S_0, S_1)$ where $I_m$ is the mnemonic of the instruction whose execution resulted in the CSR transition. $S_0$ and $S_1$ are CSR values before and after the transition as defined in subsection III-C. Revisiting the same example given in subsection III-C, the unprivileged CSR-transition in lines 3 and 4 in Figure 4 can be represented as (fdiv.s, 0000, 0003). We include instruction mnemonic because the same transition can be triggered by different instructions. For example, both floating-point division and floating-point square-root instructions can trigger the same transition in fflags CSR in RISC-V ISA due to invalid operations. Nevertheless, only the invalid operation of floating-point division instruction might contain a bug. Only the mnemonic of the instructions is included to ignore repetitive transitions that get triggered by different operands of the same

TABLE I: CSR selection for RISC-V ISA implementation of ProcessorFuzz along with the criteria that was used to select them. Here, C1 and C2 correspond to two criteria that we describe in Section III-C3.

| CSR Group | CSR | Description | Criteria |
|---|---|---|---|
| Privileged | mstatus.xIE | Controls the global interrupt enable bit for privilege x, x = {M, S, U} | C2 |
| | mstatus.xPIE | Holds the value of interrupt-enable bit active prior to the trap for privilege mode x | C1 |
| | mstatus.xPP | Holds the previous privilege mode active prior to a trap taken to privilege mode x | C1 |
| | mstatus.XS | Contains the state of any additional user-mode extensions | C1 |
| | mstatus.FS | Contains the state of the floating-point unit | C1 |
| | mstatus.MPRV | Controls the privilege mode in which the memory operations are performed | C2 |
| | mstatus.SUM | Controls the permission for accessing user memory from supervisor mode | C2 |
| | mstatus.MXR | Controls the privilege with which loads access virtual memory | C2 |
| | mstatus.TVM | Controls the ability to edit virtual-memory configuration from supervisor mode | C2 |
| | mstatus.TW | Controls the privilege modes that wait for interrupt (WFI) is allowed to execute | C2 |
| | mstatus.TSR | Provides the ability to trigger a trap when SRET instruction is executed in supervisor mode | C2 |
| | mstatus.xXL | Controls the width of an integer register for privilege mode x, x = {S, U} | C2 |
| | mstatus.SD | Indicate the combined state of mstatus.FS and mstatus.XS for context switches | C1 |
| | {m,s}cause | Contains the trap cause when a trap is taken in to machine or supervisor mode | C1 |
| | medeleg | Decides what type of exceptions are delegated to supervisor mode from machine mode | C2 |
| | {m,s}counteren | Controls the availability of the hardware performance-monitoring counters for supervisor or user mode | C2 |
| Unprivileged | frm | Controls the dynamic rounding mode for floating-point operations | C2 |
| | fflags | Holds the accrued exceptions from the floating-point operations | C1 |

TABLE II: CSRs not monitored by ProcessorFuzz along with the reason for exclusion.

| Category | CSR | Description | Reason for Exclusion |
|---|---|---|---|
| Privileged | misa | Reports the CPU capabilities of a hart | Holds a constant value during testing |
| | mhartid | Contains the integer ID of the hardware thread running the code | |
| | {m,s}tvec | Contains the trap handler base address and vector configuration for machine or supervisor mode | |
| | satp | Controls supervisor-mode address translation and protection | |
| PMP | pmpcfg | Contains the physical memory protection configuration | |
| | pmpaddr | Contains the physical memory protection addresses | |
| Interrupt | {m,s}ip | Reports pending interrupts in machine or supervisor mode | Not supported by the testing infrastructure (e.g., RISC-V ISA vector extension is not supported and relevant CSRs are excluded.) |
| | {m,s}ie | Control what interrupts are enabled in machine or supervisor mode | |
| | mideleg | Decides what type of interrupts are delegated from machine mode to supervisor mode | |
| Debug Extension | dcsr | Contains the configuration and status of debug extension | |
| | dpc | Holds the program counter of the next instruction to be executed before entering debug mode | |
| | dscratch | Optional scratch register that holds temporary values | |
| | tselect | Control which trigger is accessible through the other trigger registers | |
| | tdata1-3 | Holds trigger-specific data | |
| Vector Extension | vstart | Holds the index of the first element to be executed by a vector instruction | |
| | vxsat | Holds the saturation flag for fixed-point operations | |
| | vxrm | Controls the rounding mode used in the vector extension | |
| HPC | mcountinhibit | Controls which hardware performance-monitoring counters are allowed to increment | Contains information to assist designers during analysis of a hardware bug rather than revealing the fundamental issue |
| | cycle | Holds the elapsed cycle count of the CPU | |
| | instret | Holds the number of retired instruction count | |
| | hpmevent | Hardware performance-monitoring event selector | |
| | hpmcounter | Performance-monitoring counter of the event selected by hpmevent | |
| Privileged | {m,s}tval | Hold the exception-specific information when a trap is taken to machine or supervisor mode | |
| | mscratch | Holds a pointer to the machine mode context space while the hart executes in lower privilege | |
| | {m,s}epc | Contains the PC of an instruction that caused an exception for machine or supervisor mode | |
| | sscratch | Holds a pointer to the supervisor mode context space while the hart executes in user mode | |

instruction. Once tuples are created, the map is queried to check whether the detected transition is new or a duplicate. Tuples that are identified to contain new transitions are added to the map while marking the current test input as interesting. The transition map is empty at the beginning of a fuzzing session and maintained throughout the session.

### E. RTL Simulation and Trace Comparison

If the TU determines that the current input results in a unique CSR transition, ProcessorFuzz launches the RTL simulation and generates the extended RTL trace log. ProcessorFuzz then compares the extended RTL trace log with the extended ISA trace log. Any difference between these logs signifies a potential bug in the processor design and needs to be investigated further by a verification engineer. In case the input does not result in a unique transition, ProcessorFuzz discards the input and proceeds to the next fuzzing iteration.

## IV. Evaluation

In this section, we evaluate the effectiveness of ProcessorFuzz using real-world processor designs.

### A. Evaluation Setup

*1) Implementation Details:* ProcessorFuzz has two main implementation steps; generation of an extended trace log using the ISA simulator and building the TU (see Figure 3). For the former, we extended Spike [53] open-source ISA simulator to store the values of monitored CSRs (see Table I). The instrumentation overhead of Spike is 0.4% in terms of lines of C++ code, while the runtime overhead is 0.15%. For the RTL simulation of all processor designs, we used Verilator [52], an open-source RTL simulator. We used the same mutation engine (see Figure 3) as provided by DIFUZZRTL's open-source repository. Using the same engine is important since our goal is to compare two coverage feedback mechanisms (i.e., register coverage and CSR-transition coverage) rather than input generation mechanisms. We separated transitions belonging

to `frm` and `fflags` to separate floating-point operations from the rest of the CSRs.

*2) Processor Designs:* We use three real-world open-source RISC-V processors. **Rocket Core** is a Chisel [3] HDL-based open-source, general-purpose, in-order RISC-V processor core that can be generated using the Rocket Chip SoC Generator framework [2]. We used Spike [53] as a reference model to verify the correctness during fuzzing. The commit version of the Rocket core that we used is `148d5d2`. **BOOM Core** [9] is an out-of-order, superscalar RISC-V processor core. It can also be generated from the same Rocket Chip SoC Generator framework [2] and is also designed in Chisel HDL. We used Spike ISA simulator to verify the correctness during fuzzing. The commit version of the BOOM core that we used is `148d5d2`. **BlackParrot Core** [48] is an open-source 64-bit RISC-V core, designed in the industry-standard SystemVerilog HDL. BlackParrot is silicon-validated and is in active development. We used Dromajo [56] as a reference model to expose the bugs in BlackParrot (commit `bc3b48b`).

*3) Settings:* We compared ProcessorFuzz with two different settings of DIFUZZRTL. The first setting is `no-cov-difuzzrtl` where DIFUZZRTL fuzzing framework is used without any coverage guidance (i.e., as a blackbox fuzzer). For all the cores that we evaluated, we successfully used this setting as a comparison point. The second setting is `reg-cov-difuzzrtl` where DIFUZZRTL fuzzing framework relies on register coverage as a guidance mechanism. While this setting is applicable to Rocket and BOOM Cores, it is not the case for BlackParrot Core. This is because DIFUZZRTL's register coverage passes do not support SystemVerilog. They are tailored for FIRRTL [27], an intermediate representation (IR) used by Chisel HDL, which is used to design Rocket and BOOM cores. We tried to convert SystemVerilog to FIRRTL using an open-source tool (i.e., Yosys [62]), and apply DIFUZZRTL's register coverage passes. However, we observed several issues during this conversion due to the limited support for SystemVerilog to FIRTTL conversion and thus failed to instrument BlackParrot. In our experiments, we used DIFUZZRTL as the sole comparison point since it shows clear benefits over previous processor fuzzing frameworks as well as its open-source nature. Also, for each setting, we reported Time-to-Exposures (TTE) which is defined as the total elapsed time from the starting of the fuzzing session until the bug is exposed.

*4) Infrastructure:* All the experiments based on ISA and the RTL simulations were conducted on server nodes with Intel®Xeon®E5-2670 CPUs and CentOS Linux 7 as the operating system. We fuzzed each processor design 10 times for each setting and allocated 48 hours (2 days) of time limit for each fuzzing instance. For each fuzzing instance, we dedicated two cores and 8GB of memory. In total, it took 4320 CPU hours to conduct all the experiments.

### *B. Ground-truth Bugs*

As discussed by prior works [31], [39], the bug-finding capability of a fuzzer is the ultimate litmus test for a fuzzer. While there exist several fuzzing benchmarks for software programs [22], [36], this is not the case for processors. Therefore, we relied on a set of bugs (in total six bugs) previously reported by DIFUZZRTL for BOOM processor to evaluate the bug-finding capability of ProcessorFuzz and perform a head-to-head comparison with DIFUZZRTL. Overall, our evaluation aims to demonstrate that ProcessorFuzz can guide the fuzzer efficiently to discover ground-truth bugs thanks to the CSR-transition feedback obtained using the ISA simulation.

In Table III, we report the TTE of bugs in seconds for three different settings in 2nd-4th columns; `no-cov-difuzzrtl`, `reg-cov-difuzzrtl`, and ProcessorFuzz `selected` configuration. `selected` configuration of ProcessorFuzz uses the CSRs in Table I for transition extraction based on the criteria that we detailed in Section III-C3. We also provide the achieved speedups by ProcessorFuzz over `no-cov-difuzzrtl`, and `reg-cov-difuzzrtl`. Besides `selected`, we provide results for two more different configurations of ProcessorFuzz in column 5th-6th; `fp-csr`, and `all-csr`. These configurations differ in the CSRs that ProcessorFuzz monitors during fuzzing. Specifically, `all-csr` configuration monitors all implemented CSRs in the BOOM core. Here, by using `all-csr` configuration, we aim to present that ProcessorFuzz can be effectively guided towards bugs by eliminating certain CSRs that do not assist fuzzing towards exploring bugs (e.g., `instret` that repeatedly changes after an instruction retires). Finally, `fp-csr` configuration uses only the floating-point CSRs (unprivileged CSRs in Table I). The aim of this experiment is to show that ProcessorFuzz can focus on certain parts of processors by selecting a subset of CSRs (e.g., floating point unit). Overall, ProcessorFuzz `selected` configuration and DIFUZZRTL discovered five out of six bugs reported in the DIFUZZRTL within the fuzzing time limit in our experiments. Unfortunately, we could not detect #504 with any of the settings. In summary, ProcessorFuzz (`selected`) achieved, on average, 1.21× (up to 2.1×) and 1.23× (up to 2.32×) speedups over `no-cov-difuzzrtl` and `reg-cov-difuzzrtl`, respectively. `no-cov-difuzzrtl` performed slightly better than `regcov-difuzzrtl`.

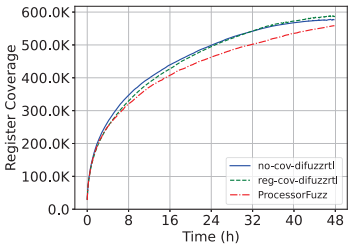
We included `fp-csr` configuration to demonstrate the ProcessorFuzz's ability to change the scope of verification by changing the CSR selection. `fp-csr` detected the bugs in the floating-point unit (issues #492, #493 and #503) 2.08× faster compared to the `selected` configuration while showing a slowdown in detecting other bugs.

We also show the effect of CSR selection on TTE of the bugs through `all-csr` configuration. `all-csr` configuration failed to detect two of the bugs within the allocated fuzzing time. Moreover, `selected` is significantly faster (i.e., 16.49× on average) than `all-csr` in detecting bugs.

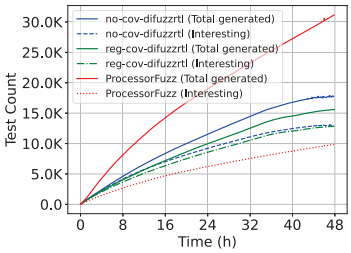
To understand the performance of ProcessorFuzz and DIFUZZRTL for different bugs, we further study the relationship among register coverage, CSR-transition coverage, and bug-finding times. Specifically, in Figure 6a, we show the measured register coverage progress for different settings of DIFUZZRTL and ProcessorFuzz. Although ProcessorFuzz covers

TABLE III: The speedup achieved by `selected` ProcessorFuzz configuration over no-cov-difuzzrtl, and reg-cov-difuzzrtl for the ground-truth bugs in the BOOM processor. We also report speedup of `fp-csr` and `all-csr` ProcessorFuzz configurations over `selected` ProcessorFuzz configuration. The runtime is set as 48 hours (172800 seconds) for bugs that could not be found.

| | **no-cov-difuzzrtl** | **reg-cov-difuzzrtl** | | **ProcessorFuzz (selected)** | | | **ProcessorFuzz (fp-csr)** | | **ProcessorFuzz (all-csr)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| Issue No | Time (s) | Time (s) | Speedup (over `no-cov`) | Time (s) | Speedup (over `no-cov`) | Speedup (over `reg-cov`) | Time (s) | Speedup (over `selected`) | Time (s) | Speedup (over `selected`) |
| #458 | 104.3 | 70.3 | 1.48 | 54 | 1.93 | 1.3 | 151324.8 | 0.0 | 172800 | NA |
| #454 | 32883.3 | 45322 | 0.73 | 25020 | 1.31 | 1.81 | 119886.2 | 0.2 | 39523.3 | 0.63 |
| #492 | 2047.2 | 4238.9 | 0.48 | 1821.2 | 1.12 | 2.32 | 1221.8 | 1.49 | 172800 | NA |
| #493 | 585.4 | 494.9 | 1.18 | 278.7 | 2.1 | 1.77 | 170.1 | 1.63 | 526.6 | 0.52 |
| #503 | 1463.7 | 1011.1 | 1.44 | 2795.9 | 0.52 | 0.36 | 757.6 | 3.69 | 62246.8 | 0.04 |
| #504 | 172800 | 172800 | NA | 172800 | NA | NA | 172800 | NA | 172800 | NA |
| Geo. | 3182.9 | 3245.1 | 0.98 | 2630.7 | 1.21 | 1.23 | 8890.2 | 0.29 | 43402.2 | 0.06 |




(a) Register coverage progress during fuzzing.




(b) Coverage increasing and total test input counts during fuzzing.

Fig. 6: Coverage details for different settings.

less number of states (i.e., achieves lower register coverage) during fuzzing, it was still able to discover bugs faster. For instance, ProcessorFuzz triggered the most challenging bug based on the TTE (i.e., #454) after exploring 303K states while `no-cov-difuzzrtl` and `reg-cov-difuzzrtl` triggered that bug after exploring 364K and 354K states, respectively. This particular bug shows that higher register state coverage does not necessarily translate to a faster bug discovery. Indeed, an increase in coverage due to value changes in datapath registers can mislead the fuzzer since inputs with similar characteristics (the multiplication example in Section II-C) are repeatedly used by the fuzzer to generate a new set of inputs.

In Figure 6b, we also show the total number of test inputs that lead to a coverage increase, i.e. 'interesting test inputs', and the total number of inputs generated by the mutation engine for the two settings of DIFUZZRTL and ProcessorFuzz. For `no-cov-difuzzrtl` and `reg-cov-difuzzrtl`, we use the register coverage metric, the same metric used in DIFUZZRTL, to realize if a test input increases coverage. For ProcessorFuzz, we use the CSR-transition coverage metric to detect inputs that resulted in a coverage increase. The results provide an important takeaway. Although ProcessorFuzz generates significantly more inputs than other approaches, it is very selective when categorizing a test input as 'interesting'. Consequently, ProcessorFuzz identified only 33% of the generated test inputs as interesting. Moreover, ProcessorFuzz could expose the bugs faster although it used the least number of test inputs for the RTL simulation. Note that ProcessorFuzz launched the RTL simulation only with interesting inputs (i.e., curved dotted red line) and discarded any other generated input. Using the fast ISA simulation enabled ProcessorFuzz to quickly eliminate inputs that do not result in a new FSM state and spend more time on inputs that explore new FSM states.

### C. Newly Discovered Bugs

In Table IV, we document the various **new** bugs discovered by ProcessorFuzz in the selected processors mentioned earlier and in the ISA simulator used as a reference model. Here, we provide detailed descriptions of three bugs chosen from different processors and reference model. The details of the remaining bugs can be found in respective processor repositories.

*1) Bug Descriptions:* **Bug 6.** Any write attempt to the zero register (i.e., `x0`) must be ignored according to the RISC-V ISA. However, in BlackParrot, we detected that the `x0` register is read as a non-zero value if one of the preceding division instructions that writes to `x0` is still in the pipeline. Further analysis revealed that this discrepancy is due to bypassing the result of division operation to the following instruction even when the destination register of a division operation is `x0`. ProcessorFuzz was able to identify this bug because a test input that has this scenario caused a CSR transition in `fflags` due to division by zero. An attacker can use this bug to obfuscate the behavior of malware. Specifically, malware can jump to an address computed by an instruction that uses `x0`.

**Bug 7.** According to RISC-V privileged specification, the effective privilege mode for implicit page table accesses should be supervisor mode. However, we observed that Dromajo accesses page tables in user mode privilege level when executing user-mode programs. Further analysis revealed that Dromajo also carries out Physical Memory Protection (PMP) checks in user mode when no PMP entries are set, violating the RISC-V ISA privileged specification in two counts.

**Bug 8.** In a multi-level page table implementation, the accessed (A), dirty (D), and user-mode (U) bits of a non-leaf page table entry (PTE) are reserved for future use and should be cleared. If these bits are set in a non-leaf PTE, the processor must raise an instruction page fault when accessing the PTE according to

TABLE IV: Brief description of bugs discovered by ProcessorFuzz, and their current status, in various processor cores.

| Bug | Core / Simulator | Brief Description of the Bug | Status (Issue No) |
|---|---|---|---|
| 1 | BlackParrot | Non-boxed single-precision floating point values are not interpreted as NaNs | Confirmed (#971) |
| 2 | BlackParrot | Read-after-Write dependencies on `fcsr.fflags` are not satisfied. | Fixed (#994) |
| 3 | BlackParrot | When `mstatus.FS` is not set and the `fcsr` is written, `FS` is unexpectedly updated. | Fixed (#969) |
| 4 | BlackParrot | The 2 low-bits of `sepc` CSR are not write-insensitive. | Fixed (#970) |
| 5 | BlackParrot | No exception raised when writing certain read-only CSRs. | Fixed (#967) |
| 6 | BlackParrot | Reading `zero` register, following specific instruction sequences, return unexpected non-zero values | Fixed (#832) |
| 7 | Dromajo | PMP checks are performed, and raise exceptions upon encountering violations, even with no PMP entries set. | Confirmed (#46) |
| 8 | Rocket & BOOM | Instruction page fault not raised when accessing non-leaf PTEs with certain unspecified page attributes. | Fixed (#2905, #570) |
| 9 | BOOM | `mstatus.FS` is gratuitously set to dirty. | Confirmed (#969) |

the RISC-V ISA. We discovered that Rocket and BOOM cores do not raise instruction page fault when software attempts to access a PTE with any of A, D, or U bits set. This bug is similar to CWE-1209 [43] where failure to disable reserved bits allows attackers to compromise the hardware state.

*2) Timing Results:* Table V provides the TTEs for six newly identified bugs (Bug 1-6) in BlackParrot. We did not include Bug 7-9 since they were easily detected in all the settings. We were only able to compare ProcessorFuzz with `no-cov-difuzzrtl`. As detailed in Section IV-A3, we could not instrument BlackParrot with register coverage since DIFUZZRTL lacks support for SystemVerilog. ProcessorFuzz does not require any instrumentation on the RTL design, therefore, could successfully guide the fuzzer with CSR-transition coverage to expose bugs. Overall, ProcessorFuzz achieved 1.57× speedup, on average, over `no-cov-difuzzrtl`. Note that only ProcessorFuzz was able to detect Bug 6 from Table IV. Similar to the experiment that we conducted in the BOOM processor using the ground-truth bugs, `selected` configuration of ProcessorFuzz performed significantly better compared to `all-csr` configuration (i.e., 15.61× faster). Moreover, `fp-csr` configuration identified floating-point related bugs fairly faster (e.g., Bug 3) compared to other type of bugs (e.g., Bug 4 that focuses on `sepc` CSR).

## V. Related Work

We first present traditional methods in hardware verification. Then, we explain fuzzing-based hardware verification approaches and how ProcessorFuzz differs.

### A. Traditional Hardware Verification

Random instruction generators [15], [20], [21], [23], [34] have been commonly used in processor verification since they require limited human expertise and are scalable to large RTL designs. The lack of coverage guidance in these tools leads to the generation of the repetitive inputs that test the same processor functionalities, thereby decreasing the chances of finding bugs [24], [33]. A verification engineer can target the uncovered RTL regions by adjusting the constraints that control the random test generator. However, this method significantly increases engineering effort, and therefore, slows down the verification process. To overcome this problem, researchers proposed several coverage-directed test generation mechanisms [4], [14], [16], [47], [54], [55], [59] that automatically direct the next round of test generation to target the uncovered parts of RTL. Unfortunately, these works are generally DUT-specific which hinders their general applicability.

Formal verification methods (e.g., symbolic execution and model checking) are also widely used in hardware verification [6], [10], [46]. These methods use mathematical reasoning to prove that a hardware design conforms to its specification. Unfortunately, formal verification methods have a well-known state explosion problem, and therefore, do not scale well for complex RTL designs such as a processor [12].

### B. Hardware Fuzzing

In Table VI, we provide a high-level overview of all fuzzing-based RTL verification approaches. For each approach, we include the input format, the coverage metric used to guide the fuzzer, and the method to identify bugs.

RFUZZ [33] proposes a new metric, the multiplexer toggle coverage. RFUZZ monitors all the multiplexers in the RTL design. It retains an input for further mutations if the input toggles a previously uncovered multiplexer selection signal. A follow-up work by Li et al. [35] enhances RFUZZ with symbolic simulation. Both RFUZZ and Li et al. are highly coupled to Chisel HDL which limits the applicability of the approach [49]. Additionally, monitoring multiplexers in complex designs introduces excessive performance overhead [24]. ProcessorFuzz is agnostic to HDL, which makes it both practical and efficient.

Trippel et al. [57] translate hardware designs to software models and fuzzes those models. This way, available coverage metrics used by software fuzzers (e.g., basic block and edge) can be used for fuzzing hardware as well. However, this method of converting hardware designs to software models introduces additional challenges such as proving the equivalency between hardware design and software model [49].

TheHuzz [30] relies on a variety of coverage metrics extracted using industrial-standard tools such as Cadence [7] and ModelSim [51]. TheHuzz profiles individual instructions to associate with relevant mutation strategies while generating new set of inputs. Unlike DIFUZZRTL or ProcessorFuzz, TheHuzz does not propose a new coverage metric. TheHuzz relies on several coverage metrics used in software testing (i.e., statement, branch, line, and expression). As discussed by prior works [24], [55], these metrics are not sufficient metrics to verify a processor. Moreover, it is not clear how registers

TABLE V: The speedup of ProcessorFuzz over no-cov-difuzzrtl, and reg-cov-difuzzrtl for the discovered bugs in the BlackParrot processor. We also report speedup of `fp-csr` and `all-csr` ProcessorFuzz configurations over `selected` ProcessorFuzz configuration. We state the maximum allowed runtime of 48 hours (172800 seconds) for bugs that could not be found.

| | **no-cov-difuzzrtl** | **ProcessorFuzz (selected)** | | **ProcessorFuzz (`fp-csr`)** | | | **ProcessorFuzz (`all-csr`)** | | |
|---|---|---|---|---|---|---|---|---|---|
| Bug | Time (s) | Time (s) | Speedup (over `no-cov`) | Time (s) | Speedup (over `no-cov`) | Speedup (over `selected`) | Time (s) | Speedup (over `no-cov`) | Speedup (over `selected`) |
| 1 | 464.9 | 230.2 | 2.02 | 430.2 | 1.08 | 0.54 | 1608.7 | 0.29 | 0.14 |
| 2 | 95695 | 57441.3 | 1.67 | 100804.9 | 0.95 | 0.57 | 122076 | 0.78 | 0.47 |
| 3 | 1520.1 | 1474.5 | 1.03 | 921.8 | 1.65 | 1.60 | 172800 | NA | NA |
| 4 | 585.3 | 308 | 1.90 | 558.8 | 1.05 | 0.55 | 13560.4 | 0.04 | 0.02 |
| 5 | 476.1 | 242.1 | 1.97 | 239.7 | 1.99 | 1.01 | 39150.9 | 0.01 | 0.01 |
| 6 | 172800 | 147942.3 | 1.17 | 148655 | 1.16 | 1.00 | 172800 | NA | NA |
| Geo. | 3849.9 | 2447.7 | 1.57 | 3044.4 | 1.26 | 0.8 | 38212.2 | 0.10 | 0.06 |

TABLE VI: Existing RTL Fuzzers.

| | **Input Format** | **Coverage Metric** | **Evaluated RTL Designs** | **Bug Discovery Method** |
|---|---|---|---|---|
| **RFUZZ [33]** | A Series of Bits | Mux Toggle | Peripherals, RISC-V Processors (Sodor 1-3-5) | Assertion |
| **Li et. al [35]** | A Series of Bits | Full Mux Toggle | Custom RISC-V Processor, OpenCore 1200 | Assertion |
| **DIFUZZRTL [24]** | Assembly | Register Coverage | RISC-V Processors (BOOM, Mork1x, Rocket Chip) | Golden Model |
| **DirectFuzz [8]** | A Series of Bits | Mux Toggle | Same as RFUZZ | Assertion |
| **Trippel et al. [57]** | Byte Sequence | Edge Coverage | RISC-V IP Cores | Golden Model, Assertion |
| **TheHuzz [30]** | Assembly | Branch, Line, Statement, Expression, DFF Toggle, FSM | RISC-V Processors ( Rocket Chip, CVA6), mor1kx, OpenCore 1200 | Golden Model |
| **HYPERFUZZER [45]** | A Series of Bits | High-Level | Custom SoC | Property Check |
| **Logic Fuzzer [29]** | A Series of Bits, Random Data | N/A | RISC-V Processors (BlackParrot, BOOM, CVA6) | Golden Model |
| **ProcessorFuzz (this work)** | Assembly | Control Path Register, ISA-Sim Transition | RISC-V processors (BOOM, BlackParrot, Rocket Chip) | Golden Model |

that control FSM coverage are identified as the industrial-tools are not open-sourced. We could not quantatively compare ProcessorFuzz with TheHuzz as TheHuzz is not open sourced.

The common goal of the aforementioned fuzzing works is to maximize coverage of an RTL design, thereby discovering bugs across the entire RTL design. Researchers have also proposed fuzzing frameworks for achieving alternate verification goals. For instance, DirectFuzz [8] adapts the notion of directed greybox fuzzing and applies it to the RTL verification. The goal of DirectFuzz is to cover certain specific RTL regions with a targeted fuzzing approach. Here, the motivation is to dedicate more fuzzing time to the RTL components that need to undergo thorough testing. HYPERFUZZER [45] introduces a new grammar that represents the hardware security properties. During fuzzing, HYPERFUZZER checks if any of the fuzzer-generated inputs violates a security property. Defining the security properties manually can be the most accurate approach if their correctness are verified. However, defining properties requires human expertise which is error-prone. Logic Fuzzer [29] randomizes control signals and states of a DUT without compromising the functional correctness of the DUT. Logic Fuzzer needs to be provided with fuzzing targets (e.g., congestible points in an RTL design), and therefore requires domain expertise. INTROSPECTRE [17] and Osiris [60] use blackbox fuzzing approach to discover microarchitectural side channels (e.g., Meltdown [37] and Spectre [32]) in processors.

## VI. Discussion and Limitations

**Other ISAs.** In this work, we demonstrated the capability of ProcessorFuzz using the RISC-V ISA. However, CSRs are not only specific to the RISC-V architecture and defined as part of many other ISAs including x86. Therefore, ProcessorFuzz is not limited to the RISC-V-based processors and can be used in processors based on other ISAs.

**Unintended RTL Transitions.** ProcessorFuzz uses ISA simulation as part of a feedback mechanism since it is faster and agnostic to the HDL. ProcessorFuzz does not use an input for RTL simulation if the input lacks a unique transition in its ISA simulation trace. One limitation of this design choice is that ProcessorFuzz can potentially miss certain bugs that follow the given scenario. If a test input would result in an unintended transition in RTL simulation but the same test input does not cause any unique transition in ISA simulation, such a test input will be discarded. Hence, the bug will not be identified.

## VII. Conclusion

This work presents ProcessorFuzz, a processor fuzzer guided by a novel CSR-transition coverage feedback obtained from ISA simulation. ProcessorFuzz demonstrates that monitoring CSR transitions can effectively guide fuzzing towards buggy processor states. Moreover, using ISA simulation instead of RTL simulation can quickly eliminate inputs that result in the same coverage, thereby helping the fuzzer to test as many qualitatively different inputs as possible. Our experimental results discovered eight new bugs in established, real-world, RISC-V processors, and one new bug in a reference model.


## Acknowledgment

Parts of this work are funded by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7856, and by NSF Awards 2118628 and 1801052. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

## REFERENCES


[1] V. V. Acharya, S. Bagri, and M. S. Hsiao, "Branch guided functional test generation at the rtl," in *IEEE European Test Symposium*, 2015, pp. 1–6.

[2] K. Asanović *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016.

[3] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Design Automation Conference*, 2012, pp. 1212–1221.

[4] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir, "A genetic approach to automatic bias generation for biased random instruction generation," in *Proceedings of the Congress on Evolutionary Computation*, vol. 1, 2001, pp. 442–448.

[5] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *IEEE Symposium on Security and Privacy*, 2014, pp. 114–129.

[6] Cadence, "JasperGold Formal Verification Platform," 2019.

[7] Cadence, "Circuit Simulation," https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-simulation.html, 2022.

[8] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *Design Automation Conference*, 2021.

[9] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun 2015.

[10] M. Chen and P. Mishra, "Property learning techniques for efficient generation of directed tests," *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 852–864, 2011.

[11] R. R. Collins, "The Pentium FOOF bug. Dr. Dobb's Journal," https://drdobbs.com/embedded-systems/the-pentium-f00f-bug/184410555, 1998.

[12] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs," in *USENIX Security Symposium*, 2019, pp. 213–230.

[13] A. Edelman, "The mathematics of the pentium division bug," *SIAM Review*, vol. 39, no. 1, pp. 54–67, 1997.

[14] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Design Automation Conference*, 2003, pp. 286–291.

[15] I. Futurewei Technologies, "force-riscv," https://github.com/openhwgroup/force-riscv, 2020.

[16] R. Gal, E. Haber, W. Ibraheem, B. Irwin, Z. Nevo, and A. Ziv, "Automatic scalable system for the coverage-directed generation (cdg) problem," in *Design, Automation & Test in Europe Conference & Exhibition*, 2021, pp. 206–211.

[17] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *International Symposium on Computer Architecture*, 2021, pp. 874–887.

[18] Google, "Oss-fuzz: Continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2016.

[19] Google, "American fuzzy lop," https://github.com/google/AFL, 2017.

[20] Google, "Riscv-dv," https://github.com/google/riscv-dv, 2021.

[21] S. Group, "Shakti aapg," https://gitlab.com/shaktiproject/tools/aapg/-/wikis/Wiki, 2018.

[22] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *ACM POMACS*, vol. 4, no. 3, pp. 1–29, 2020.

[23] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient cross-level testing for processor verification: A risc-v case-study," in *Forum for Specification and Design Languages (FDL)*, 2020, pp. 1–7.

[24] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *Security and Privacy*, 2021, pp. 1286–1303.

[25] Intel, "Intel Annual Report," https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html, 1994.

[26] Intel, "Machine check error avoidance on page size change," https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/software-security-guidance/technical-documentation/machine-check-error-avoidance-page-size-change.html, 2022.

[27] A. Izraelevitz *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *International Conference on Computer-Aided Design*, 2017, pp. 209–216.

[28] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016, p. 380–392.

[29] N. Kabylkas, T. Thorn, S. Srinath, P. Xekalakis, and J. Renau, "Effective processor verification with logic fuzzer enhanced co-simulation," in *International Symposium on Microarchitecture*, 2021, pp. 667–678.

[30] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *USENIX Security Symposium*, 2022, pp. 3219–3236.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *ACM SIGSAC CCS*, 2018, pp. 2123–2138.

[32] P. Kocher and othersl, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy*, 2019, pp. 1–19.

[33] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[34] Y. Lee and H. Cook, "riscv-torture," https://github.com/ucb-bar/riscv-torture, 2015.

[35] T. Li, H. Zou, D. Luo, and W. Qu, "Symbolic simulation enhanced coverage-directed fuzz testing of rtl design," in *International Symposium on Circuits and Systems*, 2021, pp. 1–5.

[36] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, and P. Cheng, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *USENIX Security*, 2021.

[37] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018, pp. 973–990.

[38] LLVM, "libfuzzer," https://llvm.org/docs/LibFuzzer.html\#corpus, 2021.

[39] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[40] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing cpu emulators," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 261–272.

[41] Microsoft, "onefuzz," https://github.com/microsoft/onefuzz, 2020.

[42] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the Symposium on Operating Systems Principles*, 2015, pp. 361–377.

[43] MITRE, "Hardware design CWEs," https://cwe.mitre.org/data/definitions/1194.html, 2019.

[44] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, "Abstraction techniques for validation coverage analysis and test generation," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 2–14, 1998.

[45] S. K. Muduli, G. Takhar, and P. Subramanyan, "Hyperfuzzing for soc security validation," in *Proceedings of the International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[46] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.

[47] G. Nativ, S. Mittennaier, S. Ur, and A. Ziv, "Cost evaluation of coverage directed test generation for the ibm mainframe," in *Proceedings International Test Conference*, 2001, pp. 793–802.

[48] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor, "Blackparrot: An agile open-source risc-v multicore for accelerator socs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.

[49] A.-R. Sadeghi, J. Rajendran, and R. Kande, "Organizing the world's largest hardware security competition: Challenges, opportunities, and lessons learned," in *Proceedings of the Symposium on VLSI*, 2021, pp. 95–100.

[50] O. Sahin, A. K. Coskun, and M. Egele, "Proteus: Detecting android emulators from instruction-level profiles," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 3–24.

[51] SIEMENS, "Modelsim," https://eda.sw.siemens.com/en-US/ic/modelsim/, 2022.

[52] W. Snyder, "Verilator, a Verilog/Systemverilog simulator and compiler," https://www.veripool.org/verilator/, 2018.

[53] R.-V. Software, "Spike RISC-V ISA Simulator," https://github.com/riscv-software-src/riscv-isa-sim, 2019.

[54] G. Squillero, "Microgp—an evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 247–263, 2005.

[55] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, "A functional validation technique: biased-random simulation guided by observability-based coverage," in *Proceedings of the International*

*Conference on Computer Design: VLSI in Computers and Processors*, 2001, pp. 82–88.
[56] E. Technologies, "Dromajo - Esperanto Technology's RISC-V Reference Model," https://github.com/chipsalliance/dromajo, 2019.
[57] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," *arXiv preprint arXiv:2102.02308*, 2021.
[58] J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Usenix Security Symposium*, 2018.
[59] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *Design Automation Conference*, 2005, pp. 783–788.
[60] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," in *USENIX Security Symposium*, 2021, pp. 1415–1432.
[61] R. Wojtczuk, "PV Privilege Escalation," https://lists.xen.org/archives/html/xen-announce/2012-06/msg00001.html, 2012.
[62] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/, 2014.