DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs

Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, Byoungyoung Lee*

Department of Electrical and Computer Engineering
Seoul National University
{hurjaewon, sshkeb96, dongup, ebaek, jangwoo, byoungyoung}@snu.ac.kr

***Abstract*—Security bugs in CPUs have critical security impacts to all the computation related hardware and software components as it is the core of the computation. In spite of the fact that architecture and security communities have explored a vast number of static or dynamic analysis techniques to automatically identify such bugs, the problem remains unsolved and challenging largely due to the complex nature of CPU RTL designs.**

This paper proposes DIFUZZRTL, an RTL fuzzer to automatically discover unknown bugs in CPU RTLs. DIFUZZRTL develops a register-coverage guided fuzzing technique, which efficiently yet correctly identifies a state transition in the finite state machine of RTL designs. DIFUZZRTL also develops several new techniques in consideration of unique RTL design characteristics, including cycle-sensitive register coverage guiding, asynchronous interrupt events handling, a unified CPU input format with Tilelink protocols, and drop-in-replacement designs to support various CPU RTLs. We implemented DIFUZZRTL, and performed the evaluation with three real-world open source CPU RTLs: OpenRISC Mor1kx Cappuccino, RISC-V Rocket Core, and RISC-V Boom Core. During the evaluation, DIFUZZRTL identified 16 new bugs from these CPU RTLs, all of which were confirmed by the respective development communities and vendors. Six of those are assigned with CVE numbers, and to the best of our knowledge, we reported the first and the only CVE of RISC-V cores, demonstrating its strong practical impacts to the security community.

I. INTRODUCTION

CPU security bugs critically damage all the computation-related hardware and software units. Due to the bug, CPUs may produce a wrong computational result, freeze the execution, reboot the computer, or allow unprivileged users to access privileged data. One unique and critical challenge of CPU bugs is that unlike software security bugs, it is extremely difficult to deploy the patch as CPUs are hard-wired circuits which cannot be re-wired once manufactured.

By far, many serious CPU bugs have been discovered. Focusing on the cases in open source CPUs, it is reported that OpenSparc had 296 bugs [1]. Proprietary CPUs such as Intel CPUs also suffered from the CPU security bugs. The Pentium FDIV bug [2] returned incorrect binary floating point results when dividing a number, which costed Intel 475 million dollars to replace the flawed processors [3]. More recently, a group of security researchers discovered multiple security vulnerabilities related to CPU's speculative execution—Spectre, Meltdown, SPOILER, Foreshadow, MDS [4–7]. We note there have been many more CPU security bugs other than those: Pentium F00F bug, which rebooted the computer upon executing a certain instruction; Intel SGX Bomb, which rebooted the computer upon intentionally violating memory integrity; and Intel TSX-NI bug, in which a detail of the bug is unknown but Intel disabled TSX through the microcode update [8, 9].

There have been tremendous efforts in automatically identifying CPU RTL bugs through static or dynamic analysis techniques [10–13], but the problem remains unsolved and still challenging largely due to the complex nature of CPU RTL designs. In particular, RTL designs implement complex sequential and combinational logics of hardware circuits, which in fact materializes the finite state machines (FSM). Thus, the general goal of RTL bug finding is to exhaustively explore as many states in FSM as possible. However, such an exploration either through static or dynamic techniques is challenging mostly because there are too many states to be covered.

This paper proposes DIFUZZRTL, an RTL fuzzer specifically designed to discover CPU RTL vulnerabilities. The core ideas behind DIFUZZRTL can be summarized with following two approaches: a dynamic testing approach and a differential testing approach, both of which help DIFUZZRTL to efficiently find RTL bugs. First, DIFUZZRTL takes a dynamic testing approach, particularly the coverage-guided fuzzing, so as to comprehensively explore hardware logics embodied in the RTL design. Second, DIFUZZRTL takes a differential-testing approach to clearly identify an RTL vulnerability. In other words, DIFUZZRTL keeps comparing an execution result of an RTL design with that of a golden model (i.e., an ISA-level simulation result), thus detecting the bugs at ISA level.

We find that realizing aforementioned ideas involve several challenges, particularly related to inherent characteristics of RTL designs. The first challenge is that DIFUZZRTL needs a new execution coverage metric tailored for RTL designs. A multiplexer in RTL designs may seem to be a good choice for coverage metrics, because it is similar to branches in software code. However, we find that the multiplexer-based coverage proposed by the state-of-the-art RTL fuzzer [14], has critical limitations due to following two reasons: 1) cycle-accurate natures of RTL circuit designs and 2) a vast number of multiplexers in a circuit. We also confirmed through our evaluation §VI that the above reasons clearly impose two

* Corresponding author

limitations on the fuzzer: 1) it was not able to correctly capture the states due to cycle-insensitivity; and 2) it has scalability limitations due to complex multiplexer wiring.

The second challenge is that DIFUZZRTL needs a systematic way to explore all possible input spaces of RTL designs. Conventional software fuzzing typically assumes the one-dimensional input space (i.e., file input space to fuzz user programs, or system call input space to fuzz kernels). However, CPU RTLs accept multi-dimensional inputs in the form of raw bus packets, so called stimuli, which is sent from various controllers—memory controllers (as a response to memory read/write requests), interrupt controllers (raising an interrupt request), etc. Worse yet, these packets are delivered to CPU RTLs in every clock cycle, further complicating the input space that CPU RTLs take.

DIFUZZRTL addresses above two challenges with following design features. First, DIFUZZRTL's coverage measurement is based on a control register, which is a register whose value can be used for any muxes' control signal. Then DIFUZZRTL measures the control register value every clock cycle, thereby making it clock-sensitive and correctly capture the explored states by RTL designs. Moreover, since a single control register is connected to multiple muxes' control signals, the number of control registers is far less than the number of wires connected to the mux's control signals, addressing the scalability limitation of mux-based coverage as well. We highlight that DIFUZZRTL was able to fuzz even a complex out-of-order machine which has about twenty thousands of lines of implementation complexity. Second, DIFUZZRTL provides systematic mechanisms to test a newly designed input format for CPU RTLs, SimInput. SimInput includes full-fledged information to run CPU RTLs, from memory address and value pairs to interrupt signals, and it is automatically translated into bus protocols that CPU RTLs are accepting. In order to execute CPU RTLs as specified by SimInput, DIFUZZRTL works as a pseudo SoC for the CPU, which includes a memory unit and an interrupt controller inside.

We implemented DIFUZZRTL as a full-fledged fuzzing framework for CPU RTLs. DIFUZZRTL automatically instruments a given CPU RTL to measure the register coverage, then keeps running two simulators, ISA and RTL simulators, while providing an identical input to both simulators. After each run, DIFUZZRTL cross-checks the architectural states, and if it identifies the difference, DIFUZZRTL automatically reports such an input as a potential bug. In order to demonstrate its strong practical aspect, we implemented DIFUZZRTL to support three real-world CPU RTLs: OpenRISC Mor1kx Cappuccino, RISC-V Rocket Core, and RISC-V Boom Core, which are widely used for academic researches as well as industry production. We note these CPU RTLs include not only simple in-order pipelined cores but also complex out-of-order superscalar cores.

During the evaluation, DIFUZZRTL identified total of 16 new bugs in those CPU RTLs, all of those are confirmed by the respective development communities or vendors. More importantly, six bugs of those are assigned with CVE numbers, signifying its practical impacts to the security community. To the best of our knowledge, DIFUZZRTL reported the first and only CVE vulnerabilities of any RISC-V cores. DIFUZZRTL has demonstrated the wide testing coverage with respect to the bug types in CPU RTLs, including atomic operation, instruction decoding, and even the performance bugs. In particular, DIFUZZRTL identified the vulnerability from the RISC-V boom core, which is similar to the notorious Pentium FDIV vulnerability, thereby helping to avoid unfortunate CPU recall cases that Intel experienced before. Particularly comparing the fuzzing performance of DIFUZZRTL against RFuzz [14] (i.e., the state of the art RTL fuzzer), DIFUZZRTL showed significantly better performances. In terms of execution speed, DIFUZZRTL is 40 times faster than RFuzz to run CPU, and in terms of states exploration efficiency, DIFUZZRTL is 6.4 times faster than RFuzz to identify a vulnerable state.

II. BACKGROUND

In this section, we provide a brief background of RTL verification and the concept of coverage-guided fuzzing, which has largely succeeded in software testing. Finally we introduce and argue the benefits of adopting coverage-guided fuzzing to RTL verification.

A. CPU Development and Testing

CPU Development. In general, modern CPUs are developed with the following two phases: 1) modeling a CPU architecture, called Instruction Set Architecture (ISA); and 2) implementing a microarchitecture with Register-Transfer level (RTL) abstraction, which follows the ISA. In the first phase, Instruction Set Architecture (ISA) is defined, which dictates an architectural level of inputs and outputs, as well as describing desired operational behaviors to generate an output from a given input. In particular, ISA defines how the programmer-visible states (e.g., registers and memory states) are updated in response to executing a well-formatted instruction.

Based on the ISA, a microarchitecture is designed in Register-Transfer level (RTL) abstraction, materializing the conceptual ISA model into a real hardware design. RTL can be expressed with various hardware description languages such as Verilog or VHDL [15, 16], which can be synthesized into a hardware circuit. Since ISA does not dictate the implementation details (e.g., the pipeline depth, cache size), there can be various microarchitectures for the same ISA, each of which has its own unique RTL implementation characteristics. For instance, although both Rocket and Boom cores implement the same RISC-V ISA, the former is an in-order and the latter is an out-of-order core with different pipeline stages.

During the development cycle, CPU should be thoroughly tested from many different aspects, including functionality, performance, security, etc. Particularly focusing on the dynamic testing techniques (we discuss static testing techniques in §VIII), such a testing can be performed with either an ISA simulation or an RTL simulation, as we describe next in turn.

Testing with ISA Simulation. An ISA simulator is a software-only implementation, which simulates all the ISA-

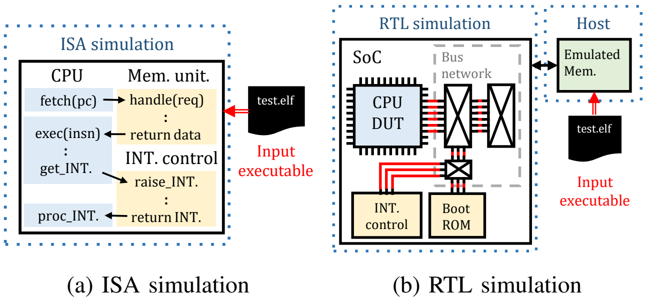


Fig. 1: Framework of ISA simulation and RTL simulation. RTL simulation requires SoC to run CPU RTL design with an executable file.

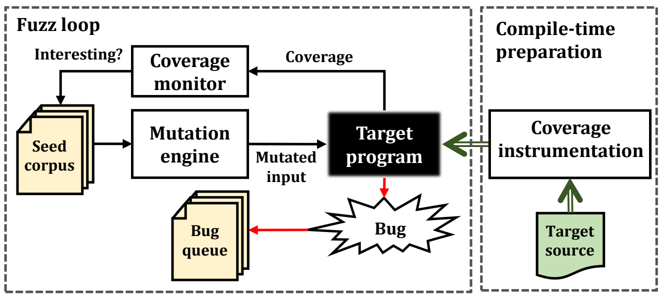


Fig. 2: A general workflow of coverage-guided fuzzing techniques

level operational behaviors (illustrated in Figure 1-(a)). To be more specific, ISA simulators mimic the behavior of the CPU and it maintains all the architectural registers and memory states as its internal value while executing instructions. Thus ISA simulator has its own memory unit, interrupt controller and other components as well as the CPU implementation. Using this ISA simulator, developers can test if the new ISA can well support various software stacks running on it without any issues. Moreover, the simulator can be used as a reference model for the architecture, manifesting how programmer-visible states should be updated if running a specific instruction.

Testing with RTL Simulation. RTL simulation is used to simulate the real-time behaviors of the design implemented in RTL. The major difference between RTL and ISA simulation is that the RTL simulation is aware of a cycle concept, representing a clock cycle of the synchronous circuit. Thus the cycle accurate behaviors of the design including microarchitectural states are tested during the simulation, which cannot be performed with the ISA simulation.

However, the RTL design alone cannot be simulated since the design is just a representation of a circuit for the CPU. To operate the RTL design, input stimuli should be provided to the ports of the design while the simulation. Thus, an SoC including memory units and interrupt controller is implemented to feed input stimuli on the CPU design. Thus, the SoC completes the CPU design for RTL simulation to run a meaningful software code on it. Figure 1-(b) shows a simple SoC to test CPU designs. Before the simulation, an input executable file is loaded to the emulated memory in host and the CPU in RTL simulation runs the executable. In the simulation, SoC continuously receives data including instructions from the emulated memory and generates input stimuli for the CPU design. Upon receiving the input, the CPU RTL design runs the intended instructions.

B. Fuzzing

Coverage-Guided Fuzzing. Fuzzing is a software testing technique, which keeps running a target program with randomly generated (or mutated) inputs so as to discover previously unknown vulnerabilities. In particular, coverage-guided fuzzing [17, 18] is arguably the most popular fuzzing technique, which focuses on extending the execution coverage. In each run of the target program, it measures the execution coverage at runtime, and leverage the measured coverage as feedback to generate (or mutate) the next input to be tested. In order to measure the coverage, it is assumed that the target program is instrumented beforehand such that the fuzzer can collect certain coverage information at runtime (including a basic block coverage, edge coverage, etc.)

For instance, Figure 2 shows a typical workflow of coverage-guided fuzzers. The fuzzer starts by randomly choosing one of the input from the input corpus, which maintains a set of interesting inputs. Then this chosen input is randomly mutated (such as a random bit flip, merging two bytes in random offsets, replacing with a specific value, etc.). Next, the fuzzer runs the target program with the mutated input, while measuring the execution coverage of the mutated input. If this mutated input covers the new execution coverage that were not explored before, it is saved back to the input corpus so that it can get another chance to be fuzzed in the future running. If not, the mutated input is thrown away. The fuzzer repeats aforementioned steps indefinitely, which results in coverage-guided fuzzing because it is more likely to fuzz the input that is more likely extending the coverage.

Differential Fuzz Testing. Most fuzzers have focused on identifying memory corruption bugs [17–34], mostly because 1) it has strong security implications; 2) memory violation conditions are relatively easy to define and thus relatively easy to detect. On the other hand, fuzzing to find semantics bugs, which identifies logical vulnerabilities deviating from developer-intended program behaviors, are not well explored compared to finding memory corruption bugs. In general, semantic bugs are known to be difficult to find because it is difficult to express semantic violation cases into well-formed safety violation conditions, because many of those require specific domain knowledge of target programs. To tackle this problem, previous works introduced differential fuzzing techniques, where the fuzzer identifies a bug by comparing the output of multiple programs of the same purpose [35, 36]. In fact, such differential testing techniques are also used for RTL verification as well, particularly comparing one RTL's execution results with a golden model's execution results [37], which inspired the design of DIFUZZRTL.

III. Motivation

The major motivation for DIFUZZRTL is to design a fuzzing framework considering unique characteristics of RTL designs. To this end, this section discusses two issues to design the RTL fuzzer, coverage definitions for RTL fuzzing (§III-A) and input space for RTL Fuzzing (§III-B).

A. Coverage Definition for RTL fuzzing

Example: A Memory Controller. Suppose a developer wants to develop a simple memory controller, which connects CPU with SDRAM and flash memory. Considering the unique hardware characteristics, the memory controller takes 8-bits from the flash at once. However, it takes 4-bits from the SDRAM since SDRAM transfers only 4-bits per cycle. Thus, the memory controller should assemble two data packets from SDRAM to forward the entire 8-bits [38].

To this end, the developer designs following two independent FSMs as shown in Figure 3-(a). First, the FSM for the flash begins with the ready state ($\mathtt{R}_\mathrm{F}$). If the valid signal is one, it transitions to the busy state ($\mathtt{B}_\mathrm{F}$) while taking the 8-bits from the flash. Then it goes back to the ready state. Second, the FSM for the SDRAM is similar to that of the flash, but the key difference is that it has one more state in the middle, the pending state ($\mathtt{P}_\mathrm{S}$). This is because since SDRAM sends 8-bits of data with two consecutive 4-bits of transmissions, the controller should maintain two states (i.e., $\mathtt{P}_\mathrm{S}$ and $\mathtt{B}_\mathrm{S}$) to represent the completion of the first- and second-half transmission, respectively.

Based on these two FSMs, the sequential circuits using RTL can be implemented for the memory controller as in Figure 3-(b). For simplicity, we omitted the data flow in this illustration. When implemented with RTL, the current states are maintained with state register variables (i.e., $\mathtt{state}^\mathrm{F}$ and $\mathtt{state}^\mathrm{S}$), because the state is later used for determining the next state. Moreover, a state transition is implemented with a multiplexer (i.e., `mux`), because `mux` outputs an appropriate input according to the select signal (i.e., the state transition condition in the FSM can be represented with the select signal of `mux`).

The sequential circuit for the flash has 1-bit register, $\mathtt{state}^\mathrm{F}$, where 1'b0 (i.e., the bit value 0) represents $\mathtt{R}_\mathrm{F}$ and 1'b1 (i.e., the bit value 1) does $\mathtt{B}_\mathrm{F}$, respectively. It is assumed that $\mathtt{state}^\mathrm{F}$ is first initialized with $\mathtt{R}_\mathrm{F}$. When $\mathtt{valid}_F$ is asserted (represented with ①), the mux $\mathtt{M}_0^F$ forwards $\mathtt{B}_\mathrm{F}$ from the two inputs (i.e., $\mathtt{state}^\mathrm{F}$ and $\mathtt{B}_\mathrm{F}$). Next, $\mathtt{M}_1^F$ forwards the output of $\mathtt{M}_0^F$, which is $\mathtt{B}_\mathrm{F}$ (②), since the select signal of $\mathtt{M}_1^F$ is $\mathtt{R}_\mathrm{F}$. Then $\mathtt{state}^\mathrm{F}$ is updated with $\mathtt{B}_\mathrm{F}$ (③), completing the first clock cycle. In the next clock cycle, the circuit is processed when $\mathtt{state}^\mathrm{F}$ is $\mathtt{B}_\mathrm{F}$, updating $\mathtt{state}^\mathrm{F}$ with $\mathtt{R}_\mathrm{F}$ in the end of the clock cycle.

The sequential circuit for the SDRAM has 2-bit register, $\mathtt{state}^\mathrm{S}$, because it has three states to be represented: 2'b00 for $\mathtt{R}_\mathrm{F}$, 2'b01 for $\mathtt{P}_\mathrm{S}$, and 2'b11 for $\mathtt{B}_\mathrm{S}$. Thus, it has one extra `mux` to implement an extra state transition, but it is largely similar to the sequential circuit for the flash.

Vulnerability in the Memory Controller. This memory controller has a vulnerability breaking the memory consistency, which is related to the constraint that the memory controller can only forward 8-bits (sent from either flash and SDRAM) to CPU per cycle. If both flash and SDRAM completes the 8-bit transmission at the same clock cycle (i.e., reaching $\mathtt{B}_\mathrm{F}$ and $\mathtt{B}_\mathrm{S}$ at the same clock cycle), the memory controller cannot handle both. In other words, it can only forward one 8-bits transmission, and should drop the reset transmission. As a result, one of the data (transmitted by either flash or SDRAM) will be lost, thus breaking the memory consistency.

This vulnerability cannot be captured with two individual FSMs that we presented before (Figure 3-(a)), which assumes that the flash and SDRAM operations are independent to each other. However, since these two are in fact dependent with respect to the memory controller, two individual FSMs should be merged into a single FSM where its states are a product of all states (Figure 3-(c)). As shown in the figure, once the memory controller reaches the state ($\mathtt{B}_\mathrm{F}$, $\mathtt{B}_\mathrm{S}$), then data loss or corruption occurs.

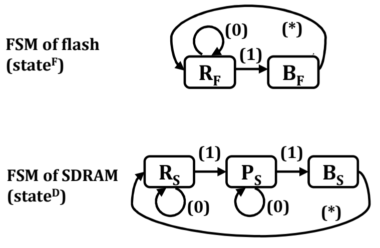
In order to fix this vulnerability, the memory controller should handle only one of two transmissions during the vulnerable clock cycle, and the other should be handled in the next clock cycle. Thus, the developer should patch with an extra state transition, from ($\mathtt{B}_\mathrm{F}$, $\mathtt{B}_\mathrm{S}$) to ($\mathtt{R}_\mathrm{F}$, $\mathtt{B}_\mathrm{S}$) or ($\mathtt{B}_\mathrm{F}$, $\mathtt{R}_\mathrm{S}$)

Limitation of Previous Fuzzing Approaches. In order to identify this vulnerability, various approaches can be used, but each approach has its own limitation.

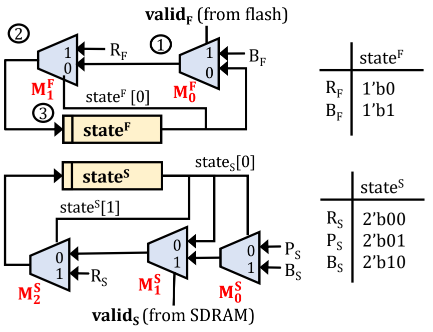
Focusing the discussion on fuzzing techniques, RFuzz [14] proposed the mux-coverage guided fuzzing technique. The core idea behind this technique is that the mux's select signal leads to a state transition, so guiding the fuzzing based on the `mux`'s select signal would lead to exploring more FSM states. To be more specific, this technique runs the sequential circuit while monitoring all the select signals of the muxes, identifying which of those were toggled in the end of running. If any mux were newly toggled, the provided input to the circuit (i.e., a series of per-cycle $\mathtt{valid}_F$ and $\mathtt{valid}_S$ signals) is considered as a valuable input and thus added to the corpus. This is because such a new toggling indicates that the new state transition has been explored by the input. If no muxes were newly toggled, the provided input is simply thrown away.

We observe two critical limitations of RFuzz's `mux`-coverage technique. The first limitation is that since `mux`-coverage metric is clock-insensitive, it cannot precisely capture FSM state transitions. In other words, it does not recognize interplay and inter-dependency between mux toggling events across clocks, so semantically different mux toggling events are considered as the same, failing to correctly identify state transitions.

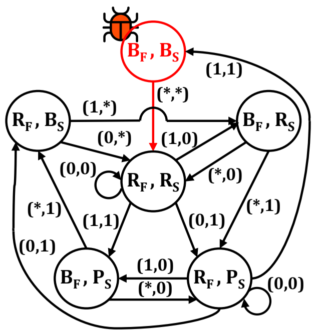
For instance, Figure 4 illustrates two different cases of running the memory controller, where the left case runs with the benign input (i.e.,($\mathtt{R}_\mathrm{F}$, $\mathtt{R}_\mathrm{S}$) to ($\mathtt{B}_\mathrm{F}$, $\mathtt{P}_\mathrm{S}$) and reaching ($\mathtt{R}_\mathrm{F}$, $\mathtt{B}_\mathrm{S}$)) and the right case runs with the vulnerability-triggering input (i.e., the input leads to reaching the state, ($\mathtt{B}_\mathrm{F}$,$\mathtt{B}_\mathrm{S}$), at `clk3`). For each case, all muxes' select signals are shown per cycle, where the mux toggling is highlighted with the red-colored box. In the end of running, the coverage map is generated which sums up all the observed toggling events. Then this coverage map is used to identify if new mux toggling is triggered by the



(a) Two independent FSMs designed for the example memory controller, where each FSM is for $\texttt{state}^F$ and $\texttt{state}^S$.



(b) Schematics of the example memory controller. For simplicity, the data flow is abstracted out.



(c) The combined FSM of the example memory controller. The red colored state, ($\texttt{B}_F$, $\texttt{B}_S$) denotes a vulnerable state.

Fig. 3: FSMs and schematic for example memory controller design.

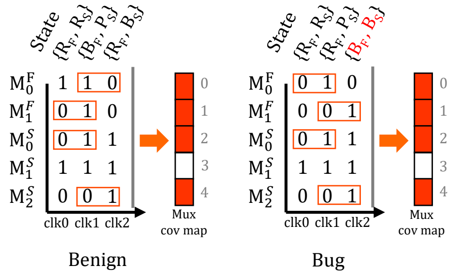


Fig. 4: The workflow of RFuzz's mux coverage schemes for benign (left) and bug triggering inputs (right)

provided input. However, although benign and vulnerability-triggering cases are clearly reaching different states, coverage maps of those are the same. This is because although the mux toggling of $\texttt{M}^F_0$ and $\texttt{M}^F_1$ takes place at different clock cycles, RFuzz's mux-coverage metric cannot capture such differences. As a result, RFuzz would not be able to properly guide the fuzzing procedure towards exploring more states as its metric collapses multiple states into one state.

The second limitation is related to instrumentation overhead of monitoring all muxes' select signal. From the implementation perspective of the RFuzz's mux-coverage approach, the required resources (e.g, wires and registers) for instrumentation quadratically increases as the number of muxes increases, critically limiting its runtime performance as well as scalability.

Our Approach: Register Coverage for RTL. In order to overcome the limitation, DIFUZZRTL proposes the register-coverage approach for RTL. While we provide details in §IV-C, register coverage can be summarized with two key features. First, it supports clock-sensitive coverage, so it can precisely capture FSM state transitions. Second, its measurement is based on control registers, not based on muxes' control signals, making it efficient and scalable. As we evaluate further in §VI, DIFUZZRTL's register-coverage has shown 40 times better execution speed, and 6.4 times faster vulnerable state exploration time compared to RFuzz's mux-coverage. More importantly, DIFUZZRTL was able to fuzz all three real-world RTLs including out-of-order Boom Core, while RFuzz was not able to fuzz the Boom Core due to the scalability issues.

B. Input Space for RTL Fuzzing

Limitations of CPU Testing using Entire SoC. As mentioned in §II-A, CPU designers have used SoC to simulate and test the CPU RTL designs. Leveraging entire SoC enables the comparison between ISA simulation and RTL simulation, i.e., end-to-end test, by making them take the same executable as an input. In this sense, fuzzing entire SoC can simplify the fuzzer by concerning only executable generation, but there are some fundamental limitations.

First of all, the fuzzer cannot test the entire input space of the CPU design. CPU RTL designs have several input ports including ports for data transfer, interrupt and debug interface. SoC wraps this interface by converting input from outer world (e.g., executable) into a formatted input stimuli the CPU can interpret. However, it also means limiting the input space into the space that SoC can only generate. To test various functioning of CPU such as responses to stressful cache coherence transactions or arbitrary interrupts, the intended SoC should be redesigned each time. In contrast, fuzzer which directly channels the CPU input space can generate input adaptively.

Furthermore, as the open source hardware [39] becomes popular, it is no longer true that the CPU and SoC are designed and implemented by a single vendor. The open sourced CPU RTL design can be used in various SoC designs, e.g., using Boom core in Rocket SoC [40, 41]. Thus, the CPU and SoC should be verified separately and we need a unified platform to test CPU designs only.

IV. Design

Now we present the design of DIFUZZRTL. We first introduce how DIFUZZRTL performs the mutation, which generates a new CPU input format, SimInput (§IV-A). Then we describe ISA simulation §IV-B, which accepts SimInput as input. Next, we illustrate how DIFUZZRTL compiles RTL designs to support register coverage (§IV-C), and then explain

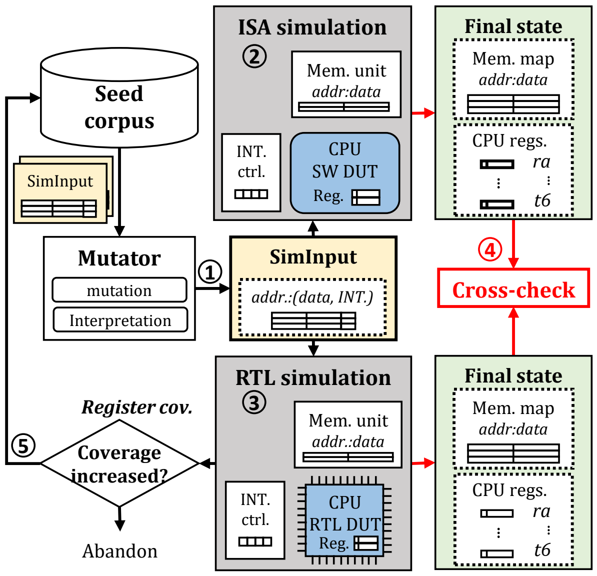


Fig. 5: An overall framework of DIFUZZRTL. It runs both ISA and RTL simulations using the same SimInput, and cross-checks the both execution results' after each run. The entire workflow leverages the register-coverage guiding, so the RTL simulation measures the register-coverage which is provided for the mutator.

RTL simulation (§IV-D). This RTL simulation also accepts the identical SimInput as the ISA simulation. Lastly, we describe how DIFUZZRTL cross-checks the execution results from ISA and RTL simulations to finally identify bugs (§IV-E).

Overall Workflow. The overall framework of DIFUZZRTL is shown in Figure 5. First, mutator randomly generates input (①), and DIFUZZRTL runs both ISA simulation and RTL simulation using the input (②, ③). After the simulations end, DIFUZZRTL takes a snapshot of the final memory states and architectural registers of both designs to cross-check (④). Thus, DIFUZZRTL finds a potential bug if execution results are different. Above four steps make one fuzz iteration, and such an iteration keeps repeated indefinitely while the input mutation and selection is guided to increase register-coverage (⑤).

A. SimInput Mutation

In order to provide a consistent and identical input to both SW and RTL designs, DIFUZZRTL defines a new input format for CPU, SimInput. SimInput thoroughly includes all input space that CPU takes, ranging from all address and value (both code and data) to associated interrupt events. Figure 6 shows a simple example of SimInput.

For a given SimInput, DIFUZZRTL randomly mutates it with following two phases: 1) DIFUZZRTL enumerates all instructions in SimInput and performs per-instruction mutation, which determines opcode and operands of the instruction; After mutating all instructions, DIFUZZRTL starts interpreting instruction by instruction, which populates all remaining information of SimInput (including an address of an instruction, address and value of data, and a list of interrupts).

```
0x100 : (0130071b | addiw   a4, zero, 0x13  INT.: 0000->0000)
0x104 : (01c0036f | jal     t1, pc + 0x1c   INT.: 0000->0100->0000)
0x120 : (02e32823 | sw      t4, 0x30(t1)    INT.: 0100->0100->0000)
0x300 : (3943648f | unknown                 INT.: 0000)
0x310 : (064ff13b | unknown                 INT.: 0000)
```

Fig. 6: Input generated by mutator. Each address is associated with an instruction and a list of interrupt events

It is worth noting that, ISA and RTL simulation runs CPU according to SimInput as follows: 1) The map of address and value pair is used as an initial memory state for both SW and RTL designs. 2) The associated interrupt event is also accordingly raised when designs execute the corresponding instruction. Since CPU is operating based on the initial memory layout and interrupt events, SimInput ensures the deterministic execution on both designs.

Per-Instruction Mutation. In general, DIFUZZRTL's per-instruction mutation is a grammar-aware and structured mutation approach while allowing some randomness, such that a mutated instruction is likely a valid instruction (but an invalid instruction with a low probability). This ensures that when the instruction is executed by SW or RTL design, it would not always be rejected due to the invalid instruction format. More specifically, per-instruction mutation determines following two fields: opcode and operands. First, the opcode of the instruction is determined at random from the list of valid opcodes in the ISA specification.

Next, the operands (including register indices and immediate/address values) are determined at random. Note that DIFUZZRTL's mutator keeps track of which register indices and immediate/address values were assigned before, and attempt to reuse such indices and values. Thus, this would increase the data dependency between instructions, thereby allowing DIFUZZRTL to stress test the design. (e.g., on a corner implementation case of SW design or data/control hazard detection or resolution logics in RTL design).

Regarding the space complexity, DIFUZZRTL requires two pools of variables (i.e., used registers and immediates) per mutation. Each pool can have variables up to the length of generated instructions, thus the space complexity is $O(N)$ where N is the number of instructions. DIFUZZRTL requires a predefined set of opcodes and register indices but they are statically determined before the fuzzing.

Interrupt Mutation. After the instruction mutation, the sequence of interrupts is mutated. As in the instruction case, mutator randomly appends or deletes interrupts in the given sequence. Then, the generated interrupt sequence is paired with the instruction sequence. In the simulation, the paired interrupt values are injected every execution of the corresponding instructions.

Population through Instruction Interpretation. After mutating all instructions and interrupts, DIFUZZRTL populates all remaining information, i.e., map of initial address to instructions, data and interrupts, to SimInput. However, directly placing all the instructions sequentially in a defined memory region would end in meaningless executions since the prepared instructions will not be executed after control flow changes. Thus, the population is performed through instruction interpretation, as it necessarily requires to understand how the instruction would be executed at runtime.

To be more specific, DIFUZZRTL first determines the address of the first instruction to be a given entry symbol, and interprets the instruction (i.e., executes the instruction). During the interpretation, if the instruction attempts to load the data from `addr` where `addr` is not specified in SimInput, DIFUZZRTL provides a random value `v` while populating SimInput with (`addr`, `v`). Note that, if the instruction attempts to store data, DIFUZZRTL would not update SimInput as it is not part of the input to the CPU. Moreover, an interrupt event is pushed to the interrupt event list of the current PC. Each interrupt event can be either None (i.e., the interrupt signal should not be asserted) or the interrupt signal value (i.e., the interrupt signal should be asserted with the specified IRQ number).

After finishing the interpretation of the first instruction, the next PC value will be determined (i.e., a target address if it is a branch instruction, or PC+4 otherwise). Then DIFUZZRTL updates SimInput so that the next instruction (i.e., a next mutated instruction) has the next PC address, and start interpreting this new instruction.

B. ISA Simulation

DIFUZZRTL's ISA simulator runs CPU SW design as instructed by SimInput. To this end, DIFUZZRTL tailors how CPU SW design takes input from other components including a memory unit and an interrupt controller. In the case of the memory unit, DIFUZZRTL populates the initial memory layout by embedding SimInput in the base memory template. Then, DIFUZZRTL loads the initial memory layout to the ISA simulation.

In the case of the interrupt, DIFUZZRTL implements a pseudo interrupt controller in the ISA simulator, which raises an interrupt when CPU SW design executes a specific PC. More specifically, the controller raises the interrupt value which is paired to the instruction pointed by PC.

The simulation continues until the CPU SW design reaches a specific address (i.e. the end of the execution) which is embedded in the base memory template. The SimInput population ensures that the control flow always converges to the end of the execution. The base memory template also contains instructions to save achitectural register-files to the specific memory address so that DIFUZZRTL can take the snapshot of programmer-visible states (i.e. memory and registers) by reading specified memory addresses. The snapshot is later used to cross-check with the RTL simulation generated one.

C. RTL Compilation with Register Coverage

As described in §III-A, a mux-coverage technique has two limitations: 1) it is clock-insensitive, it cannot precisely capture FSM state transitions; and 2) it imposes huge instrumentation costs, critically slowing down the runtime performance of simulation as well as limiting its scalability.

Algorithm 1 Algorithm of DIFUZZRTL's control register identification

```
Input: $G = (V, E)$, graph of nodes (muxes, wires, registers, ports)
    parsed from HDL source code
Output: $C$, set of all control registers
 1: for each $m : Mux \in V$ do
 2:     $C_M \leftarrow findSrcRegs(m, \phi, \phi)$
 3:     $C \leftarrow C \cup C_M$
 4: return $C$
 5:
 6: function FINDSRCREGS(m, S, T)
 7:     if $m \in register$ then
 8:         $S \leftarrow S \cup \{m\}$
 9:     else if $m \notin \{port \cup T\}$ then
10:         $T \leftarrow T \cup \{m\}$
11:         for each $(m', m) \in E$ do
12:             $S \leftarrow S \cup findSrcRegs(m', \phi, T)$
13:     return $S$
```

In order to overcome such limitations, DIFUZZRTL proposes a new coverage metric: register-coverage. DIFUZZRTL's register-coverage metric has two key features: 1) it is based on control registers, not based on muxes' control signals, making it performance efficient and scalable; and 2) it is clock-sensitive coverage (i.e., measures the coverage every clock cycle), so it can precisely capture FSM state transitions. In the following, we first describe how DIFUZZRTL identifies control registers in RTL through a static analysis, and then describe how DIFUZZRTL measures clock-sensitive register-coverage at runtime.

Identifying Control Registers. DIFUZZRTL's coverage measurement focuses on monitoring value changes in a control register—a register where its value is used as any muxes' control signal. In other words, since value changes of control registers would lead to the FSM state transition, it can also be used to explore more FSM states for fuzzing. Since a single control register is often connected to multiple muxes' control signals, the number of control registers is far less than the number of wires connected to the mux's control signals.

The problem arising here is that RTL design has a vast number of registers, and only a small set of registers are control registers. Hence, DIFUZZRTL performs a static analysis to identify control registers. In particular, the analysis first builds a graph representing the connections between all elements (e.g., registers, wires, and muxes) in the module. Then, we recursively perform a backward data-flow analysis for each mux's control signal as shown in Algorithm 1. If the backward data-flow tracing reaches a register, then we conclude such a register is the control register. This is because this register's value will be directly or indirectly (i.e., through a combinational logic) used to control the mux's behavior. If the backward tracing either goes beyond the module boundary or reaches the already traced point (because a circuit is circular), the analysis stops.

In terms of analyzing the algorithmic complexity,

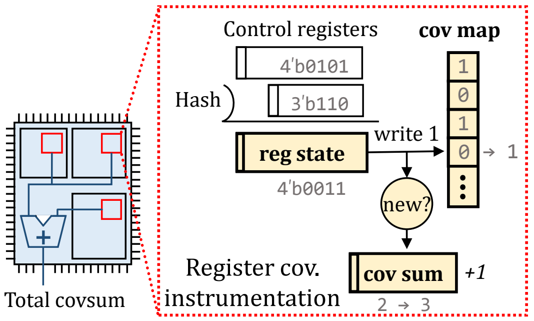


Fig. 7: A register-coverage instrumentation of DIFUZZRTL, which adds following three registers: `regstate` represents the state of the module, `covmap` remembers the reached states, and `covsum` summarize the number of reached states.

DIFUZZRTL finds all the control registers with $O(V^2 \cdot E)$ of time complexity where V is the number of elements and E is the number of connections between them. The space complexity is $O(V \cdot E)$ for managing the graph representing the connections between all the elements.

Clock-Sensitive Register Coverage. DIFUZZRTL measures the register-coverage every clock cycle. Note that this clock-by-clock coverage measurement is nearly infeasible using mux-coverage due to its performance and scalability issues. In the case of DIFUZZRTL, however, such clock-by-clock measurement became feasible with DIFUZZRTL's register-coverage technique, which we demonstrate more details in (§VI).

For each RTL module, DIFUZZRTL inserts three new registers, `regstate`, `covmap`, and `covsum` (shown in Figure 7). DIFUZZRTL instruments the module such that all the values in control registers are hashed into `regstate`. In order to implement the hash function, we used a series of `XOR` operations while each control register's value is left-shifted with a deterministic random offset. We note that this hash function design is inspired by AFL's edge-coverage metric using XOR operations [17]. Then the instrumented logic attempts to write 1 to the `covmap`'s slot, where the slot index is determined by the value of `regstate`. This write operation marks that the corresponding FSM state (i.e., a hash of control register values) has been explored by the instrumented module.

When this write operation to `covmap` takes place, DIFUZZRTL increments `covsum` only if the value in the corresponding `covmap` slot was zero. If it were already 1, it implies that the RTL module has already explored such an FSM state in the previous clock cycles. If it were zero, it implies that it has just explored the new FSM state during the current clock cycle. Since such `covmap` and `covsum` updates are carried out every clock cycle, DIFUZZRTL's register-coverage mechanism is clock-sensitive. Then this `covsum` is wired out to the parent RTL module, which sums up `covsum` values from all child RTL modules. This tree structure-like `covsum` summation is performed until reaching the top level RTL module.

The `covsum` value in the top level RTL module is used as a final coverage value throughout DIFUZZRTL's fuzzing. It is worth noting that the values in `covmap` and `covsum` are maintained during the fuzz iterations, thus the states once covered are not recognized as new in the later iterations.

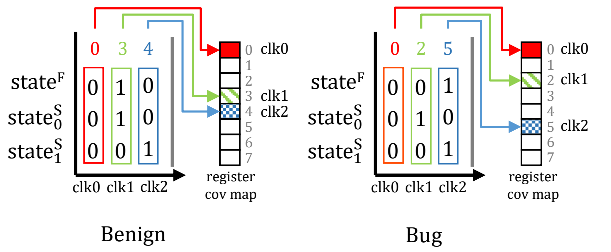


Fig. 8: The workflow of DIFUZZRTL's clock-sensitive register-coverage schemes for benign (left) and bug triggering inputs (right).

Revisiting the Memory Controller Example. Using the register-coverage, DIFUZZRTL can correctly distinguish states in the memory controller example, which were not detected by RFuzz's mux coverage (previously shown in Figure 4). Figure 8 illustrates how DIFUZZRTL handles the same example case. In each clock cycle, DIFUZZRTL computes the hash of all control registers (i.e., $\texttt{state}^F$ and $\texttt{state}^S$), which updates `covmap` every clock cycle. Since the hash of control registers is computed every clock cycle, DIFUZZRTL can correctly identify the state difference between benign and bug-triggering cases, thereby enabling DIFUZZRTL to correctly guide the fuzzing procedure towards the buggy states.

D. RTL Simulation

DIFUZZRTL's RTL simulator runs CPU RTL design as specified by SimInput. While overall simulation mechanism is similar to how DIFUZZRTL runs ISA simulator, DIFUZZRTL's RTL simulation has following two key differences. First, since CPU RTL design takes input stimuli following the specific protocol, DIFUZZRTL accordingly generates the input stimuli after interpreting SimInput. Second, an interrupt processing mechanism in RTL design is different from SW design for the following two reasons: 1) RTL design runs at a CPU cycle level while SW design runs at an instruction level; and 2) RTL design may defer when to process an asynchronous interrupt but SW design does not defer.

Stimuli Generation. In order to handle the first difference, DIFUZZRTL generates formatted stimuli based on the protocol. As in the ISA simulation, DIFUZZRTL runs as a pseudo memory unit in RTL simulation, which is initialized with the memory layout generated from SimInput. DIFUZZRTL monitors the data bus interface of RTL design and generates input stimuli which contains the data mapped to the requested address.

Interrupt Generation. In order to generate an interrupt, DIFUZZRTL designs a pseudo interrupt controller for RTL design. According to SimInput, DIFUZZRTL monitors executed PC every cycle so as to assert interrupt signal at a specific PC. One challenging issue here is unlike in the case of ISA simulation, it is difficult to control when the raised interrupt would be processed by the RTL design. Specifically, a majority of

ISAs dictates that processing of an asynchronous interrupt can be deferred [42, 43], meaning that each microarchitecture can make their own decision on when to process the asynchronous interrupt. Thus, even if DIFUZZRTL raises an interrupt when RTL design's PC has a specific address value, RTL design may not process the interrupt when executing that address value—it may process the interrupt much later. In order to address this challenge, DIFUZZRTL exhaustively attempts to find the correct cycle by re-running the RTL simulation while advancing the interrupt assertion cycle one by one from the cycle when corresponding PC is committed.

E. Checking Execution Results

Once the running of both ISA and RTL simulation for a given SimInput ends, DIFUZZRTL starts the cross-checking process. First, DIFUZZRTL checks if the control-flows of ISA and RTL simulations are the same. If both simulations correctly reach the end of execution, DIFUZZRTL reads programmer visible states of RTL design as we have done for the ISA case, and then compares it with ISA one's. If the architectural states do not match, DIFUZZRTL saves the corresponding SimInput as a potential bug since both designs should be in the same state after the same execution.

Note that the design of DIFUZZRTL relies on the ISA simulator, which implies two functional limitations: 1) The target CPU RTL design should have the ISA simulator; and 2) While DIFUZZRTL can detect ISA level bugs, it cannot detect non-ISA level bugs. The first limitation can be mitigated if two different RTL designs implement the same ISA. In this case, DIFUZZRTL can be extended to perform the differential testing between those two (e.g., Rocket and Boom cores are two different RTL designs implementing the same ISA). The second issue can be partially addressed if the designer implements manual hardware assertions checking micro-architectural contexts, which can be retrofitted by DIFUZZRTL to detect non-ISA level bugs.

Then, if new register-coverage is discovered, the corresponding SimInput is saved to the corpus so that it can get another chance to be fuzzed later, thereby DIFUZZRTL's fuzzing procedure is register-coverage guided. DIFUZZRTL repeats above steps while resetting the design every iteration.

V. IMPLEMENTATION

For DIFUZZRTL, we implemented both RTL compiler pass for register-coverage instrumentation and CPU fuzzing framework. DIFUZZRTL is open-source and available at https://github.com/compsec-snu/difuzz-rtl.

A. RTL compiler pass for register-coverage Instrumentation

We modified two different HDL processing tools: 1) `Pyverilog`, for codes written in Verilog [44]; and 2) `FIRRTL` compiler, for `FIRRTL` codes which is the intermediate language of Chisel [45]. These tools thus automatically find control registers, instrument register-coverage, then produce instrumented Verilog code as a final output. Our implementation includes 1.5 `k` lines of python code (in `Pyverilog`) and 2 `k` lines of `Scala` code (in `FIRRTL` compiler).

B. CPU fuzzing framework

CPU fuzzing framework of DIFUZZRTL runs as a testbench for the CPU RTL designs while running SimInput mutation, ISA simulation and bug checking. For SimInput mutation and other functionalities outside the RTL simulation, DIFUZZRTL consists of 4 `k` lines of Python, which include manually determined opcodes and registers for instruction generation. We also added 800 lines to the ISA simulator for instruction emulation and a pseudo interrupt controller. The framework also relies on gnu toolchains (i.e. gcc, nm) to generate simulation inputs. Then, we implemented the prototype of DIFUZZRTL on two different RTL testing environments, the software simulation and the FPGA emulation.

Prototyping for Software Simulation. We used `cocotb` [46], a python based test bench tool for RTL codes, to implement the prototype on RTL software simulation. The designs are then simulated using an RTL simulator (i.e., Verilator [47] or icarus Verilog [48]). This prototype includes 1.5 `k` lines of python codes for stimuli generation and monitoring the simulation.

Prototyping for FPGA emulation. In order to test DIFUZZRTL using FPGA emulation, we incorporated the fuzzing framework into `FireSim`, which is an FPGA-accelerated simulation platform developed by BAR [49]. Thus, DIFUZZRTL's implementation on `FireSim` automatically instruments register-coverage when building an FPGA image, thereby enabling the fuzzing framework. To this end, we modified 200 lines of `Scala` and 500 lines of `C++` codes in `FireSim`.

VI. EVALUATION

This section evaluates DIFUZZRTL on various aspects. We first describe the evaluation setup of DIFUZZRTL (§VI-A). Then we evaluate the effectiveness of register-coverage with synthetic RTL designs (§VI-B). Next, we analyze the performance of DIFUZZRTL with real-world CPU designs in two different testing environments: 1) software simulation (§VI-C); and 2) FPGA emulation (§VI-D). Then we describe the list of new bugs that DIFUZZRTL found (§VI-E), and introduce case studies of finding real-world bugs through DIFUZZRTL and other approaches (§VI-F).

A. Evaluation Setup

1) RTL Designs: In order to evaluate DIFUZZRTL, we performed the fuzz testing with various RTL designs, from synthetic RTL designs to real-world OpenRISC and RISC-V CPU cores.

Synthetic RTL. In order to clearly understand DIFUZZRTL, we developed a synthetic RTL design based on the code example shown in Figure 22. We added one more register variable from the example so that the bug is triggered when the registers reach a specific state. We also tested synthetic RTL design while varying the number of states by adding more states per each register (i.e., adding more bits to each register). In total, four different versions of RTL designs with the different number of finite states were tested: 27 , 64 , 125, and 216 states.

Real-World OpenRISC Mor1kx Cappuccino. This is a five stage pipelined core which implements the OpenRISC ISA [43]. Modules such as MMU, cache, and FPU are included in this design, building a full-fledged core. Thus, this core supports basic and floating point instruction set in OpenRISC ISA and is able to boot Linux. In order to perform the differential testing while fuzzing, DIFUZZRTL used OpenRISC Or1ksim as a golden model, which is an OpenRISC ISA simulator.

Real-World RISC-V Rocket Core. Rocket core is an in-order pipelined core which is included in RISC-V Rocket Chip [41]. This core is supported by industry for the chip prototyping. We note that Rocket core is extensively verified by the steering research group. In order to perform the differential testing while fuzzing the Rocket core, we used a RISC-V reference ISA simulator, Spike, as a golden model. Spike is commonly used to verify the correctness of new hardware designs.

Real-World RISC-V Boom Core. Boom Core is an out-of-order superscalar core which can also be used in RISC-V Rocket Chip SoC. Features for out-of-order cores such as issue queue or ROB are implemented in Boom core thus its micro architecture is much more complex than in-order cores. Boom core is also able to boot linux and widely verified by the steering research group. To perform the differential testing, we used Spike as we have done for the Rocket core.

2) Fuzz Testing Environment: We evaluated DIFUZZRTL in two different environments for testing RTL designs, i.e. software simulation and FPGA emulation.

Software Simulation. All our experiments based on software simulation were carried out on a machine of Intel Xeon Gold 6140 with 72 CPU cores and 512GB RAM, which runs Ubuntu 18.04 LTS. For synthetic RTL designs, we fuzzed each version 1,000 times and plotted a graph representing the distributions so as to come up with a more robust statistical conclusion. For real-world RTL designs, we fuzzed each RTL design with the corresponding ISA simulator for three times and plotted a graph showing an average value as well as minimum and maximum values along with an error bar.

FPGA Emulation. We ran DIFUZZRTL using FPGA emulation on an Amazon EC2 F1 instance [50], which offers a customizable hardware acceleration feature through FPGA. Similar to the case of the software simulation, we fuzzed Rocket core and Boom core three times and plotted a graph showing an average value and error bar.

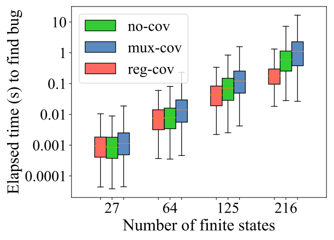
3) Coverage Guiding Setup for Fuzzing: To compare the effectiveness DIFUZZRTL's register-coverage guided fuzzing, we ran DIFUZZRTL while changing the coverage guiding feature: 1) `no-cov`, which does not leverage any coverage-guided feature; 2) `mux-cov`, which utilizes the mux-coverage guided fuzzing as proposed by RFuzz [14]; and 3) `reg-cov`, which utilizes the register-coverage guided fuzzing that we propose with DIFUZZRTL.

B. Fuzzing Synthetic RTL

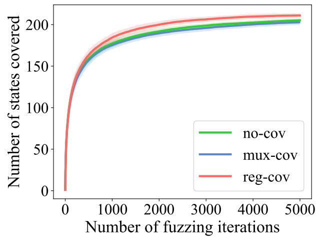
Static Instrumentation Overhead. To compare the efficiency of DIFUZZRTL's register-coverage and RFuzz's mux-coverage scheme, we instrumented each coverage scheme on the synthetic RTL designs. Figure 9 shows the statistics of each synthetic RTL design and the overheads of coverage instrumentations. The design has total six registers and 18 muxes, where only three registers are identified as control registers by DIFUZZRTL. The number of muxes that mux-coverage has to monitor linearly increases as the number of states increases, but the bits in control registers that register coverage should monitor marginally increases.

Project	Num. reg. (Bits)		Num. muxes	Verilog	Overhead (%)	
	Total	Control	Total	Original	`mux-cov`	`reg-cov`
Synthetic RTL (27)	6 (18)	3 (6)	9	123	118	24
Synthetic RTL (64)	6 (18)	3 (6)	12	145	121	20
Synthetic RTL (125)	6 (21)	3 (9)	15	167	121	17
Synthetic RTL (216)	6 (21)	3 (9)	18	187	121	15

Fig. 9: Coverage instrumentation overheads for synthetic RTL designs



(a) Average time (s) to find bug



(b) Reached states

Fig. 10: Efficiency of coverage guided fuzzing for synthetic RTL designs

Runtime Instrumentation Overhead. In the case of runtime overhead due to the instrumentation, simulation speeds were not largely different by each other. More specifically, the original performance before the coverage instrumentation showed 2.36 MHz, and after the coverage instrumentation, DIFUZZRTL's register-coverage showed 2.29 MHz (i.e., 4.2% slowdown) and RFuzz's mux-coverage showed 2.02 MHz (i.e., 15.4% slowdown). As we will show later with real-world RTL designs, if the target RTL designs are complicated enough, DIFUZZRTL's performance improvement over RFuzz becomes significant—up to 40 times for the RISC-V Rocket core.

Efficiency of Register Coverage-Guided Fuzzing. Coverage metrics during the RTL fuzzing should efficiently guide an input stimuli to unexplored states. To this end, we measured the average cycles to reach the vulnerable state in each version of synthetic RTL design with different coverage guidances. In the evaluation, the fuzzer randomly generates input which is defined as a concatenation of bit vectors fed to the design every cycle. We use a simple random bit flip algorithm for mutation so that the only difference is the coverage for input guidance, i.e., 1) no-coverage guided, 2) mux-coverage guided, and 3) register-coverage guided.

As illustrated in Figure 10, register-coverage guided fuzzer shows a remarkable improvement over other two coverage-guiding methods thanks to the judicious guidance. On the other hand, mux-coverage was even worse than the no-coverage due to the limitations of the mux-coverage. This is because the

	no-cov	mux-cov	reg-cov
no-cov	$N.A.$	-	-
mux-cov	$9.46e-13$	$N.A.$	-
reg-cov	$2.64e-33$	$1.25e-34$	$N.A.$

Fig. 11: `p-values` of the Mann-Whitney U test between distributions, where each distribution is populated with each coverage-guiding method. Two distributions are considered significantly different if `p-value` is less than 0.05 [51].

	no-cov	mux-cov	reg-cov
no-cov	$N.A.$	-	-
mux-cov	no-cov	$N.A.$	-
reg-cov	reg-cov	reg-cov	$N.A.$

Fig. 12: The results of Vargha Delaney's A12 measure between the distributions. The coverage-guiding method name in a cell is the result of the VDA measure [52], which is expected to show the higher performance between the methods in the corresponding row and column (i.e., exploring a more number of states in a given number of iterations).

mux-coverage guided fuzzer not only failed to recognize the valid input but also mis-recognized the invalid input as a valid one. As a result, it inserted such invalid inputs to the fuzzing corpus and wasted the cycles to mutate the invalid inputs which do not help to explore the new state space. This tendency becomes clarified as the number of finite states increases, meaning that the register-coverage becomes more efficient. Compared to mux-coverage guided fuzzer, register-coverage guided fuzzer reached the bug state almost 6.4 times faster.

In terms of the average number of reached states during the fuzzing iterations, register-coverage has shown the best performance—i.e., register-coverage guided fuzzer explored the highest number of states as shown in Figure 10-(b). To come up with a more robust statistical conclusion, we performed two statistical testings, the Mann-Whitney U test [53] and the Vargha Delaney's A12 (VDA) measure [52], on the distributions obtained from the number of reached states during 5,000 iterations.

According to the Mann-Whitney U test, `p-value` between each distribution (i.e., a distribution populated with no-coverage, mux-coverage, and register-coverage guiding method) was always less than 0.05 as shown in Figure 11, suggesting that all three distributions show clear statistical differences. The VDA measure also demonstrated that the register-coverage always showed higher improvement over the other two coverage-guiding methods (shown in Figure 12)—the VDA score was always larger than 0.71. One thing to note is that the mux-coverage clearly decreased the performance even more than the no-coverage case as we described before.

Project	**Num. reg (Bits)**		**Num. muxes**	**Verilog**	**Overhead (%)**	
	All	Control	Total	Original	mux-cov	reg-cov
mor1kx	258 (780)	90 (106)	1.33 k	8.31 k	✗	21
Rocket	1,3 k (15.3 k)	207 (661)	5.3 k	69.2 k	112	18
Boom	4.90 k (36.6 k)	330 (990)	21 k	168 k	✗	15

✗ RFuzz failed to instrument Mor1kx cappuccino and Boom core

Fig. 13: Instrumentation overhead for real-world CPU RTLs

Project	**Simulation speed (Hz)**			**Slowdown (%)**		**Fuzzing speed[*] (Hz)**	
	no-cov	mux-cov	reg-cov	mux-cov	reg-cov	mux-cov	reg-cov
mor1kx	2.94 k	✗	2.76 k	✗	6.1	✗	0.41
Rocket	2.44 k	56.5	2.27 k	97	6.9	0.006	0.23
Boom	1.71 k	✗	1.60 k	✗	6.4	✗	0.15

[*]Fuzzing speed is defined as the number of fuzzing iterations per second.

✗ RFuzz failed to instrument Mor1kx cappuccino and Boom core

Fig. 14: Runtime overheads of register-coverage and mux-coverage for real-world CPU RTLs.

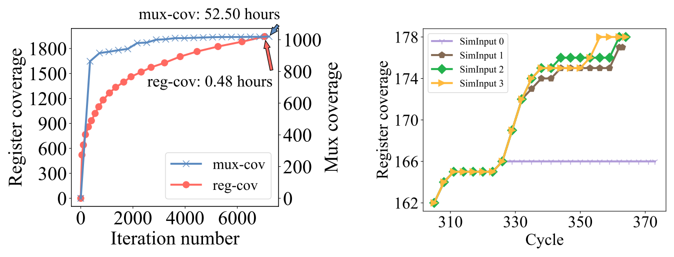
C. Fuzzing Real-World Designs (Software Simulation)

Static Instrumentation Overhead. Secondly, we evaluated the efficiency of register-coverage on real world designs. The statistics of designs and overhead of mux-coverage and register-coverage instrumentations are shown in Figure 13. As in the second column, the total number of registers ranges from two hundreds to five thousands depending on the complexity of a design. However, DIFUZZRTL classified a few hundred of registers into control registers regardless of the design. Approximately, 10% of registers were classified as control registers, and the total bit width has decreased about 97%.

When it comes to the line of Verilog, register-coverage instrumentation showed the minimal increase, which is about 17% as in the last column. Even with Boom core which has almost 200,000 Verilog code lines, the overhead was moderate (14.8%) due to the efficient instrumentation. On the other hand, mux-coverage instrumentation on Rocket core increased the number of line almost twice because of the wiring cost of monitoring per each mux. Moreover, RFuzz [14] was not able to instrument Boom core due to the resource constraint.

Regarding the instrumentation detail, DIFUZZRTL allocates more space for each variables (i..e, `regstate`, `covmap` and `covsum`) as the number of control registers increases in a module. While the maximum size of variables is a configurable parameter, we set the maximum size of `regstate` to be 20-bits for this evaluation, which covers up to 2^{20} bits (i.e. 1 `Mb`). In the case of Boom core, five out of 151 modules were instrumented using the maximum size variables, and LSU was the module which had the most control registers with 156 total bit width. The `regstate` in such a large module can have hash collisions, which is also an important factor for fuzzing, but we leave the detailed result of hash collision in §XI due to the space limit.

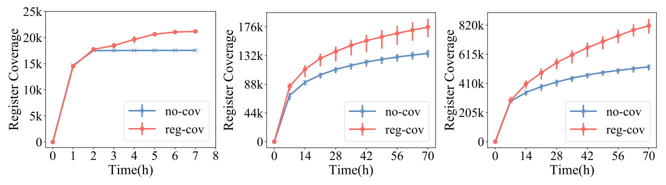
Runtime Instrumentation Overhead. The run time overheads of the coverage instrumentations are shown in Figure 14. As expected, the simulation speed has decreased as the design complexity increases. Mor1kx is slow because it can only be simulated using slow Icarus Verilog simulator [48]. The decreased simulation speed due to the register-coverage instrumentation was about 7% thanks to the simple operation



(a) Mux and register-coverage increments per fuzzing iterations

(b) register-coverage increment in one iteration

Fig. 15: Efficiency of register-coverage for real world RTL designs



(a) Mor1kx cappuccino (b) Rocket core (c) Boom core

Fig. 16: RTL simulation results on register-coverage guided fuzzing and no-coverage guided fuzzing

of `regstate` hashing and `covmap` checking. However, instrumenting mux-coverage extremely decreased the simulation speed because it connects the select signals of all muxes to the top level module and monitors the signal every cycle. In §XI, we summarize further explanation on the fundamental difference of register-coverage and mux-coverage. Overall, register-coverage instrumented design runs almost 40 times faster then mux-coverage instrumented design.

Efficiency of Register Coverage-Guided Fuzzing. To compare the efficiencies of the coverage, we measure the register-coverage and mux-coverage while running same inputs on the instrumented designs. The inputs were collected while running the fuzzer with only mux-coverage for 52 hours. The coverage results are shown in Figure 15-(a) using the iteration number as the x-axis to eliminate the effect of runtime overhead. Mux-coverage quickly saturated as the iteration proceeds, but the fuzzer using register-coverage continuously found new seeds and increased the coverage. This is because register-coverage captures fine-grained behaviors of the design by capturing the state every cycle as illustrated in Figure 15-(b).

Fuzzing Performance. To evaluate the performance of fuzzer, we compared random testing without coverage guiding (`no-cov`) and DIFUZZRTL (`reg-cov`), where the random testing only generates formatted instructions without guidance. The results are illustrated in Figure 16. Looking at the results, DIFUZZRTL reached higher coverage as the design complexity increases. Especially in Boom core, the fuzzer achieves remarkable gain over random testing.

Effectiveness of Interrupt Mutation. One of the reason DIFUZZRTL has chosen to fuzz CPU directly is that large input space leads to an extensive exploration. We wonder how DIFUZZRTL can leverage such opportunity by generating

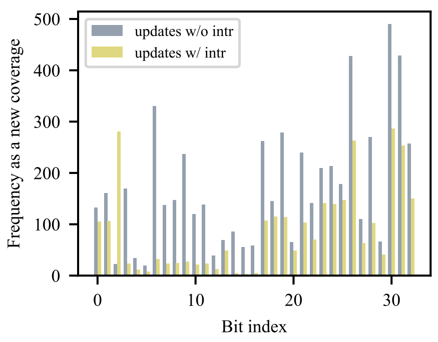


Fig. 17: Frequency of registers bit values in Rocket core when new coverage is explored

Project	FPGA emulation (Hz)		Fuzzing speed* (Hz)	
	`mux-cov`	`reg-cov`	`mux-cov`	`reg-cov`
Rocket	5.73 M (90 M)	5.74 M (90 M)	4.13	4.13
Boom	✗	5.73 M (90 M)	✗	4.13

*Fuzzing speed is defined as the number of fuzzing iterations per second.
✗ RFuzz failed to instrument Boom core due to the out-of-resource issue.

Fig. 18: Runtime overheads of register-coverage and mux-coverage for FPGA emulation. The numbers in the brackets are timing constraints for the FPGA bitstream.

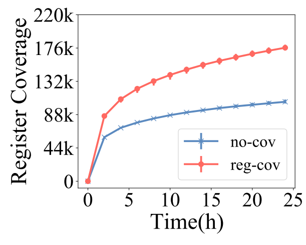
input stimuli which includes both memory values and interrupt signals. Thus we designed an experiment to answer the question: does raising interrupt signals lead to the exploration of an unknown state.

To discriminate the impact of interrupt assertion, we repeated running Rocket core with two SimInput which are only different in that the latter one contains non-zero interrupt values. Thus the latter SimInput explores the new states introduced by the interrupt assertions. For each run, we collected the values of control registers as a bit vector whenever a new state is explored, which represents a control state of the module.

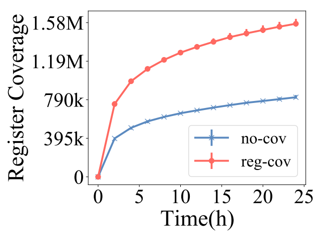
Then, we summarized the bit vectors as a histogram which is shown in Figure 17, thereby each bar represents the frequency of the bit when a new state is explored. Among the bars, we found that only the bit in index three shows increased frequency when the interrupt is used. After manually auditing the source code, we confirm that the bit in index three was belonging to the `wb_reg_xcpt` register in Rocket core, which is used for exception handling. Therefore, we conclude that raising interrupts enables DIFUZZRTL to explore unknown states, which is otherwise not possible to be reached.

D. *Fuzzing Real-World Designs (FPGA Emulation)*

Since the performance of fuzzing is highly depending on the speed of running each iteration, FPGA emulation can significantly improve the performance of fuzzing—FPGA runs on a synthesized hardware circuit. In this sense, we attempted to evaluate DIFUZZRTL and RFuzz with the FPGA emulation, particularly focusing on following two aspects: the runtime instrumentation overheads and the fuzzing performance.



(a) Rocket



(b) Boom

Fig. 19: FPGA emulation results of register-coverage guided fuzzing and no-coverage guided fuzzing

To shortly summarize the results, DIFUZZRTL successfully instrumented both Rocket core and Boom core, and showed improved runtime instrumentation overheads and fuzzing performance as expected. In the case of RFuzz, it was able to instrument Rocket core and showed the improved performance as expected. However, it was not able to instrument Boom core, because mux-coverage of RFuzz requires more resources than available, leading to the instrumentation failure.

Runtime Instrumentation Overhead. As shown in Figure 18, DIFUZZRTL instrumented and compiled both Rocket core and Boom core at 90 MHz clock frequency using `Vivado 2018.3`. In each iteration, the synthesized core runs at 5.73 MHz. This clock slowdown is as expected because DIFUZZRTL should perform following fuzzing operations: 1) monitoring and scheduling the running instance; and 2) fuzzing management. Also, operations outside FPGA (i.e., fuzzing corpus management and SimInput mutation) becomes a bottleneck since DIFUZZRTL alternatively runs the mutation and FPGA emulation. This clock slowdown can be mitigated by employing general optimization techniques, such as running the mutation and FPGA emulation in parallel, but we leave those as future work.

RFuzz was also able to run mux-coverage instrumented Rocket core at 5.73 MHz with 90 MHz clock frequency. However, RFuzz failed to build Boom core, so we were not able to perform the evaluation for RFuzz's case on Boom core.

Fuzzing Performance. The register-coverage guided fuzzer using FPGA emulation showed much better performance compared to the random testing without coverage guiding as shown in Figure 19. Specifically, register-coverage guided fuzzing reached the two times more number of register-coverage than without coverage guiding after fuzzing 9 hours. It is worth noting that, as we have shown before, the difference was only about 20 % given the same 9 hours fuzzing time in software simulation. This clear improvement compared to the software simulation (i.e., from 20 % to 2 `X` for the given 9 hours) is thanks to the fact that the FPGA emulation runs almost 30 times faster than the simulation.

E. Newly Discovered Bugs in Real-World RTL Designs

During the evaluation, DIFUZZRTL found 16 new bugs in total (listed in Figure 20), all of which were confirmed by the respective development communities or vendors and some of those are already being patched. More importantly, five of those were assigned with CVE numbers, signifying its strong impacts to the security community. These discovered bugs can be classified into four categories: atomic operation related bugs, instruction decoding bugs, memory related bugs, and performance bugs, demonstrating the wide coverage of DIFUZZRTL with respect to bug types in CPU RTLs.

First, incorrect behaviors in atomic memory operations (load-reserve store-conditional instructions) were frequently found. According to the OpenRISC ISA [43], reservation set by load-reserve instruction should be unset when the snooping hit occurs. However, Mor1kx cappuccino did not follow the ISA, causing memory consistency problems when the bug occurs (CVE-2020-13455).

Similar to OpenRISC, RISC-V does not allow a load-reserve instruction on a misaligned address to set the reservation. However, Boom core did not follow the specification, causing the following store-conditional instruction to succeed. Especially the bug was only triggered when the related address is cached in the core, since the reservation signal reaches the cache line before the exception signal abort the reservation (CVE-2020-29561).

Bugs related to instruction decoding were found as DIFUZZRTL randomly provides illegal opcode to RTLs. Especially in Boom core, floating point instructions with unallowed rounding bits successfully updated the floating point registers. This bug may incur an incorrect results of floating point instructions, which is critical in a scientific computing, as we have observed from the notorious Pentium FDIV bug [2] (Issue #458). Mor1kx cappuccino also has a decoding bug related to bit processing instructions (Issue #114).

DIFUZZRTL was also able to discover bugs related to memory bus. Boom core was incorrectly setting the source field in ProbeAckData which is used for cache coherence memory protocol (CVE-2020-13251). However, the bug was not found even with several verification including running large programs on the chip. We assume the SoC used for Boom core test is tolerant to the bug, but the results will be critical if the Boom core is used with other intolerant (correctly designed) SoC.

Other bugs related to the performance were also found, such as setting `FS` bits in `mstatus` register (Boom core), which is used to decide whether save or not the floating point registers during the context switching.

F. Case Study with Real-World Bugs

In order to clearly showcase the effectiveness of DIFUZZRTL's coverage guiding, we performed a case study with two bugs which we discovered with DIFUZZRTL (i.e., Issue #492 and CVE-2020-29561 in Figure 20, both of which were found on Boom core). In particular, we compare the results of DIFUZZRTL with following two approaches: 1) `riscv-torture` [37], which randomly generates instructions using a pool of handcrafted instruction sequences without any coverage guided feature. This `riscv-torture` is widely used to verify various RISC-V cores by the development community; and 2) mux-coverage guided fuzzer, which generates

Project	**ISA**	**Bug ID**	**Description**	**Confirmed**	**Fixed**
Mork1x	OpenRISC	CVE-2020-13455	Reservation is not cancelled when there is snooping hit between lwa and swa	✓	pending
		CVE-2020-13454	Jump to link register does not assert illegal instruction exception	✓	pending
		CVE-2020-13453	Misaligned swa raise exception when reservation is not set	✓	pending
		Issue #114	`l.fl1`, `l.ff1` instruction decoding bug	✓	✓
		Issue #99	`eear` register not saving instruction virtual address when illegal instruction exception	✓	✓
Rocket chip	RISCV	Issue #2345	Instruction retired count not increased when ebreak	✓	pending
Boom	RISCV	CVE-2020-13251	Source field in `ProbeAckData` does not match the sink field of `ProbeRequest`	✓	✓
		Issue #458	Floating point instruction which has invalid `rm` field does not raise exception	✓	✓
		Issue #454	FS bits in `mstatus` register is set after `fle.d` instruction	✓	pending
		Issue #492	When `frm` is set `DYN`, floating point instruction with `DYN` `rm` field should raise exception	✓	✓
		Issue #493	Rounding mode in `fsqrt` instruction does not work	✓	✓
		Issue #503	`invalid operation` flag is not set after invalid `fdiv` instruction	✓	✓
		CVE-2020-29561	Misaligned `lr` instruction on a cached line set the reservation	✓	✓
Spike	RISCV	CVE-2020-13456	Misaligned `lr.d` should not set load reservation	✓	✓
		Issue #2390	Reading `dpc` register should raise exception in machine mode	✓	✓
		Issue #426	Faulting virtual address should not be written to `mtval` when `ebreak`	✓	✓

Fig. 20: A list of newly discovered bugs by DIFUZZRTL. DIFUZZRTL identified 16 bugs in total, all of those were confirmed by respective vendors. Five of those were assigned with CVE numbers.

Bug ID	Elapsed time (h)		
	riscv-torture	`mux-cov`	`reg-cov`
Issue #492	118	✗	20.3
CVE-2020-29561	✗	✗	31.7

✗Not able to reproduce bug

Fig. 21: Average time to find real-world bugs using each approach.

instructions on DIFUZZRTL fuzzing framework with RFuzz's mux-coverage guiding. Since RFuzz compiler pass was not able to instrument all the muxes in Boom core due to the resource constraints, we modified the pass to randomly and selectively instrument muxes.

Case Study on Issue #492. The Issue #492 bug (i.e., dynamic `rm` bug) can be found with the following three steps: 1) Starting from the initial state, the `FS` bit in `mstatus` register should be set; 2) `frm` bits in `fcsr` register should be set with a specific value; and 3) A floating point instruction with a `DYN`-enabled `rm` field should be executed [54].

For each fuzzing iteration, we configured each approach to generate (or mutate) a sequence of instructions and tested designs. The first row of Figure 21 shows an average elapsed time to find the bug using each approach. DIFUZZRTL with register-coverage guidance was almost 6 times faster than riscv-torture in finding the bug since it captures each step above as a new coverage and guides input efficiently. However, the fuzzer which used mux-coverage was not able to find the bug due to the slow fuzzing speed and inefficient guidance.

Case Study on CVE-2020-29561. The CVE-2020-29561 bug (i.e., misaligned `lr` bug) can be found with the following three steps: 1) A special memory instruction which fetches the accessed cache line (e.g., `amoand`) should be executed; 2) Load-reserve instruction (i.e., `lr`) should be executed on the same cache line but with a misaligned address. Such an instruction raises a misaligned load exception, and the program counter jumps to the address of exception vector [54]; and 3) After exception, additional store-conditional instruction (i.e., `sc`) which accesses the same cache line should be performed, but this time, with a correct address alignment.

Due to the complex nature of the bug, it took approximately 30 hours for DIFUZZRTL to find the bug as shown in Figure 21. Meanwhile, none of other techniques were able to find the bug even after running ten times of elapsed time for DIFUZZRTL (i.e., 300 hours). When reasoning about the bug reproducing, the first step (i.e., fetching a specific cache line) does not have any architectural effect on the second step (i.e., loading and reserving the same cache line). However, DIFUZZRTL captures the new coherency state of the cache line explored by the two steps, and successfully guides inputs to the next step.

VII. LIMITATIONS

Confirming the Semantic Bugs. Since DIFUZZRTL detects semantic bugs by comparing the execution results of ISA and RTL simulation, the system alone cannot confirm which one is responsible for the found bug. Thus DIFUZZRTL requires manual inspection of the specification to confirm the bug. DIFUZZRTL may leverage the results of different implementations of the same specification (e.g. Rocket and Boom core) to at least confine the suspicious design as proposed in [36]. While these are an interesting research direction, we leave them as future work.

Applying Register Coverage on General RTL Designs. The design of DIFUZZRTL has several design points that can only be used for the RTL designs that have golden models. However, DIFUZZRTL's idea on register coverage is generic enough such that it can be applied to various RTL designs in the future, such as cache, ALU, rob, etc. It is also possible to run a targeted fuzzing on a module such as D-cache while simulating entire CPU because DIFUZZRTL leverages the module by module approach.

Identifying Side-Channel CPU Bugs. Since DIFUZZRTL relies on differential testing to discover bugs, DIFUZZRTL alone cannot identify side-channel CPU issues such as Spectre, Meltdown, Foreshadow, MDS. In order to discover these bugs, DIFUZZRTL's compilation framework can be extended to monitor the microarchitectural states of modules to check the state changes depending on a secret value.

VIII. Related work

Dynamic RTL Verification. Dynamic RTL verification is an old verification methodology which is still widely used. In [55], authors introduce pseudo random test generation for verifying DECchip and challenges for defining coverage and test generation. Shai et al. [56] designs a neural network to generate input stimuli using coverage but they require deep knowledge on the design. MicroGP [57] is similar to DIFUZZRTL in that it verifies a processor using coverage for input generation. However, the work requires external tools for coverage evaluation thus increasing manual work.

Recently, various open-source tools for CPU verification [37, 58] have been introduced with the rise of RISC-V, which randomly generate instructions to test the processors. While random instruction generation is efficient for testing a wide range of processor functionalities, such tools do not have a coverage-guiding feature to rigorously test the processors. DIFUZZRTL, on the other hand, employs the coverage-guiding with carefully designed coverage metrics to further verify the processors in deeper aspects.

Other CPU fuzzing works [59, 60] aim at finding hardware flaws as well as undocumented instructions through exhaustively searching the instruction space of the x86 architecture. Instead of testing a sequence of instructions, these works focus on generating a single potentially harmful instruction opcode because the x86 architecture has a huge instruction space with a variable instruction length. While DIFUZZRTL is designed for testing a sequence of instructions, DIFUZZRTL can also help these approaches if the RTL source code is available—i.e., DIFUZZRTL can provide the coverage metric for the instruction decode unit.

Coverage Definition. On the other hand, researchers have continued to find good coverage definition. Coverage metrics are classified into code coverage and functional coverage, where code coverage relies on the analysis of the code while functional coverage relies on design specific information [61]. Code coverages including statement, or toggle coverage are easy to obtain but insufficient to guide input. Functional coverage requires designers manual setup.

Moudanos et al. [10] introduce state coverage and transition coverage which are directly defined on the FSM. However, the metrics is not scalable because of the state explosion problem. HYBRO [62] tries to reach hard to reach states while maximizing branch coverage. As mentioned, branch coverage on RTL code has fundamental limitation. Recently, RFUZZ [14], proposes mux coverage which can be synthesized into FPGA accelerated simulation. Nevertheless, the most widely used coverage is the functional coverage which is manually defined by the designers [14].

Static RTL Verification. Along with dynamic verification, methods to verify RTL using static analysis have been developed. Symbolic execution mathematically verifies all the design space of the RTL code. STAR [13] employs hybrid approach, using random input generation and symbolic execution but the method has limitations on the sequential depth of a variable. Zhang et al. [63] introduces backward symbolic execution to reach assertion violation.

Coverage-guided Fuzzing. Many recent studies leverage coverage-guided fuzzing approach and try to achieve higher coverage and vulnerabilities in user programs and kernels [18–32]. For example, taint-analysis [20, 24] and symbolic execution [21, 23] techniques have been proposed to overcome the limitations of coverage-guided fuzzing such as magic bytes and nested branches.

Differential Fuzzing. Differential fuzzing is designed to discover semantic bugs by observing inconsistent behaviors across similar applications. For example, [64, 65] leverages differential fuzzing to find the inconsistent behaviors across Java Virtual Machines (JVMs). Nezha [35] defined the notion of δ-diversity, which represents the asymmetric behaviors between testing programs, to guide the fuzzer to disclose semantic bugs in softwares such as SSL/TLS libraries and PDF viewers.

IX. Conclusion

This paper proposed DIFUZZRTL, an RTL fuzzer to discover CPU bugs in RTL designs. The key design features of DIFUZZRTL includes register-coverage guided fuzzing techniques, which can be used as a drop-in-replacement fuzzer to test various CPU RTLs. DIFUZZRTL is implemented to perform the fuzz testing for three open-source CPUs, OpenRISC Mor1kx Cappuccino, RISC-V Rocket Core, and RISC-V Boom Core. During the evaluation, DIFUZZRTL identified 16 new bugs from these CPUs, demonstrating its strong practical prospects.

X. Acknowledgment

We would like to thank Matthew Hicks and anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. We also thank the developers of the open-source CPUs for their helpful feedback and responses. This work was partially supported by National Research Foundation (NRF) of Korea grant funded by the Korean government (NRF-2021R1A2C3014131, NRF-2019R1C1C1006095), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work. The Institute of Engineering Research at Seoul National University provided research facilities for this work.

References

[1] Kypros Constantinides, Onur Mutlu, and Todd Austin. Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Washington, DC, November 2008.

[2] Pentium fdiv: The processor bug that shook the world. https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773.

[3] Intel's fdiv bug: Empty cells at empty tables. https://www.olenick.com/blog/articles/infamous-software-bugs-fdiv-bug/.

[4] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[5] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[6] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *Proceedings of the 28th USENIX Security Symposium (Security)*, SANTA CLARA, CA, August 2019.

[7] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[8] Intel finds tsx enterprise bug on haswell, broadwll cpus. https://www.pcworld.com/article/2464880/intel-finds-specialized-tsx-enterprise-bug-on-haswell-broadwell-cpus.html.

[9] Intel november 2019 microcode update. https://access.redhat.com/solutions/2019-microcode-nov.

[10] Dinos Moundanos, Jacob A Abraham, and Yatin Vasant Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, 1998.

[11] Yanhong Zhou, Tiancheng Wang, Huawei Li, Tao Lv, and Xiaowei Li. Functional test generation for hard-to-reach states using path constraint solving. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):999–1011, 2015.

[12] Jian Wang, Huawei Li, Tao Lv, Tiancheng Wang, Xiaowei Li, and Sandip Kundu. Abstraction-guided simulation using markov analysis for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(2):285–297, 2015.

[13] Lingyi Liu and Shabha Vasudevan. Star: Generating input vectors for design validation by static analysis of rtl. In *2009 IEEE International High Level Design Validation and Test Workshop*, San Francisco, CA, November 2009.

[14] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: coverage-directed fuzz testing of rtl on fpgas. In *Proceedings of the 37th IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Diego, CA, November 2018.

[15] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[16] Peter J Ashenden. *The designer's guide to VHDL*. Morgan Kaufmann, 2010.

[17] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[19] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[20] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[21] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[22] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, September 2018.

[23] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[24] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[25] syzkaller. https://github.com/google/syzkaller.

[26] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[27] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[28] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*, SANTA CLARA, CA, August 2019.

[30] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[31] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October 2017.

[32] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[33] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[34] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[35] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[36] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[37] Risc-v torture test. https://github.com/ucb-bar/riscv-torture.

[38] Memory controller ip core. https://opencores.org/projects/mem_ctrl.

[39] Open source hardware association. https://www.oshwa.org/.

[40] Boom: Berkeley out-of-order machine. https://github.com/riscv-boom/riscv-boom.

[41] Rocket chip generator. https://github.com/chipsalliance/rocket-chip.

[42] Risc-v isa manual (privileged). https://riscv.org/specifications/privileged-isa/.

[43] Openrisc isa manual. https://openrisc.io/or1k.html.

[44] Pyverilog: Python-based hardware design processing toolkit for verilog hdl. https://github.com/PyHDI/Pyverilog.

[45] Chisel 3: A modern hardware design language. https://github.com/freechipsproject/chisel3.

[46] Cocotb, a coroutine based cosimulation library for writing vhdl and verilog testbenches in python. https://github.com/cocotb/cocotb.

[47] Verilator open-source systemverilog simulator and lint system. https://github.com/verilator/verilator.

[48] Icarus verilog. http://iverilog.icarus.com/.

[49] https://bar.eecs.berkeley.edu/projects/firesim.html.

[50] Aws ec2 fpga development kit. https://github.com/aws/aws-fpga.

[51] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[52] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
[53] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.
[54] Risc-v isa manual (unprivileged). https://riscv.org/specifications/unpriviledged-isa/.
[55] Michael Kantrowitz and Lisa M Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor. In *Proceedings of the 33rd annual Design Automation Conference (DAC)*, Las Vegas, NV, June 1996.
[56] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of the 40th annual Design Automation Conference (DAC)*, Anaheim, CA, June 2003.
[57] Giovanni Squillero. Microgp—an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6:247–263, 09 2005.
[58] Sv/uvm based open-source instruction generator for risc-v processor verification. https://github.com/google/riscv-dv.
[59] Sandsifter: the x86 processor fuzzer. https://github.com/xoreaxeaxeax/sandsifter.
[60] Xixing Li, Zehui Wu, Qiang Wei, and Haolan Wu. Uisfuzz: An efficient fuzzing method for cpu undocumented instruction searching. *IEEE Access*, 7:149224–149236, 2019.
[61] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 2001.
[62] Vineeth V Acharya, Sharad Bagri, and Michael S Hsiao. Branch guided functional test generation at the rtl. In *2015 20th IEEE European Test Symposium (ETS)*, Cluj-Napoca, Romania, May 2015.
[63] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka, Japan, October 2018.
[64] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of jvm implementations. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.
[65] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, June 2016.
[66] Prem C Consul and Gaurav C Jain. A generalization of the poisson distribution. *Technometrics*, 15(4):791–799, 1973.

XI. Appendix

This appendix section provides supplementary information regarding this paper.

Listing of Motivation Example. Figure 24 shows the appended Verilog code to the original code (shown in Figure 22) for the instrumentation. Only `state`^{S} and `state`^{F} registers are used because they are wired into the mux control signal. The registers are hashed into `regstate` through random offset ({ }) and xor operation (ˆ). Then, the logic for `covmap` and `covsum` is instrumented.

Calculation on Collision Probability. Since DifuzzRTL hashes control register value into the `regstate`, collision could happen where different control register values are hashed into the same `regstate` value. To this end, DifuzzRTL allocates more space for each variables (i.e. `regstate`, `covmap` and `covsum`) as the number of control registers increases. However, the collision is unavoidable if the total bit width of control registers is larger than the maximum size of `regstate`.

In order to understand the collision probability, we measured the register-coverage instrumentation details. While the maximum size of variables is a configurable parameter, we set the maximum size of `regstate` to be 20-bits for this evaluation, which covers up to 2^{20} bits (i.e. 1 `Mb`). In the case of Boom core, five out of 151 modules were instrumented using the maximum size variables, and LSU was the module which had the most control registers. The total bit width of control registers in LSU was 156 bits. When all the control registers are hashed into 20 bits `regstate`, the collision probability becomes $1-(1-\mathtt{p})^{{}_kC_2}$ where `p` is a probability of two different states being hashed to the same `regstate` value and `k` is the number of simulated cycles. Thus, the total collision probability converges to 1 as the simulated cycle increases, reaching 99.9 % at 5,384 simulated cycles. Since the number of collisions follows a

```
module mem_ctrl(
  input clock,

  input         sdram_valid,
  input [3:0]   sdram_data_i,
  input         flash_valid,
  input [7:0]   flash_data_i,

  output        sdram_ready,
  output        flash_ready,

  output        out_valid,
  output [7:0]  out_data
);

  reg[1:0] state_sdram; // READY, RECEIVING, BUSY
  reg      state_flash; // READY, BUSY

  reg[7:0] data_sdram;
  reg[7:0] data_flash;

/********** combinational **********/

  assign sdram_ready = (state_sdram != `BUSY_S);
  assign flash_ready = (state_flash != `BUSY_F);

  assign out_valid = (state_sdram == `BUSY_S) ||
                     (state_flash == `BUSY_F);
  assign out_data = (state_sdram == `BUSY_S) ?
                     data_sdram : data_flash;

/********** sequential **********/

  always @(posedge clock) begin
    if (state_sdram == `READY_S) begin
      if (sdram_valid) begin
        state_sdram <= `PENDING_S;
        data_sdram <= {4'b0000, sdram_data_i};
      end
    end else if (state_sdram == `PENDING_S) begin
      if (sdram_valid) begin
        state_sdram <= `BUSY_S;
        data_sdram <= {sdram_data_i, 4'b0000} | data_sdram;
      end
    end else if (state_sdram == `BUSY_S) begin
      state_sdram <= `PENDING_S;
    end

    if (state_flash == `READY_F) begin
      if (flash_valid) begin
        state_flash <= `BUSY_F;
        data_flash <= flash_data_i;
      end
    end else if (state_flash == `BUSY_F) begin
      state_flash <= `READY_F;
    end
  end
endmodule
```

Fig. 22: Verilog code example of a simple memory controller which connects sdram and flash memory to the outer world. Bug is triggered when both `state`^{S} and `state`^{F} at BUSY.

```
/********** patch **********/

end else if (state_flash == `FLASH_BUSY) begin
  if (state_sdram != `SDRAM_BUSY) begin
    state_flash <= `FLASH_BUSY;
  end
end
```

Fig. 23: Bug fix for Figure 22. $\texttt{state}^{F}$ waits until $\texttt{state}^{S}$ is not busy

```
/********** instrumented **********/

reg[2:0] reg_state;
reg[2:0] covsum;
reg      covmap[8:0];

always @(posedge clock) begin
  reg_state <= { state_sdram, 1'h0 } ^ { 2'h0, state_flash };
  covmap[reg_state] <= 1;

  if (!covmap[reg_state]) begin
    covsum <= covsum + 1;
  end
end
```

Fig. 24: Instrumented result of Figure 22; Only $\texttt{state}^{S}$ and $\texttt{state}^{F}$ registers are instrumented.

Poisson distribution, we conclude that average 6 collisions occur in one iteration [66].

While this collision probability suggests that the collision is unavoidable given the large state space of RTL modules, we believe a loss would be minimal. This is because an input which discovers a new state further reaches multiple new states in one iteration, and DIFUZZRTL will save the input as a new seed if at least one new non-colliding state is discovered. Also, a carefully designed hashing algorithm for coverage-guided fuzzing would be able to reduce the collision probability as proposed by `collAFL` [26].

Fundamental Differences between Register-coverage and Mux-coverage. To shortly summarize, register-coverage of DIFUZZRTL has three fundamental differences from mux-coverage of RFuzz [14], making DIFUZZRTL's simulation more efficient. First, instrumenting control registers is more efficient than instrumenting all the muxes. Second, DIFUZZRTL summarizes the coverage in each module and wires only the sum of coverage until the top level module, but RFuzz needs to propagate all the mux control signals upto the top level module. Third, DIFUZZRTL employs simple hashing for the coverage computation while RFuzz requires a saturating counter for each mux.

As a result, assuming the number of modules in a design is M and the number of elements (i.e., control registers for DIFUZZRTL, muxes for RFuzz) in a module is N, DIFUZZRTL requires $O(M \cdot N + M)$ of computation resources for hashing control registers and connecting the sum of coverage to the top level module. On the other hand, RFuzz requires $O(M^2 \cdot N + M \cdot N)$ resources for connecting all the mux control signals to the top level module and attaching a saturating monitor for each mux.

Potential Security Impacts of Hardware Flaws. Abnormal behaviors of the processors failing to follow the ISA could result in critical security damages. DIFUZZRTL found several hardware flaws that can be potentially destructive, thus assigned with CVE reference numbers.

Among those, CVE-2020-13455, CVE-2020-13453, CVE-2020-13251, and CVE-2020-29561 directly harm the memory consistency of the processors, thus can result in potential race bugs. It is widely known that the race bugs are harmful to the security of the entire system and the attackers can abuse such bugs to compromise the system. More critically, it would take long time to identify that the root causes of such bugs are hardware flaws since the race bugs are already too subtle and non-deterministic to be detected.

CVE-2020-13454 and CVE-2020-13453 can be abused by the attackers to bypass ROP defense schemes. Conventional defense tools against ROP attacks would not assume those instructions as a potential gadgets because the instructions should result in an exception in correct semantics. However, attackers with the knowledge of these hardware flaws could rely on the fact that the victim instructions do not trap, avoiding the expected exceptions and thus completing the ROP attacks.