Cascade: CPU Fuzzing via Intricate Program Generation

Flavien Solt
ETH Zurich

Katharina Ceesay-Seitz
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

Generating interesting test cases for CPU fuzzing is akin to generating programs that exercise unusual states inside the CPU. The performance of CPU fuzzing is heavily influenced by the quality of these programs and by the overhead of bug detection. Our analysis of existing state-of-the-art CPU fuzzers shows that they generate programs that are either overly simple or execute a small fraction of their instructions due to invalid control flows. Combined with expensive instruction-granular bug detection mechanisms, this leads to inefficient fuzzing campaigns. We present Cascade, a new approach for generating *valid* RISC-V programs of arbitrary length with highly *randomized* and *interdependent* control and data flows. Cascade relies on a new technique called asymmetric ISA pre-simulation for entangling control flows with data flows when generating programs. This entanglement results in non-termination when a program triggers a bug in the target CPU, enabling Cascade to detect a CPU bug at program granularity without introducing any runtime overhead. Our evaluation shows that long Cascade programs are more effective in exercising the CPU's internal design. Cascade achieves 28.2x to 97x more coverage than the state-of-the-art CPU fuzzers and uncovers 37 new bugs (28 new CVEs) in 5 RISC-V CPUs with varying degrees of complexity. The programs that trigger these bugs are long and intricate, impeding triaging. To address this challenge, Cascade features an automated pruning method that reduces a program to a minimal number of instructions that trigger the bug.

1 Introduction

With the increasing popularity of open-source RISC-V CPUs and the high cost of formal verification, CPU fuzzing is gaining momentum [8, 9, 14, 30, 32, 34, 35]. The effectiveness of CPU fuzzing strongly depends on the quality of the test cases and efficient bug detection. State-of-the-art CPU fuzzers fail at both: they execute only a small fraction of the intended instructions per test case, and rely on incomplete or expensive instruction-granular bug detection mechanisms at runtime.

This paper presents Cascade, a new CPU fuzzer that explicitly generates *valid* RISC-V programs of arbitrary length with highly randomized and interdependent data and control flows. Executing many instructions per program enables Cascade to efficiently trigger bugs. When a bug is triggered, the interdependence of the data and control flows results in program non-termination, enabling Cascade to detect bugs at program granularity without any runtime overhead. Our evaluation shows that Cascade achieves significantly more coverage than the state-of-the-art fuzzers and discovers a large number of new bugs in RISC-V CPUs of various complexities.

Programs for CPU fuzzing. To investigate properties of the programs generated by state-of-the-art CPU fuzzers [32, 35, 36], we defined two metrics measuring 1) *completion*: the proportion of the instructions in a program that actually execute, 2) *prevalence*: the overhead due to non-randomized initial and final sequences. We find that for each generated program, only a small fraction of the instructions gets a chance to actually execute on the target CPU and that most executed instructions perform non-randomized initialization. These findings hint at low instruction throughput in existing state-of-the-art CPU fuzzers. Furthermore, detecting a bug triggered by these programs introduces its own set of challenges.

Bug detection. Generally, the most obvious way to check whether a program triggered a bug is by checking the CPU's architectural state (i.e., registers and memory) instruction by instruction against a golden model [8, 9, 30, 34, 35]. Apart from performance implications, this approach has practical limitations; as an example, the registers are not always easy to identify in a given CPU's RTL design. In particular, out-of-order CPUs may keep multiple versions of the registers at the same time and identifying the correct ones for monitoring introduces a non-trivial effort when porting a fuzzer to a new CPU [4]. It is also possible to force termination by handling uncontrolled exceptions in the programs [32]. While this technique reduces the problem to only checking the architectural state at the end, it can miss bugs that happen in the middle of the program. An ideal solution generates programs that execute to completion without the need for checking the state during program execution

Cascade. We rely on the idea that finding a bug-triggering program in a CPU is equivalent to finding a program for which the CPU behaves incorrectly. We propose a new fuzzer called Cascade that constructs complex and random RISC-V programs that exert the CPU's architectural and microarchitectural features without requiring time-consuming and error-prone expert effort specific to each CPU. The programs generated by Cascade mix a highly randomized data flow with the control flow, while steering the control flow at all times. To efficiently predict some necessary register values, we introduce the novel notion of asymmetric ISA pre-simulation, where instead of using the Instruction Set Architecture (ISA) simulator to compare the CPU under test with a golden reference model, we use it to construct programs with valid and predictable architectural control flows. Constructing valid programs with highly interdependent control and data flows provides us with three interesting properties. First, the programs can be long, which significantly boost fuzzing performance. Second, with highly randomized data and control flows, we explore unusual operand values and control flows. Lastly, the highly entangled data and control flows enables the non-pervasive detection of bugs amidst long programs by transforming bug expressions into program non-terminations, enabling Cascade to detect bugs without any runtime overhead.

The programs generated by Cascade exercise a rich set of functionalities provided by the RISC-V ISA; they support exceptions without (necessarily) causing termination, data flow-dependent privilege transitions, complex FPU (Floating-Point Unit) operations, and operations under randomized Control and Status Registers (CSRs) exploring different operational states of the CPU. We evaluate Cascade on 6 real-world RISC-V CPUs of different complexities and ISA extensions: VexRiscv, PicoRV32, Kronos, CVA6, Rocket and BOOM. Compared to the state-of-the-art fuzzers such as TheHuzz and DifuzzRTL, Cascade achieves the same coverage 28.2 and 97 times faster, respectively. Cascade discovers 37 new bugs (29 new CVEs) in 5 of these 6 designs which is more than all the state-of-the-art CPU fuzzers combined [10, 14, 32, 35, 36]. We additionally found a critical bug in the popular Yosys synthesizer that results in a wrong netlist.

Automated program reduction. Cascade-generated programs that trigger these new bugs can be long and highly complex, making the analysis of the non-termination intractable by humans. To tackle this challenge, Cascade relies on a new automated program reduction technique that creates minor in-place modifications to a bug-triggering program. These modifications iteratively reduce the program to a minimal bug-triggering form while preserving the sufficient bug-triggering CPU state. Using this technique, we find that these bugs result in hangs, wrong values and exceptions, decode issues and quantifiable performance counter inaccuracies. Furthermore, we find that Cascade's program reduction is significantly helpful when reporting the bugs to the respective CPU maintainers.

Contributions. Our contributions are as follows:

- We design and implement Cascade for generating valid RISC-V programs with highly complex and entangled data and control flows for finding CPU bugs. Cascade correct-by-construction programs achieve high fuzzing performance without introducing any overhead for bug detection.
- We design and implement a new analysis method to efficiently reduce Cascade-generated programs to a minimal form, while preserving the bug-triggering behavior.
- We evaluate the performance of Cascade in terms of speed and coverage and compare it with state-of-the-art CPU fuzzers [32, 35, 36]. We report a total of 37 new bugs found in 5 of the 6 considered RISC-V CPU designs. We further report a bug in the popular open-source Yosys synthesizer.

Open sourcing. For the benefit of the research and CPU design and testing communities, we publish the source code and experiments of Cascade at this URL: https://comsec.ethz.ch/cascade.

2 Background

In this section, we provide background on formal verification, fuzzing software and hardware, and finally on RISC-V.

2.1 Formal verification of hardware

Assertion-based formal verification aims to prove that a hardware design satisfies certain properties, for all possible input values that it may receive and for infinite depth in time. Usually, formal verification engineers manually write properties that target specific verification goals, often taking design-specific knowledge into account. Tools for automated property generation either generate a certain kind of properties, like information flow properties [19], require a (semi-)formal model of the specification [21, 38], or use novel languages [18]. Furthermore, exhaustively verifying complex properties on real world designs does not always scale and requires semi-manual abstraction techniques such as black-boxing or initial value abstraction [20, 25]. Given the high computational and manual cost of formal verification, alternative fuzzing techniques are starting to gain popularity.

2.2 Software fuzzing

Fuzzing consists in applying random inputs to a unit under test and observing whether the unit behaves as expected and whether the output is correct. The goal is to find unknown bugs by covering as much state space as possible by iteratively mutating the input data. While fuzzing usually does not provide formal guarantees of correctness, it has been shown to be a very effective technique to find bugs [23].

Every fuzzer needs a strategy to *generate test cases* and to *detect bugs* when they happen. Software fuzzers may rely on some form of coverage feedback [15, 26, 29, 58], static or dynamic taint analysis [16, 27, 40, 42] or grammars [6, 24, 28, 44, 51, 57] to incrementally find test cases that better exercise the software's functionality. Software fuzzers mainly rely on two techniques to detect when bugs are triggered: crashes [29] and sanitizers [45]. Sanitziers provide, for example, address related checks like buffer overflows [45], checks for undefined behavior such as division by zero [22] or floating-point numerical issues [17].

2.3 Hardware fuzzing

Due to the high cost of formal verification, CPUs are an interesting target for fuzzing. Hardware fuzzers for CPUs differ from software fuzzers on both aspects of test-case generation and bug detection. While software fuzzers often generate random input data streams with little or no input format considerations, hardware fuzzers need to prioritize generation of inputs that follow certain protocols, like bus or ISA specifications, in order to be effective [55]. Inspired by software fuzzing, state-of-the-art hardware fuzzers generate new test cases by generating random instruction sequences and mutating the test case [8–10, 14, 30, 32, 34–36]. Every new CPU fuzzer finds new bugs, but often fewer [8, 9, 14, 30, 32, 34, 35].

Since processors hang only in rare cases, crash detection is not sufficient for bug detection, and sanitizers are currently limited to handwritten SystemVerilog assertions [36]. Therefore, hardware fuzzing needs new methods for bug detection. Most hardware fuzzers apply differential fuzzing by comparing register values of the CPU under test with the results from a purely software-based Instruction Set Simulator (ISS) that serves as a golden model [32]. Comparing the result of every instruction is expensive in terms of runtime, and only feasible for simple CPUs where a direct mapping from RTL design signals to registers is possible. Some CPU fuzzers [32] dump the register values via storage instructions at the end of a test case and compare the results with the ones from the ISS. Such a comparison is only possible if a test case completes and intermediate deviations between the ISS and the CPU under test are propagated through time until the end of the test case.

2.4 RISC-V

RISC-V [59] is a free and open ISA, consisting of an unprivileged and a privileged specification. RISC-V targets a large diversity of CPUs, and therefore provides a set of options. First, CPUs may comply with the 32-bit or the 64-bit specification, which share most features. Second, CPUs may implement ISA extensions. In addition to the base ISA, common extensions are F (floating-point), D (double-precision support), M (integer multiplication and division), A (atomic operations), and C (compressed instructions). Compared with other established ISAs, RISC-V ISA features a small number of instructions. In particular, RISC-V requires two instructions to load an immediate 32-bit value into a register (`lui` followed `addi`). Furthermore, the only operations that influence the program control flow are `jal` (direct jump), `jalr` (indirect jump), branches, exceptions and privilege level changes. Note that in RISC-V, the targets of branches are immediates. In the case of self-modifying code, the `fence.i` instruction must be executed before executing newly-stored instructions.

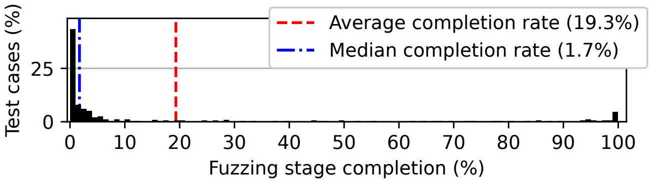


Figure 1: Completion rates of DifuzzRTL executions.

RISC-V, in its common implementation in open source CPUs, supports up to three privilege levels which are machine mode (M), supervisor mode (S) and user mode (U), in decreasing order of privilege. The M mode is the only mandatory privilege level. Upward privilege transitions are done through interrupts or exceptions, and downward transitions happen through specific instructions (`mret` and `sret`). Depending on the target privilege level, the architectural fetch address following an exception is pre-set in the `mtvec` or `stvec` Control and Status Registers (CSRs). CSRs are further used to configure how the CPU should operate in certain conditions, such as whether exceptions should be delegated to another privilege level, or if the floating-point unit is enabled. Spike [47] is a widely used open-source ISS for RISC-V.

3 Motivation and Challenges

In this section, we first analyze important aspects of the recently-published CPU fuzzers DifuzzRTL [32] and TheHuzz [35]. From these observations, we describe challenges which will guide the design of Cascade.

3.1 Observations

We collected 500 CPU inputs generated by DifuzzRTL [32] for the Rocket core to understand key aspects of test cases generated by a state-of-the-art CPU fuzzer. We also analyze the test cases generated by TheHuzz [35] based on the description in their paper since TheHuzz is not open source at the time of this writing.

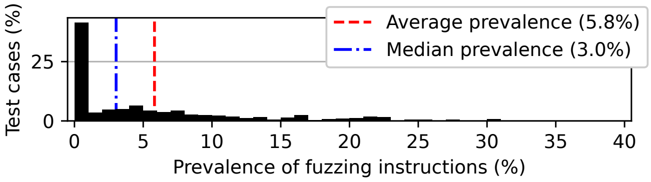


Figure 2: Prevalence of fuzzing instructions in DifuzzRTL.

Completion. Figure 1 shows the percentage of the fuzzing instructions that execute at least once in each test case produced by DifuzzRTL. We measure completion on the ISS, assumed bug-free. We observe that these programs do not generally complete the execution of their fuzzing sections, mostly because of the difficulty to predict intermediate values that affect the control flow, such as operands of branch instructions. Similarly, TheHuzz limits its test cases to 20 instructions, (10 of the first being non-control-flow instructions), again, because it is difficult to control the behavior of control-flow instructions when fuzzing.

Observation 1. Completion: CPU fuzzers struggle with fully executing their test cases.

Prevalence. In the programs generated by DifuzzRTL, we separate the instructions in two categories: *overhead* and *fuzzing* instructions. *Overhead* instructions are generic, non-randomized instructions such as setup routines or hard-coded exception handlers, while *fuzzing* instructions are the ones actually randomized. We define *prevalence* as the proportion of executed instructions that are *fuzzing* instructions. Figure 2 shows the prevalence in the programs generated by DifuzzRTL. Strikingly, only a small proportion of the executed instructions are fuzzing instructions. Similarly, the very short fuzzing sequences of TheHuzz are preceded by an overwhelming amount of configuration (overhead) instructions [35, 46].

Observation 2. Prevalence: most executed instructions correspond to overhead and are not fuzzing.

Conclusion. The existing coverage-guided approaches are insufficient. They produce malformed non-completing programs dominated by overhead instructions.

3.2 Overview of challenges

Our analysis suggests that we might significantly improve fuzzing results by constructing programs that address these limitations. We provide an overview of the challenges based on our previous observations and how we address them in the rest of this paper. The first challenge concerns completion and prevalence of the generated programs.

Challenge 1. How to generate programs that complete and have a high fuzzing prevalence?

Section 4 discusses a new design for program construction. Longer programs have a higher prevalence, but only if they complete. We propose to build *valid* programs that are expected to complete, by pre-defining a control flow ahead of execution. This is in contrast with existing CPU fuzzers that rely on mutation-based test-case generation strategies. The result of this step is a set of *intermediate* programs that have instructions with complex data flows, but control flows that are not dependent on the (complex) data flows. Our next challenge is to make the control flows dependent on the complex data flows while ensuring completion. Entangling data flows into control flows has two major benefits. First, it helps finding bugs related to (speculative) control flows. Second, it enables transforming a data-flow bug symptom into a program non-termination, effectively providing a non-pervasive design-agnostic way of detecting data-flow bugs in arbitrarily long and complex programs without any runtime overhead.

Challenge 2. How to generate valid programs with a high degree of dependence between data and control flows?

Section 5 proposes a novel method for efficiently constructing a complex data-dependent control flow for test cases, called asymmetric ISA pre-simulation. Asymmetric ISA pre-simulation mixes a highly randomized data flow with the control flow of the intermediate programs. This method is based on the new insight that instead of using an ISS for differential fuzzing, we can use it for generating a *valid* program. Repeated usage of the ISS for the ultimate program generation, however, would introduce a large performance penalty. We design a new scheme that allows us to reduce the number of ISS calls to only one per generated program.

Our prototype fuzzer implementation of the ideas presented in Sections 4 and 5, called Cascade, generates programs that trigger a large number of new bugs, more than any CPU fuzzer so far. The programs leading to these bugs, however, may be long and complex by construction, to an extent that makes it inconceivable to interpret them manually from logic waveforms, like it may have been done so far. This leads us to our last challenge.

Challenge 3. How to extract and interpret the bug from a complex program?

Section 6 discusses a new algorithm for iterative program reduction to find a minimal number of instructions inside program triggering CPU bugs. We successfully applied it for all the bugs found by Cascade, which significantly helped us when reporting the issues to the respective CPU maintainers.

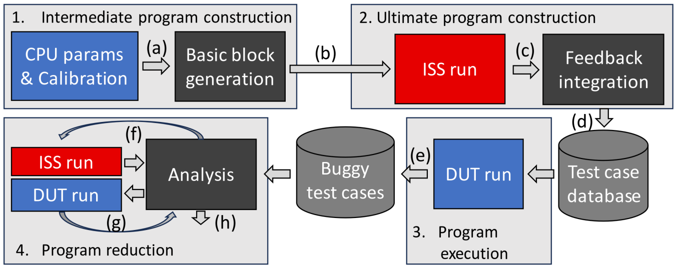


Figure 3: Overview of Cascade.

4 Design

We outline Cascade, before explaining the intermediate program construction. In Section 5, we explain how Cascade uses ISS feedback to entangle data and control flow.

4.1 Cascade overview

Figure 3 provides an overview of the fuzzing process and its components. Cascade proceeds in four steps: (1) intermediate program construction, (2) ultimate program construction, (3) program execution and (4) program reduction. The program construction (steps 1 and 2) is decoupled from the other steps, so the same program can be executed on multiple versions of the same CPU, or on CPUs with compatible extensions.

Intermediate program construction. Cascade takes as an input the supported ISA extensions, addresses for dumping registers and stopping the RTL simulation, and descriptions of known bugs to circumvent, and starts with a brief calibration stage (taking less than a second) to evaluate some CPU parameters, such as supported CSR bits (a). Generating programs is then a parallel task. Cascade first generates the *intermediate* program as a sequence of basic blocks, where control flow is isolated from the data flow (b).

Ultimate program construction. An ISS then executes the whole *intermediate* program once to collect the data-flow dependent values of registers (c). Cascade uses this information to entangle the data flow with the control flow, as explained in Section 5 to produce the *ultimate* program (d). It is defined as a triple (ELF file, descriptor, and the expected final register values (optional)), where the descriptor is a short identifier that permits re-generating the same program.

Program execution. The program execution is also a parallel task. Each ultimate program is executed independently on a simulated design under test. The output (e) is a pair (descriptor, success). The descriptor is the same as in (d). The success flag is raised if the program terminated successfully, and optionally if the dumped register values match.

Program reduction. The program reduction phase takes as input a test descriptor that leads to an execution failure and produces a reduced program that preserves the buggy behavior while reducing the program complexity as described in Section 6. The analysis consists in iteratively reducing the program (f) and re-running it to check if the bug is still triggered (g), and to produce a minimal program that is easy to understand (h), as explained in Section 6.

Listing 1: Example basic block.

```
xor       x3,x4,x9
csrrwi    x9,mcause,15
beq       x9,x4,0x8000098e
fld       f8,(x3)
feq.s     x4,f9, f8
jalr      x9,(x7)
```

4.2 Intermediate program construction

We first explain the structure of the programs generated by Cascade. We then show how Cascade selects instructions to form basic blocks. We finally discuss the memory management mechanism for ensuring sound program construction.

4.2.1 High-level program structure

Basic blocks. A program is a sequence of instructions within basic blocks. Basic blocks are made of zero or more instructions that do not affect the control flow, followed by a single instruction that affects the control flow. Accordingly, Cascade constructs programs as sequences of basic blocks, where the last instruction of a basic block steers to the next basic block. Each basic block is placed at a random location in memory (see Section 4.2.3). The order of basic blocks' execution is not necessarily identical with their placement in memory. The memory outside of basic blocks and of their data is left uninitialized, defaulting to zeros in simulation.

Initial and final basic blocks. All programs start with an initial basic block that sets up an initial state and random register values before jumping to the first fuzzing basic block, which eventually jumps to the next, and so on. The program ends with a final basic block that optionally dumps register values, and sends a signal indicating that the program's end was reached. No overhead instruction, as defined in Section 3, is executed between the initial and final basic blocks.

4.2.2 Basic block generation

Control-flow behaviors. We call instructions that do not change the control flow of a given program *still* instructions, and others *hopping* instructions. For example, a branch can be taken (still) or non-taken (hopping). Similarly, a memory load instruction such as `lw` can succeed (still) or raise an exception (hopping). We define the instructions that may be still or hopping depending on register values, or that are unconditionally hopping but whose destination depends on register values, as *cf-ambiguous*. For example, `beq` and `lw` are *cf-ambiguous*, but add and illegal instructions are not. Listing 1 shows an example basic block, where cf-ambiguous instructions are represented in red.

Picking instructions. Instructions are grouped in categories, which are listed in Appendix A. Some groups of instructions will behave in the same context-dependent way, for example all floating-point instructions will be conditioned by whether an FPU is present and activated. Hence, Cascade picks instructions hierarchically, by first choosing a category, and then a specific instruction. Both are chosen randomly with certain probabilities, which are varied between programs. When picking a cf-ambiguous instruction, Cascade chooses immediately whether it must be still or hopping. Cascade biases the choice of operands by granting higher probabilities to registers recently used as outputs.

In particular, Cascade supports complex FPU operations and CSR interactions. It additionally supports exceptions and privilege switches as simple hopping instructions, which can only be picked under certain architectural conditions that are generated by the mechanism explained in Section 5. Ultimately, the complex data flow originates from combining initial static data with a diverse stream of random instructions.

Circumventing known bugs. Ideally, discovered bugs should be fixed immediately. In reality, however, this may take time and effort. Due to limited human resources, some bugs may remain unfixed for a long time, polluting bug reports and potentially restrain test case continuation. Known bugs may be ignored after triaging the causes of bug reports. However, known CPU bugs can influence the control flow of Cascade programs early, shadowing the rest of the program. Furthermore, triaging is an expensive operation that must then be repeated many times for the instances of the same bug. Instead, Cascade allows circumventing known bugs by constructing programs that will not trigger them. By increasing the completion rate, Cascade is able to progress and find more bugs faster. As an example, in the Rocket core, the retired instruction performance counter ignores ecall and ebreak instructions (R1) [1]. The circumvention configures Cascade to avoid that generated programs read this counter after these instructions until it has been written again, hence it covers exactly the bug.

Note that Cascade's approach of circumventing certain bugs may result in the under-exploration of the components in which the bugs exist. Automatically circumventing bugs via automated RTL patching is an interesting future direction that can address this issue altogether.

4.2.3 Memory management

To produce sound programs, Cascade imposes some constraints on the memory layout of the generated programs. In particular, Cascade ensures that (a) instructions do not overlap, (b) store operations do not overwrite instructions, and (c) later transformations of the program (discussed in Sections 5 and 6) do not have unintended effects. Intuitively, the constraint (b) would prevent from detecting some potential bugs related to self-modifying code. RISC-V requires self-modifying code to use fence.i, which means that such bugs are limited in scope. We leave the exploration of such bugs as future work and instead focus on non-self-modifying code.

Progressive memory allocation. Cascade allocates memory on the fly for each new instruction. Whenever a hopping instruction is generated, space is allocated for the first instruction in the next basic block, at a reachable random new address that offers enough space for a new basic block. Initially, space is allocated (at random locations) for the initial and final basic blocks, which have known upper bounds in length. Additionally, to anticipate program reduction, Cascade allocates space for a *context setter* block used for program reduction and leaves it empty (details are discussed in Section 6).

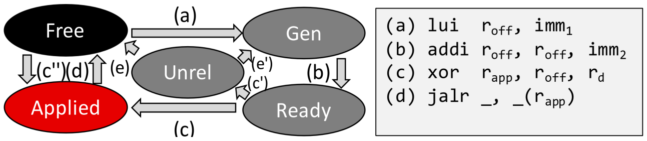
Strong memory allocation. The memory allocator can *strongly* allocate memory areas, i.e., forbid loads from there. Reading from a memory location could entangle its data with the data and control flows. Strong allocations prevent reading from a memory location that stores very specific parts of the program where some instructions differ between the *intermediate* and *ultimate* programs as described in Section 5.

Memory operations. Memory stores, randomized in number and size, can only target some specific memory areas, allocated at the start of the program's creation. Memory loads can target any address, except for the strong allocations. Heuristically, we bias memory loads to target more often memory areas that have been recently written, although so far, this specific heuristic has not been critical in finding bugs.

5 Ultimate Program Construction

The fundamental idea behind asymmetric ISA pre-simulation is to execute an *intermediate* program on the ISS to collect feedback for constructing the *ultimate* program with a control flow that is identical but dependent on the data flow.

Steering the control flow. There are three schemes for generating an arbitrary, but controlled, control flow. The first scheme is not to control the values used by cf-ambiguous instructions such as jalr but to place the next basic block accordingly. This scheme is not viable because most addresses are inaccessible or already allocated. The second scheme is not to involve the random data flow and rely on direct branches or registers loaded with fixed values. This is how the control flow of the *intermediate* program is constructed. The third scheme entangles data and control flow by observing the data flow and applying an offset to a register used by a cf-ambiguous instruction. We rely on the third scheme to construct the control flow of the *ultimate* program.



```
(a) lui  r_off, imm_1
(b) addi r_off, r_off, imm_2
(c) xor  r_app, r_off, r_d
(d) jalr _, _(r_app)
```

Figure 4: Life cycle of registers regarding offset construction. The instructions can be separated by other (unrelated) instructions.

5.1 Offset construction and register lifecycle

Some cf-ambiguous instructions such as indirect jumps (`jalr`) require a specific operand value `val` that Cascade intends to impose. Following the principle of dependency preservation, we let `val` depend on a randomly picked *dependent register* r_d. Since we do not want to constrain r_d's value, r_d is generated by the random data flow and its value is unknown when we pick the cf-ambiguous instruction. Hence, to calculate `val` we propose to generate an *offset register* value to be eventually combined with r_d's value. The combination of r_d and r_{off} is performed by an *offset applier* instruction, for instance `xor`, whose output is defined as the *applied register* r_{app} and holds the intended value `val`.

Branches. Because *applied registers* are a somehow precious resource, Cascade does not use this method for branches. Instead, it uses the ISS feedback to obtain the operand values and selects a suitable branch opcode, depending on whether the branch should be still or hopping.

Registers involved. In total, this construction involves two registers (r_d, whose value is randomized by the program's data flow, and r_{off} to offset its value). Since there is no reason to specifically reuse r_d or r_{off} as an output of the offset applier, a third register could be used as r_{app}, in accordance with the principle of maximizing the degrees of freedom.

Offset state machines. This scheme implies that 1) r_{app} must be available for use when the cf-ambiguous instruction is picked, 2) r_{app}'s calculation requires that r_{off} is ready before the offset applier instruction, and 3) r_d must be available when the offset applier instruction is executed. To comply with these requirements, we maintain a simple state machine for each architectural integer register. The state machine is composed of five states: *free*, *under generation (gen)*, *ready*, *unreliable (unrel)* and *applied*, as illustrated in Figure 4. All registers initially start in the *free* state. Cascade can create instruction (a) to move a *free* register to state *gen*, or (b) to move a register from *gen* to *ready*. When there is at least one register, r_{off}, in the *ready* state, then Cascade can pick an offset applier instruction. If Cascade chooses to define $r_{app} = r_{off}$, then r_{off} is moved to the *applied* state (c). Else, r_{off} is moved to the *unrel* state (c') and r_{app} is moved to the *applied* state (c"). Once r_{app} is used by a cf-ambiguous instruction (d), it returns to the *free* state. An *unrel* register must not be used as an input to any instruction and may be overwritten by an instruction to become *gen* (e'), or by an ordinary instruction to become *free* (e).

Pre-simulation. Cascade relies on an ISS to determine the value of dependent registers when encountering a cf-ambiguous instruction. So far, the state of the art [8, 9, 14, 30, 32, 34, 35] always submits the same program that is given to the CPU to the ISS. If we imitated the state of the art, the ISS should be called every time a cf-ambiguous instruction is encountered because r_{app} would be random until the correct r_{off} can be computed from r_d. The ISS could not proceed to the next basic block (or complete a correct memory operation) without running at least once per cf-ambiguous instruction.

We introduce the *asymmetric* ISA pre-simulation method to address this critical performance bottleneck by requiring a single ISS execution. The fundamental insight is to provide to the ISS not the *ultimate* program that the CPU under test will execute, but an *intermediate* program such that (a) the values of all *dependent registers* (in the constructions of *applied registers*) are identical between the two programs and (b) the programs' control flows are identical and (c) in the *intermediate* program, no applied register depends on the randomized data flow.

Concretely, the *intermediate* program differs from the intended *ultimate* program in three aspects. When generating the *intermediate* program: 1) we choose the immediates `imm1` and `imm2` to set the value of r_{off} to `val`, 2) we substitute of the offset-applying instruction with a `mv` instruction, which copies r_{off} into r_{app}, and 3) we transform non-taken branches into NOPs, because the value of the operands are not yet known. Given these transformations, the *intermediate* program complies with the aforementioned requirements.

Intermediate register states. This scheme for offset construction guarantees by design that at any point in the program, all *free* and *applied* registers have the same value in the *intermediate* and *ultimate* program. This invariant does not hold for registers in the *gen*, *ready* and *unrel* states. Therefore, they should never be used as the input for any instruction other than the next instruction in the offset construction cycle, respectively steps (b) and (c) in Figure 4. Note that state machine transitions are a specific instruction category in Cascade, as enumerated in Appendix A.

5.2 Privilege transitions

Exceptions are a particular case of hopping instructions. They require some basic bookkeeping in the fuzzer to maintain the privilege state and delegation flags, which indicate the target privilege level active when some exception occurs.

We use one to two offset/dependent/applied register triples per exception. The first is used to populate the trap vector, either `mtvec` or `stvec`, depending on the current privilege state and whether the exception will be delegated. The second is used when an exception depends on a register value, e.g., in

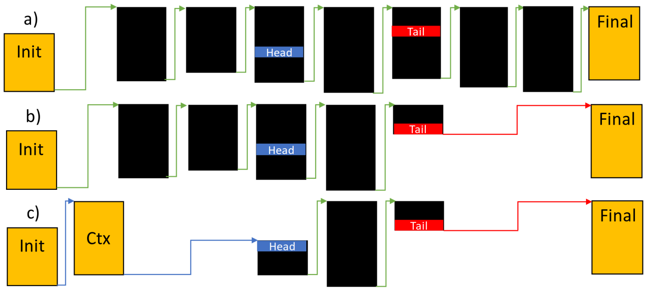


Figure 5: (a) Original, (b) tail-reduced and (c) fully reduced program. Black rectangles and arrows represent basic blocks and control flow. Ctx is the context setter block.

a misaligned memory load exception. It is populated similarly as an indirect jump or a load would be. Note that since basic blocks are generated on the fly, the value to be inserted into the trap vector is only known when the exception-triggering instruction is generated, which may be many basic blocks later; while this adds some necessary complexity to the fuzzer, it does not negatively affect the program's degrees of freedom in any way. The implementation of downward privilege transitions is in all respects comparable with exceptions.

6 Program Reduction

Cascade generates potentially long test cases, and CPU bugs are revealed by programs not terminating, thanks to the entanglement of the data and control flows. To understand the underlying CPU bug, it is necessary to reduce the programs to a minimal form, while preserving the instructions and states that trigger the CPU bug. We first show how to find the last instruction (*tail*) involved in triggering the bug, then the first (*head*). Figure 5 illustrates the program reduction process.

6.1 Identifying the bug's tail

We propose to reduce the program progressively by transforming some instructions into direct jumps that skip some of the last basic blocks and observing whether the bug is still triggered. The result is illustrated in Figure 5 (b).

Identifying the tail basic block and instruction. Cascade finds via a binary search the last basic block which, when omitted along with its successors, erases the buggy behavior. To remove such a final sequence, Cascade replaces its predecessor's hopping instruction with a direct jump toward the final block. Cascade then searches the bug's tail instruction in the converse way.

Failing control-flow instructions. If the tail instruction is a hopping instruction, the algorithm above will find a tail basic block B_n, but no tail instruction. This is because for skipping the basic block B_n but not B_{n-1}, a direct jump instruction toward the final block will replace the (bug-triggering) hopping instruction of B_{n-1}, hence removing B_n erases the buggy behavior. The tail instruction being a hopping instruction is hence the necessary and sufficient condition for failing.

6.2 Identifying the bug's head

Most often, a single instruction, provided with the correct architectural context, is sufficient to trigger the bug reliably. In such cases, finding the tail instruction is enough to understand the bug. However, some bugs required a sequence of instructions (up to 2), possibly far apart, to be triggered, such as the bugs V1-V9 and V14 that we describe in Section 7, because they rely on a specific microarchitectural context. Hence, for these bugs, identifying the head is necessary. The result of this step is illustrated in Figure 5 (c).

Maintaining the architectural context. Identifying the tail instruction can be done by exclusively inserting direct jump instructions in the right places iteratively, but identifying the head instruction is more challenging because removing predecessor instructions ahead may influence the architectural state. We leverage the following insight to identify the bug's head: we find the head by preserving the architectural state but simplifying the microarchitectural state. Concretely, only for this step, a *context setter* basic block is inserted, by replacing the initial block's hopping instruction. Once a candidate head basic block and instruction is chosen, the context setter uses the ISS to infer the architectural context, including for instance register values, privilege level and some performance counter values. It then loads the architectural state of the CPU, using simple instructions such as wide loads for register values, and basic instruction sequences for populating CSRs and setting the proper privilege. The detection of the bug's head is then performed similarly to the tail, following the converse algorithm where the head instruction is the first instruction that, when omitted, erases the buggy behavior.

Sandwich instructions. For all bugs found by Cascade so far, identifying the tail and head instructions has always been sufficient for understanding and reproducing the bugs. However, Cascade includes facilities to flatten the remaining instructions (in black in Figure 5 (c)) and removing them iteratively. Such transformations are not guaranteed to work on a given specific program, for example if the hopping instruction was an exception and some following instruction checks the exception cause. In practice, finding another program that reveals the same bug in case of failure of advanced transformations is sufficient, notably because finding new programs that trigger the same bug is fast, as we show in Section 7.6.3.

7 Evaluation

In this section, we evaluate Cascade in terms of performance, program metrics, coverage, and the discovered bugs. We use microbenchmarks to quantify the performance of program construction and compare it with previous work (Section 7.1). We evaluate the impact of program lengths on fuzzing throughput (Section 7.2). We then evaluate the program metrics for Cascade (Section 7.3) as we did for DifuzzRTL in Section 3.1. We compare the coverage achieved by Cascade according to multiple coverage metrics and compare it with the state-of-the-art fuzzers that specifically target these metrics (Section 7.4). We then show the bug discovery efficacy of programs of different lengths (Section 7.5). We also describe the 37 new bugs found in 5 of the 6 evaluated RISC-V CPUs and in Yosys, evaluate the time to detection (Section 7.6) and provide a full list in Appendix D. Finally, we evaluate the performance of program reduction (Section 7.7).

Evaluation setting. The performance results were obtained on a machine equipped with two AMD EPYC 7H12 processors at 2.6 GHz containing 256 logical cores and 1 TB of DRAM. We use Verilator 5.005 to simulate the CPUs' RTL. As an ISS, we use spike (version 1.1.1-dev, commit `fcbdbe79`). We use a recent version of a widely-used commercial simulator to collect simulator-based coverage similar to previous work [35]. We use the most recent versions of each CPU, where bugs are fixed or circumvented by Cascade. Appendix B provides detailed information about these designs. Notably, we tested Cascade on a variety of CPU complexities, from a simple minimal 32-bit integer core (PicoRV32) to an application-class Linux-capable out-of-order core (BOOM). We implemented Cascade as 6 k lines of Python code.

Baselines. We compare with the existing open-source generic CPU fuzzers, which are RFUZZ [36] and DifuzzRTL [32]. Despite the claim made in the original paper [35], TheHuzz is not open source at the time of this writing, and its authors were reluctant to answer any question or share their code over a period of a year, hence we rely on the results reported in their paper [35]. We re-implemented RFUZZ to support Verilog, and relied on the Docker image provided by DifuzzRTL [31].

7.1 Program generation performance

To quantify the performance of program construction, we measure the amount of time spent in intermediate program construction, asymmetric ISA pre-simulation and RTL simulation, for each CPU under test, over 24 hours of fuzzing. Figure 6 shows the results.

Results. While by construction, the duration of the instruction generation and asymmetric ISA pre-simulation is identical across designs, the RTL simulation time increases with the complexity of the design, hence the proportion of time spent generating programs largely decreases with the complexity of the designs. When generating the programs in real time, the program generation takes between 26.5% and 78.9% of the total fuzzing time.

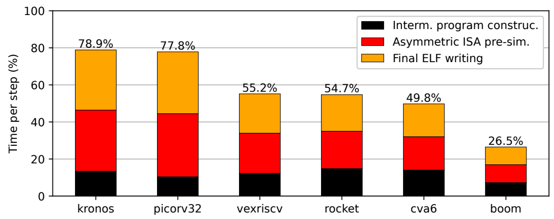


Figure 6: Performance of program construction. The rest of the time is spent in the RTL simulation.

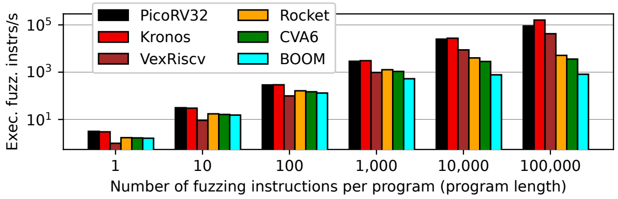


Figure 7: CPU-under-test execution throughput given program length. Note the logarithmic Y axis.

Input reuse. To make Cascade's evaluation as pessimistic as possible, we systematically dynamically generate new inputs by default. Note that by construction, Cascade's inputs are reusable across designs that share compatible ISA extensions, and across CPU generations. Hence the input generation is, in fact, a one-time cost that can further be amortized.

Runtime overhead. DifuzzRTL reports a runtime overhead of 6.1% to 6.9% for control register coverage, and 97% for multiplexer select coverage. TheHuzz reports a runtime overhead of 71% These slowdowns do not include input generation. Cascade incurs no runtime overhead by design.

7.2 Throughput of long programs

To understand the throughput boost provided by long programs, we measure the number of fuzzing instructions executed per second when controlling the number of instructions per program generated by Cascade, including program generation time. The experiment was run for 1 core-hour per bar. Figure 7 shows the throughputs of program execution, while Figure 8 details the average duration of a single program generation. Constructing very long programs requires managing a wider memory range, resulting in longer program generation times. Consequently, the overhead of generating longer programs counteracts the improvements in terms of fuzzing throughput when programs become too large.

To find the sweet spot for the length of programs, Figure 9 shows that the effective fuzzing throughput when considering both the raw fuzzing throughput and the overhead of program generation at the same time. These results show that programs of 10 k instructions generally provide the best effective fuzzing throughput, and that the fuzzing throughput is improved by three orders of magnitude between single-instruction and 10 k-instruction programs.

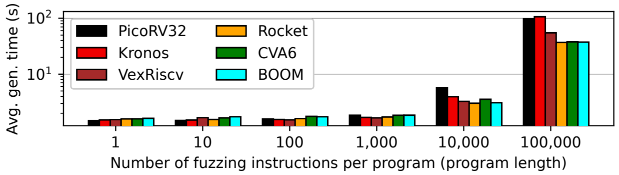


Figure 8: Program generation performance given program length. Note the logarithmic Y axis.

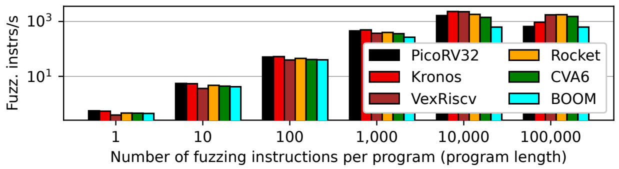


Figure 9: Effective fuzzing throughput given program length. Note the logarithmic Y axis.

7.3 Program metrics

As part of our initial observations in Section 3, we exposed the completion and prevalence program properties. We evaluate these metrics for Cascade. We additionally evaluate the length of dependency chains between fuzzing instructions, which matters for non-termination when a bug is triggered.

Prevalence and completion for Cascade. Since Cascade always completes except when finding a CPU bug, we observe the expected completion rate of 100%. Figure 10 shows the prevalence of fuzzing instructions for Cascade. The high prevalence is because programs are relatively long and fully randomized except the initial and final basic blocks.

Dependencies. We analyze dependency chains between fuzzing instructions. On top of entangling the data flow with the control flow to force non-termination when a bug is triggered, these dependencies can further exercise corner cases inside a CPU's design. For this analysis, we calculate the length of instruction dependency chains. Each register starts with a dependency number of -1, which gets reset when it is a destination of a CSR read or of an instruction that only takes immediates. We calculate an instruction's dependency number as the maximum of the dependency numbers of the source registers, plus one, which also becomes the new dependency number of the destination register.

Figure 11 and Figure 12 show the distribution of the length of the dependency chains for DifuzzRTL and Cascade, respectively. The fuzzing instructions with zero dependencies here are instructions in the form of `xor rd, rd, rd`. The programs generated by DifuzzRTL have very few interdependencies. Similarly, TheHuzz does not explicitly generate or favor programs with dependencies [35], hence we expect even lower numbers. In contrast, Cascade generates programs with longer dependency chains, which could further be improved if needed by lowering the probabilities of picking dependency-resetting instructions such as CSR reads.

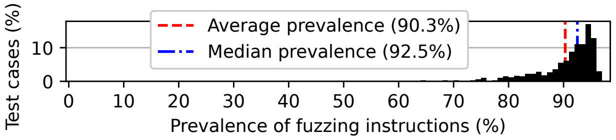


Figure 10: Prevalence of fuzzing instructions for Cascade.

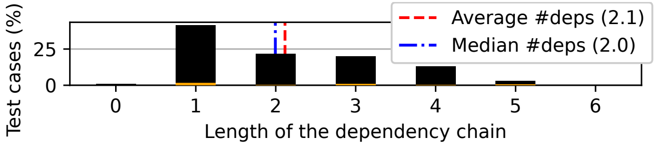


Figure 11: Length of the dependency chains for DifuzzRTL. Control-flow instructions are represented in orange.

7.4 Coverage evaluation

We show that Cascade is faster in increasing coverage compared to the state-of-the-art fuzzers with their own coverage metrics. We consider the open-source coverage metrics used for fuzzing (i.e., multiplexer select and control register coverage). To compare with the results reported by TheHuzz [35], we additionally consider the coverage metrics provided by the commercial simulator (branches, conditions, expressions, FSM states and transitions, statements, and toggles).

7.4.1 Control register coverage (DifuzzRTL)

We first compare the control register coverage of Cascade with the one achieved by DifuzzRTL, which explicitly aims at maximizing this coverage. DifuzzRTL supports legacy versions of Rocket and BOOM. Running Cascade on this BOOM version leads to quasi-systematic timeouts, revealing old bugs in the obsolete BOOM RTL. Hence, we executed all test cases on the legacy Rocket core provided in the Docker image [31], which was already exempt of unexpected bugs.

Results. Figure 13 shows the control register coverage achieved by Cascade and DifuzzRTL. Cascade achieves more coverage than DifuzzRTL, in a shorter time. In particular, Cascade achieves the same coverage in 30 minutes (15 minutes when using a pre-generated corpus) as DifuzzRTL in 48 hours, which is a speedup of 97x (respectively 186x).

7.4.2 Multiplexer select coverage (RFUZZ)

We now compare the multiplexer select coverage achieved by Cascade with the one achieved by RFUZZ. We adapted RFUZZ as a sequence of Yosys [60] passes to support Verilog, more general than FIRRTL [33], and ensured that the results match with the original open-source implementation. We will open-source the instrumentation code to foster further research. We execute Cascade on the RFUZZ-instrumented versions of the designs as well. We execute the RFUZZ experiments until completion, i.e., as implemented in the original RFUZZ fuzzer, when all input sequences in the adherence

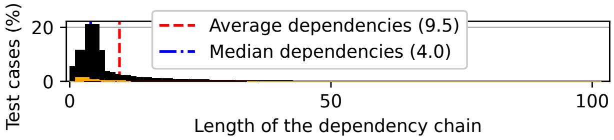


Figure 12: Length of the dependency chains for Cascade. Control-flow instructions are represented in orange. Fuzzing instructions depended on up to 270 other fuzzing instructions.

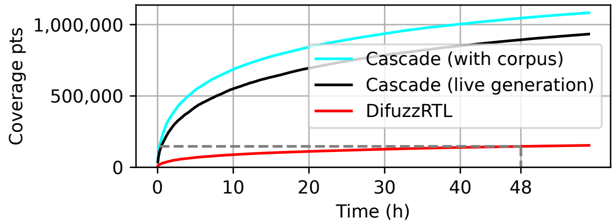


Figure 13: Achieved control register coverage.

to the corpus cease to increase coverage. We had to exclude CVA6 because of the Yosys bug found by Cascade.

Results. Figure 14 shows the multiplexer select coverage achieved by Cascade and RFUZZ. First, in terms of coverage, for the two simpler designs, RFUZZ is a bit slower than Cascade (note that in this experiment, Cascade also suffers from the runtime overhead due to the instrumentation), but eventually achieves a superior coverage. Since RFUZZ is a lower-level fuzzer, it is expected to be able to toggle some more multiplexer signals eventually, for example by fuzzing the bus protocol, whereas an ISA-level fuzzer like Cascade and many others [8, 9, 14, 30, 32, 34, 35] will not explore these state machines. However, as soon as CPUs become more complex, RFUZZ is unable to make any progress.

Second, regarding delay, the order of magnitude to saturate coverage for Cascade is approximately 100 seconds, which corresponds to its order of magnitude for finding bugs as we show later. This suggests that, although a naive RFUZZ implementation seems inadequate to finding bugs in CPUs, especially when they are complex, its coverage metric seems relevant to evaluating the quality of a fuzzing campaign.

7.4.3 Simulator-based coverage (TheHuzz)

We compare the simulator-based coverage achieved by Cascade with the one reported by TheHuzz. We reproduce the experiment from TheHuzz that led to their Figure 5 [35]. We compare the simulator coverages of Cascade and DifuzzRTL, and use DifuzzRTL as a pivot to compare with TheHuzz. Note that to fit the methodology of TheHuzz, both DifuzzRTL and Cascade rely on pre-generated corpuses in this experiment.

Results. Figure 15 shows a per-instruction speedup of Cascade over DifuzzRTL of 27x for the simulator-collected coverage, while TheHuzz reports a speedup of 3.33x. Since TheHuzz reports a runtime slowdown of 71%, Cascade is 28.2x faster than TheHuzz on TheHuzz's target coverage metric.

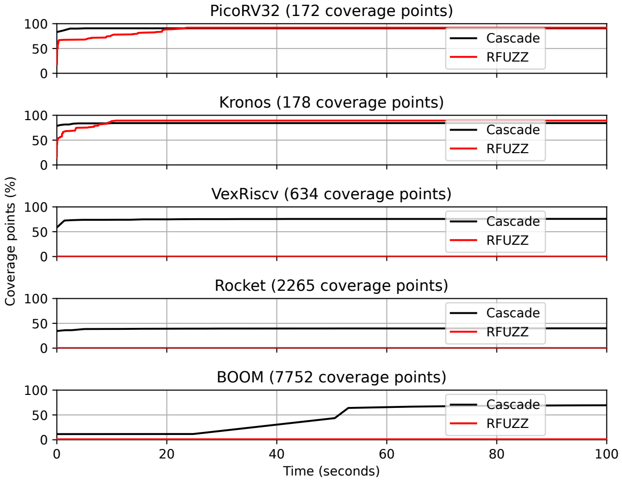


Figure 14: Multiplexer select coverage achieved by Cascade.

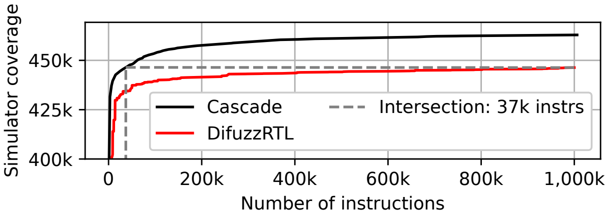


Figure 15: Simulator-collected coverage per instruction of Cascade and DifuzzRTL. Note that the y-axis starts at 400k.

7.5 Efficacy of long programs

To better understand the influence of program length on the efficacy of finding bugs, we enforce the number of fuzzing instructions per program, fuzz on 64 cores and report the bug discovery times in Figure 16. Even limited to a single fuzzing instruction, Cascade discovers C8-C9 in a core-hour, undetected by TheHuzz [35] and HypFuzz [14]. Yet clearly, longer programs are more efficient at finding bugs. Also note that some bugs are hard to separate for this measurement, in particular C2-C7 and C10, and may overlap. The bug found only by the longest programs in 24 core-hours is K3.

7.6 Bug discoveries

Cascade discovered 37 new bugs in 5 CPUs and 1 bug in the Yosys synthesizer. Figure 17 classifies the new bugs in six categories, and Appendix D provides more information in terms of description, CWEs and CVEs. We first analyze the discovered bugs and their security implications, and then evaluate the bug detection performance of Cascade.

7.6.1 Bug descriptions

Exceptions. Cascade discovered 11 exception-related bugs in 4 designs, which we define as missing or spurious exceptions. For example, in VexRiscv, interactions with a disabled

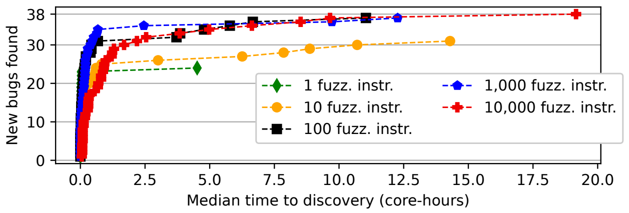


Figure 16: Time to reveal each bug when fuzzing with fixed numbers of instructions on 64 cores.

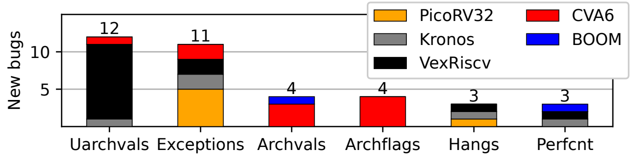


Figure 17: Discovered CPU bugs. Exceptions: missing or spurious exceptions. Uarchvals: wrong computations under microarchitectural conditions. Archvals: systematic wrong computations. Archflags: wrong status flags. Hangs: CPU hangs. Perfcnt: wrong performance counter values

FPU are wrongly permitted (V10, V11). In Kronos, writes to a non-existent CSR fail to trigger an exception (K3). In PicoRV32, Kronos and CVA6, spurious exceptions may be triggered by some CSR accesses (P2-P5, C8-C9) or incorrectly decoded valid instructions (P6, K5).

Microarchitectural-state-dependent wrong computations. Cascade discovered 12 bugs that produce wrong computations under certain microarchitectural conditions (*Uarchvals*) in Kronos and VexRiscv. In Kronos, a bug in the hazard detection unit (K1) causes, under some conditions, wrong register forwarding in a read-after-write-after-write double-hazard, where it forwards the first written value instead of the second. In VexRiscv and CVA6, wrong calculations occur under some microarchitectural FPU conditions (V1-V9, V14, C10). Such bugs are often hard to fix. Indeed, the VexRiscv maintainers proposed a fix, which solved most of the occurrences, but Cascade discovered a way to tamper with the FPU's microarchitectural state again with a different approach (V8, V9, V14), which has ultimately been fixed by the maintainers.

Systematic wrong computations. Cascade discovered 4 bugs in BOOM and CVA6 that produce wrong output values regardless of the microarchitectural state (*Archvals*). In BOOM, double-precision divisions and square roots ignore the (immediate) static rounding mode (B1). In CVA6, we found two wrong output sign bugs (C1, C6), and one unexpected infinity bug (C7).

Systematic wrong flags. Cascade discovered 4 incorrect flag bugs in CVA6 (*Archflags*) (C2-C5). Since flags are typically used to guide a control flow, emitting FPU operations that will set flags incorrectly may provide an illegitimate control flow influence. Such bugs are difficult to fix. The CVA6 maintainers contacted us to test a fix, which we found to fix some bug (C2), but preserving another (C3).

Hangs. Cascade discovered 3 bugs that cause hangs in Kronos, PicoRV32 and VexRiscv (V12, P1, K2). In PicoRV32, the hang is systematic for accessing some CSR addresses. The hang in Kronos, also related to CSR access, depends on the microarchitectural state. The hang in VexRiscv is achievable when speculatively executing an illegal compressed floating-point instruction and later executing a legitimate floating-point instruction: microarchitectural resources are reserved but are never released, which results in a deadlock. The latter bug is similar to the recently-discovered Zenbleed bug [39], where an architectural bug leaves traces after speculation.

Performance counter inaccuracies. Cascade discovered 3 inaccurate performance counter bugs (*Perfcnts*) in Kronos, VexRiscv and BOOM (K4, V13, B2). They incur an offset in the retired instruction counters when written by software.

Yosys logic synthesizer bug. Yosys [60] is a popular open-source logic synthesizer, which is typically used for instrumenting a CPU [32, 36, 48, 54]. It may also be used to part of an emulation or ASIC flow. Cascade found a bug (Y1) that leads to incorrect logic synthesis of CVA6's FPU.

Other findings. Besides discovering new bugs in CPUs and in Yosys, Cascade found known bugs in CVA6, and the performance counter inaccuracy in Rocket discovered by DifuzzRTL and reported by TheHuzz and HypFuzz [14, 32, 35].

Additionally, Cascade found two issues related to X over-propagation in VexRiscv (shadowing of RTL logical signals by undefined values [33, 56]), which may happen under some conditions. First, some uninitialized instruction cache lines can enter the pipeline and pollute signals. Second, in case of FPU underflow, a hardware table is accessed with a too large index. HypFuzz [14] discusses the implications of such vulnerabilities when describing their third vulnerability [14].

7.6.2 Security implications

We classify the security implications of the discovered bugs into six categories, as presented in Figure 19. The same bug may belong to more than one category.

Data-flow integrity. We define data-flow integrity bugs as allowing a malicious time-shared user to cause a victim entity to execute computations with a wrong result. Cascade found 12 such bugs, where preparing the (micro)architectural state can cause wrong computations (V1-V9, V14, K1, B1). For example, exploiting the BOOM dynamic rounding bug (B1), an attacker process can force a victim process to take a different floating-point rounding mode than expected.

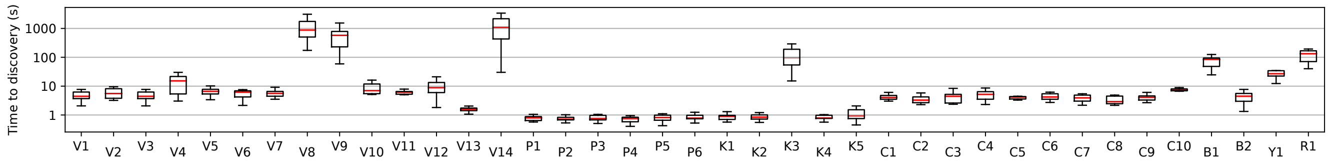


Figure 18: Time to reveal each bug when fuzzing on 64 processes. R1 is the known instruction counting bug in Rocket. Note the logarithmic scale. Bugs are labeled as follows: B (BOOM), C (CVA6), K (Kronos), P (PicoRV32), V (VexRiscv), and Y (Yosys).

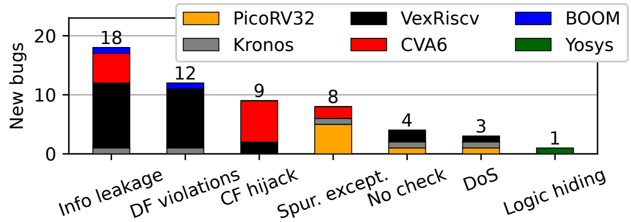


Figure 19: Security implications of discovered bugs. Note that some bugs belong to more than one category.

Information leakage. Sources of information leakage are many. For example, data-flow integrity bugs may also enable information leakage in the opposite direction, from the victim to an attacker running on the same core. So do bugs that illegally set some flags (C2-C5), and some unauthorized accesses (V10, V11), creating core-local side channels.

Control-flow hijack. Cascade found 9 bugs that let an attacker process influence the control flow of a victim process (C1-C7, V5, V6). For example, CVA6 sets wrong flags in diverse situations (C2-C5) and produces wrong finitude or sign results (C1, C6, C7), which typically influence the control flow. Concretely, we found that the inexact flag is not set in some cases of underflow or overflow (C4), preventing some potential security checks. On VexRiscv, the microarchitecture can be prepared to corrupt comparisons, for example making two equal registers be considered distinct (V5, V6). This preparation must be done by a process on the same core.

Spurious exceptions. Cascade found 8 spurious exception bugs (P2-P6, K5, C8, C9), which break isolation boundaries from higher privilege levels, for example, in the context of trusted execution environments. For example, in PicoRV32, reading some mandatory CSRs causes exceptions (P2), hence, a malicious machine could give the illusion that they are accessible, but actually the machine solely emulates the interactions, providing arbitrary values.

Missing checks. Cascade found 4 missing checks, with several implications. First, they may permit to bypass security checks (V10, V11), problematic, for example, if the FPU is time-shared with other cores. Second, they can deceive a victim to believe that a feature is supported (K3, P5). They can additionally be exploited to escape program analysis, where supposedly dead code is shadowed by an exception.

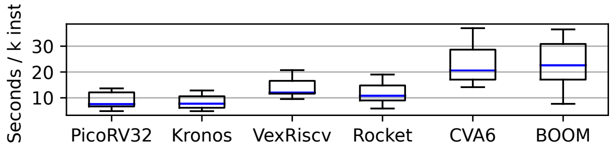


Figure 20: Program reduction performance.

Denial of service. All the hangs that we found (V12, P1, K2) can happen at any privilege level, leading to DoS attacks.

Logic hiding. Using the discovered Yosys bug (Y1), a malicious contributor can inject bugs into a design while submitting an apparently innocuous RTL design, by transforming the RTL into an equivalent one that will trigger Y1 accordingly.

7.6.3 Bug detection performance

We fuzz on 64 processes for 24 hours and summarize the time to discover each bug in Figure 18, where we repeated the discovery 10 times with different seeds. Note that some bugs are hard to separate for this experiment, in particular C2-C7 and C10, and may overlap. In most runs, bugs are discovered in less than 18 core-hours (17 minutes). Design and bug complexities influence the discovery time.

7.7 Program reduction performance

We evaluate the performance of Cascade's program reduction. Since all designs still have at least one non-fixed bug (V14, P6, K4, C9, B2 and R1), we focus on them for this analysis. We run the fuzzer to collect 100 programs that trigger the bug in each design. Then, we perform the program reduction on these programs, and measure the time required to reduce each program and normalize it by the number of instructions in the program, including the initial and final blocks.

Results. Some program reductions could fail, e.g., because a load instruction would target the hopping instruction of a basic block preceding a candidate tail block during the reduction phase and its result would propagate to the control flow. Experimentally, we did not observe such failures. Figure 20 shows the time to reduce each program, normalized by the length of the program. program lengths varied between 193 and 62,870 instructions. The timeout upper bound, conservatively set to $30 \times n_{instructions} + 1000$ cycles, influences program reduction performance. In this specific experiment, all programs were reduced to a single instruction.

8 Discussion

We discuss porting Cascade to other ISAs and limitations of coverage metrics used by the state-of-the-art CPU fuzzers.

Adaptations to other ISAs. While the fundamentals of Cascade's approach comply with most widespread ISAs, its current implementation is RISC-V-specific. Adaptation to RISC ISAs such as ARM and MIPS requires ISS and instruction generator adaptations. We expect porting Cascade to CISC ISAs to be more challenging due to more instructions.

Coverage metrics. The coverage metrics used by the state-of-the-art CPU fuzzers such as TheHuzz [35] and DifuzzRTL [32] are dominated by ineffective terms while introducing runtime overhead. For TheHuzz, toggle coverage represents around a million points and 89% of the achievable points on Rocket. Given that TheHuzz considers the sum of all coverage points of all types, any mutation that would produce such a single new toggle would be considered as discovering new coverage. Limitations of such simple metrics are well-known [53], yet the more powerful alternative, functional coverage, requires significantly more effort [7]. This explains why HypFuzz, building on the same metric as TheHuzz, found a single new bug (see Appendix C). DifuzzRTL defines control registers as registers which control a multiplexer inside the same HDL module. This coverage metric continues to increase when exploring registers whose fanout multiplexers already took all values. Ultimately, DifuzzRTL optimizes for exploring an immense number of combinations of values for registers that are mainly arbitrarily selected [12].

9 Related Work

In the recent years, hardware fuzzing has flourished. We first cover generic hardware fuzzers, then CPU fuzzers.

Generic hardware fuzzers. RFUZZ [36] is a generic hardware fuzzer that relies on multiplexer control signals to generate inputs based on AFL [29]. DirectFuzz [10] targets RFUZZ toward specific modules. Trippel et al. [55] proposed to fuzz the hardware simulation binary itself, using software fuzzing methods. Ruep et al. [43] proposed a fuzzer for SpinalHDL designs. Li et al. [37] proposed a new coverage metric for fuzzing. Ragab et al. [41] proposed a distance-to-target feedback metric to direct fuzzers towards specific targets. None of these publications reported the discovery of any bugs.

CPU fuzzers. Many active open-source repositories rely on testing tools, such as the RISC-V compliance suite [46], which generates basic unit tests, and RISCV-DV, a UVM-based testing framework based on commercial simulators [2]. To address their insufficiencies, several projects proposed fuzzing CPUs. DifuzzRTL [32] generates instructions and collects control register coverage to guide the fuzzing process and discovered 16 new bugs. Its source code, while relying on obsolete languages, is available for fuzzing legacy versions of Rocket and BOOM [31]. ProcessorFuzz [12] is a concurrent work that generates instructions and collects coverage of control and status registers on the ISS and reported the discovery of 9 new bugs (see Appendix C), including 7 on BlackParrot, maintained by certain authors of ProcessorFuzz. Kabylkas et al. present Dromajo and Logic Fuzzer [34] and report the discovery of 13 new bugs, including 4 with the help of Logic Fuzzer. Bruns et al. [8] exploited ISS coverage to find new VexRiscv bugs. Similarly, Herdt et al. [30] exploited ISS coverage to find 10 new bugs in their own core. N. Bruns et al. [9] generate an infinite instruction stream on the memory channel to find a bug in their own core. Such an infinite instruction stream is known to cause compatibility issues due to strong microarchitectural assumptions [8]. Kande et al. [35] proposed TheHuzz, based on commercial RTL simulator coverage feedback, and fond 7 new bugs. Chen et al. [14] proposed a technique to speed up the coverage obtained by TheHuzz and found one new bug (see Appendix C). Cascade is the first fuzzer to take a constructive approach to tackle the observations and challenges exposed in Section 3.

10 Conclusion

We presented Cascade, a CPU fuzzer based on the explicit construction of intricate RISC-V programs. Cascade is effective: it finds 37 new bugs on 5 RISC-V CPUs with varying degrees of complexity and vastly outperforms the state-of-the-art coverage-guided CPU fuzzers. What sets Cascade apart from the state of the art is its ability to efficiently construct long complex programs that enable high-throughput CPU fuzzing while terminating by design. Any non-termination signifies the discovery of a bug in the target CPU. We described how Cascade generates its programs using a new technique, asymmetric ISA pre-simulation, which enables Cascade to efficiently entangle an arbitrarily-complex control flow with an arbitrarily-complex data flow. Since the bug-triggering programs generated by Cascade may be long and complex, we introduced a new technique for program reduction transforming a program to the only few instructions that trigger the CPU bug.

Ethical considerations. We reported all bugs to their respective maintainers, and proposed fixes when our understanding of the language and design was sufficient.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback, Tobias Kovats for his contribution to RFUZZ re-implementation, and the maintainers of the designs we tested for their support in understanding and fixing some of the bugs. This work was supported in part by a Microsoft Swiss JRC grant and by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] Chips Alliance. Ebreak instruction retires. https://github.com/chipsalliance/rocket-chip/issues/2672. [Online; accessed 4-June-2023].

[2] Chips Alliance. Risc-v dv. https://github.com/chipsalliance/riscv-dv. [Online; accessed 4-June-2023].

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *UC Berkeley*, 2016.

[4] K. Asanovic, D. A. Patterson, and C. Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, UC Berkeley, 2015.

[5] K. Asanovic, D. A. Patterson, and C. Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. *UC Berkeley*, 2015.

[6] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz. Grimoire: Synthesizing structure while fuzzing. In *USENIX SEC*, 2019.

[7] T. Bojan, M. A. Arreola, E. Shlomo, and T. Shachar. Functional coverage measurements and results in post-silicon validation of core™ 2 duo family. In *2007 IEEE HLDVT*, 2007.

[8] N. Bruns, V. Herdt, D. Große, and R. Drechsler. Efficient cross-level processor verification using coverage-guided fuzzing. In *VLSI*, pages 97–103, 2022.

[9] N. Bruns, V. Herdt, E. Jentzsch, and R. Drechsler. Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging. In *DATE*, 2022.

[10] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021.

[11] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi. Processorfuzz: Guiding processor fuzzing using control and status registers. *arXiv:2209.01789*, 2022.

[12] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *HOST*, 2023.

[13] C. Chen. Miss illegal instruction exception when rd of mulhu is the same as rs1 or rs2. https://github.com/openhwgroup/cva6/issues/885#issuecomment-1469547149. [Online; accessed 4-June-2023].

[14] C. Chen, R. Kande, N. Nyugen, F. Andersen, A. Tyagi, A. R. Sadeghi, and J. Rajendran. Hypfuzz: Formal-assisted processor fuzzing. *arXiv:2304.02485*, 2023.

[15] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu. Fot: A versatile, configurable, extensible fuzzing framework. In *ACM FSE*, 2018.

[16] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *IEEE SP*, 2018.

[17] C. Courbet. Nsan: a floating-point numerical sanitizer. In *ACM SIGPLAN CC*, 2021.

[18] S. Deng, D. Gümüşoğlu, W. Xiong, S. Sari, Y. S. Gener, C. Lu, O. Demir, and J. Szefer. Secchisel framework for security verification of secure processor architectures. In *HASP*, 2019.

[19] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton. Isadora: Automated information flow property generation for hardware designs. In *ASHES*, 2021.

[20] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz. Gap-free processor verification by s2qed and property generation. In *DATE*, 2020.

[21] K. Devarajegowda, E. Kaja, S. Prebeck, and W. Ecker. Isa modeling with trace notation for context free property generation. In *DAC*, 2021.

[22] LLVM Developers. Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Online; accessed 4-June-2023].

[23] Z. Y. Ding and C. Le Goues. An empirical study of oss-fuzz bugs. In *MSR*, 2021.

[24] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. Evolutionary grammar-based fuzzing. In *SSBSE*, 2020.

[25] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *DAC*, 2020.

[26] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++ combining incremental steps of fuzzing research. In *WOOT*, pages 10–10, 2020.

[27] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, 2009.

[28] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ASPLOS*, 2008.

[29] Google. American fuzzy lop. https://github.com/google/AFL. [Online; accessed 4-June-2023].

[30] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler. Efficient cross-level testing for processor verification: A risc-v case-study. In *FDL*, 2020.

[31] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. https://github.com/compsec-snu/difuzz-rtl. [Online; accessed 4-June-2023].

[32] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE SP*, 2021.

[33] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *ICCAD*, 2017.

[34] N. Kabylkas, T. Thorn, S. Srinath, P. Xekalakis, and J. Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO*, 2021.

[35] R. Kande, A. Crump, G. Persyn, P. Jauernig, A. R. Sadeghi, A. Tyagi, and J. Rajendran. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *USENIX SEC*, 2022.

[36] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *ICCAD*, 2018.

[37] T. Li, H. Zou, D. Luo, and W. Qu. Symbolic simulation enhanced coverage-directed fuzz testing of rtl design. In *ISCAS*, 2021.

[38] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz. Properties first—correct-by-construction rtl design in system-level design flows. In *IEEE TCAD*, 2020.

[39] T. Ormandy. Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html. [Online; accessed 2-September-2023].

[40] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX SEC*, 2020.

[41] H. Ragab, K. Koning, H. Bos, and C. Giuffrida. Bugsbunny: Hopping to rtl targets with a directed hardware-design fuzzer. In *SILM*, 2022.

[42] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[43] K. Ruep and D. Große. Spinalfuzz: Coverage-guided fuzzing for spinalhdl designs. In *IEEE ETS*, 2022.

[44] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan. Grammar-based fuzzing. In *IVMEM*, 2018.

[45] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. https://clang.llvm.org/docs/AddressSanitizer.html. [Online; accessed 4-June-2023].

[46] RISC-V Software. Risc-v unit tests. https://github.com/riscv-software-src/riscv-tests. [Online; accessed 4-June-2023].

[47] RISC-V Software. Spike risc-v isa simulator. https://github.com/riscv-software-src/riscv-isa-sim. [Online; accessed 4-June-2023].

[48] F. Solt, B. Gras, and K. Razavi. Cellift: Leveraging cells for scalable and precise dynamic information flow tracking in rtl. In *USENIX SEC*, 2022.

[49] SonalPinto. Kronos risc-v (with all cascade fixes). https://github.com/cascade-artifacts-designs/cascade-kronos. [Online; accessed 7-December-2023].

[50] SpinalHDL. Vexriscv. https://github.com/SpinalHDL/VexRiscv. [Online; accessed 2-September-2023].

[51] P. Srivastava and M. Payer. Gramatron: Effective grammar-aware fuzzing. In *ISSTA*, 2021.

[52] S. Sutherland. I'm still in love with my x! In *Design and Verification Conference (DVCon)*, 2013.

[53] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. In *IEEE Design & Test of Computers*, 2001.

[54] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, 2009.

[55] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. Fuzzing hardware like software. In *USENIX SEC*, 2022.

[56] M. Turpin and P. V. Engineer. The dangers of living with an x (bugs hidden in your verilog). In *Synopsys Users Group Meeting*, 2003.

[57] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *ICSE*, 2019.

[58] P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu. Sok: The progress, challenges, and perspectives of directed greybox fuzzing. *arXiv:2005.11907*, 2020.

[59] A. Waterman and K. Asanovic. The risc-v instruction set manual. `https://github.com/riscv/riscv-isa-manual`. [Online; accessed 4-June-2023].

[60] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free verilog synthesis suite. In *Austrochip*, 2013.

[61] YosysHQ. Picorv32 - a size-optimized risc-v cpu. `https://github.com/YosysHQ/picorv32`. [Online; accessed 2-September-2023].

[62] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. In *VLSI*, 2019.

A Instruction Categories

The instruction categories used in Cascade are the following: REGFSM (Register lifecycle), FPUFSM (Update the FPU state), ALU (rv32 ALU operations), ALU64 (rv64 ALU operations), MULDIV (rv32 multiplications and divisions), MULDIV64 (rv64 multiplications and divisions), AMO (rv32 atomics), AMO64 (rv64 atomics), JAL (Direct jumps), JALR (Indirect jumps), BRANCH (Branches), MEM (rv32 integer memory operations), MEM64 (rv64 integer memory operations), MEMFPU (rv32 floating memory operations), FPU (rv32 floating-point operations), FPU64 (rv64 floating-point operations), MEMFPUD (rv32 double memory operations), FPUD (rv32 double operations), FPUD64 (rv64 double operations), TVECFSM (Update trap vector), PPFSM (Update previous privileges), EPCFSM (Update trap previous PC), MEDELEG (Update exception delegation), EXCEPTION (Trigger an exception), RDWRCSR (Read/write some CSRs), DWNPRV (Transition privileges downward), FENCES (Fences or `wfi`).

B CPUs under Test

Our testbench consists of 6 RISC-V CPUs of varying complexities and ISA extensions. VexRiscv [50] (`e142e12`, Linux, AHB-L, FPU, rv32imfd) is a highly parametrable CPU written in SpinalHDL, supporting Linux. PicoRV32 [61] (`f00a88c`, Default, rv32im) is a size-optimized CPU written in Verilog (IEEE 1364-2005). Kronos [49] (`13678d4`, Default, rv32i) is a CPU optimized for FPGA applications, written in SystemVerilog (IEEE 1800-2017). CVA6 [62] (`109f9e9`, 4-way 8 kB caches, rv64imafd) is an application-class CPU with Linux support supported by the OpenHW Group organization, and written in SystemVerilog (IEEE 1800-2017). Rocket [3] (`004297b`, BigCore, rv64imafd) is a reference application-class CPU with Linux support, maintained by the Chips Alliance, written in Chisel. BOOM [5] (`004297b`, MediumBoom, rv64imafd) is an out-of-order application-class CPU with Linux support, maintained by the Chips Alliance, written in Chisel.

C Considerations of Previous Fuzzing Claims

In this appendix, we correct some claims from previous work and underline that an ISS-CPU mismatch is not always a bug. In TheHuzz [35], it is considered as a bug (B4) that CVA6 does not throw an exception when executing self-modifying code without `fence.i`. RISC-V does not specify any behavior in this case, hence this is a feature request and not a bug. In HyPFuzz [14], their vulnerability V2 was later denied by the CVA6 maintainers [13], and V3 is not a bug [52]. Since V3 is not a bug, we naturally did not count similar occurrences as bugs in the evaluation of Cascade (see Section 7.6.1). Conclusively, HypFuzz only describes a single new bug. In the archived version of ProcessorFuzz [11], Bug 10 is not a bug but a feature request, because the described behavior is explicitly permitted by the RISC-V specification.

D Enumeration of Discovered Bugs

Table 1 shows the bugs we found in the three designs, the corresponding CWEs and CVEs.

Concurrent findings. After a first bug report campaign, we submitted some new bug reports that were, in the meanwhile, concurrently found by the VexRiscv maintainer. These bugs are V10 and V11. From existing github issues, it cannot be excluded that symptoms of C1 had been noticed in the past. However, the root cause was clearly not understood, until Cascade and its analysis facilities allowed us to propose a fix, that has been approved and merged by the maintainers.

Table 1: Bug Report Table.

Design	Id	Bug Description	CWE	CVE	Sev.
VexRiscv	V1	Non-deterministic conversion from single-precision float to int	681	2023-34885	4.9
	V2	`fmin` with one NaN does not always return the other operand	193	2023-34885	4.9
	V3	Conversion from double to float may pollute the mantissa	681	2023-34885	4.9
	V4	Dependent arithmetic/muldiv FPU operations may yield incorrect results	193	2023-34885	4.9
	V5	Equal registers may be considered distinct by `fle.s` and `feq.s`	697	2023-34885	4.9
	V6	`flt.s` may return 1 when operands are equal	697	2023-34885	4.9
	V7	Under some microarchitectural conditions, square root may be imprecise	1339	2023-34885	4.9
	V8	Single-precision muldiv followed by conversion may pollute the mantissa	681	2023-34891	4.9
	V9	Dependent arithmetic/muldiv operations may cause largely wrong output	682	2023-34891	4.9
	V10	Operations on floating-point registers are authorized when FPU is disabled	1189	2023-34885	4.9
	V11	Wrong access control to the FPU flags leaks information	1189	2023-34892	4.9
	V12	Hang on speculatively executed compressed FPU instructions	1342	2023-34896	7.7
	V13	Inaccurate instruction count when `minstret` is written by software	684	2023-40063	3.6
	V14	Some register comparisons are still incorrect despite a partial fix	697	2023-34885	4.9
PicoRV32	P1	Accessing a non-implemented CSR causes the CPU to hang	1281	2023-34898	4.6
	P2	Spurious exceptions when reading mandatory CSRs	1281	2023-34897	2.6
	P3	Performance counters are not writable	284	2023-34900	2.6
	P4	Performance counters can only be read using some opcodes	284	2023-34914	2.6
	P5	Performance counter addresses are incorrect	684	2023-34913	2.6
	P6	Spurious exception when decoding `fence` instructions	705	2023-34899	5.0
Kronos	K1	RaWaW double-hazard may cause a wrong register value to be forwarded	226	2023-34902	6.6
	K2	Reading existing CSRs causes the CPU to hang in some uarch conditions	1281	2023-34901	7.1
	K3	In some uarch conditions, no exception when writing inexistent CSRs	1281	2023-42310	2.6
	K4	Inaccurate instruction count when `minstret` is written by software	684	2023-40066	3.6
	K5	Incorrect decode logic for `fence` and `fence.i`	684	2023-34903 2023-40064	5.0 5.0
CVA6	C1	Double-precision multiplications yield wrong sign when rounding down	682	2023-34904	4.4
	C2	Single-precision floating-point operations may treat NaNs as zeros	684	2023-34906	5.1
	C3	Division by NAN incorrectly sets NX and NV fflags	684	2023-34905	5.1
	C4	The inexact (NX) flag not set in case of overflow or underflow	684	2023-34907	5.1
	C5	Division of zero by zero incorrectly sets the DZ flag	684	2023-34909	5.1
	C6	Plus and Minus infinity microarchitectural structures are inverted	1221	2023-34910	4.4
	C7	Infinities are not rounded properly and stick to infinity	1339	2023-34911	4.4
	C8	Spurious exceptions when reading some performance counters	684	2023-34911	5.0
	C9	Wrong supervisor performance counter access control	684	2023-42311	5.0
	C10	Under some microarchitectural circumstances, wrong NAN conversion	682	2023-34908	5.1
BOOM	B1	Static rounding is ignored for `fdiv.s` and `fsqrt.s`	1339	2023-34882	6.5
	B2	Inaccurate instruction count when `minstret` is written by software	684	2023-40065	3.6
Yosys	Y1	Logic synthesis of CVA6 inserts a logic bug into the FPU	682	2023-34884	6.8