# HyPFuzz: Formal-Assisted Processor Fuzzing

Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen,
and Aakash Tyagi, *Texas A&M University;* Ahmad-Reza Sadeghi,
*Technische Universität Darmstadt;* Jeyavijayan Rajendran, *Texas A&M University*

## This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

# HyPFuzz: Formal-Assisted Processor Fuzzing

Chen Chen[†], Rahul Kande[†], Nathan Nguyen[†], Flemming Andersen[†], Aakash Tyagi[†],
Ahmad-Reza Sadeghi[*], and Jeyavijayan Rajendran[†]

[†]*Texas A&M University, USA,* [*]*Technische Universität Darmstadt, Germany*

[†]{chenc, rahulkande, nathan.tm.nguyen, flandersen, tyagi, jv.rajendran}@tamu.edu,
[*]{ahmad.sadeghi}@trust.tu-darmstadt.de

## Abstract

Recent research has shown that hardware fuzzers can effectively detect security vulnerabilities in modern processors. However, existing hardware fuzzers do not fuzz well the hard-to-reach design spaces. Consequently, these fuzzers cannot effectively fuzz security-critical control- and data-flow logic in the processors, hence missing security vulnerabilities.

To tackle this challenge, we present *HyPFuzz*, a hybrid fuzzer that leverages formal verification tools to help fuzz the hard-to-reach part of the processors. To increase the effectiveness of *HyPFuzz*, we perform optimizations in time and space. First, we develop a scheduling strategy to prevent under- or over-utilization of the capabilities of formal tools and fuzzers. Second, we develop heuristic strategies to select points in the design space for the formal tool to target.

We evaluate *HyPFuzz* on five widely-used open-source processors. *HyPFuzz* detected all the vulnerabilities detected by the most recent processor fuzzer and found three new vulnerabilities that were missed by previous extensive fuzzing and formal verification. This led to two new common vulnerabilities and exposures (CVE) entries. *HyPFuzz* also achieves $11.68\times$ faster coverage than the most recent processor fuzzer.

## 1 Introduction

Hardware designs are becoming increasingly complex to meet the rising need for custom hardware and increased performance. Around 67% of the application-specific integrated circuit (ASIC) designs developed in 2020 have over 1 million gates, and 45% of them embed two or more processors [2]. However, unlike software vulnerabilities that can be patched, most hardware vulnerabilities cannot be fixed post-fabrication, resulting in security vulnerabilities that put many critical systems at risk and tarnish the reputation of the companies involved. Hence, it is essential to detect vulnerabilities pre-fabrication. However, the emergence of hardware security vulnerabilities [19, 39, 46] shows that vulnerabilities are becoming more stealthy and harder to detect [19].

MITRE reports 111 hardware-related common weakness enumerations (CWEs) as of 2022 [49]. There exist a variety of traditional methodologies and tools for hardware security verification, each having its own advantages and shortcomings, as we explain below.

**Hardware verification techniques.** Academic and industry researchers have developed numerous hardware vulnerability detection techniques. These techniques can be classified as (i) formal: theorem proving [16], formal assertion proving [78], model checking [15], and information-flow tracking [31]; and (ii) simulation-based: random regression [52] and hardware fuzzing [32, 36, 43, 74].

Formal verification techniques prove whether a design-under-test (DUT) satisfies specified properties [42]. However, these techniques alone cannot verify the entire DUT because: (i) in most cases, writing properties requires manual effort and expert knowledge of the DUT, which is error-prone and time-consuming [19, 36], and (ii) large DUTs (such as processors) lead to state explosion, making it impractical to comprehensively verify a DUT for security vulnerabilities [14, 19].

Random regression can automatically generate test cases for verification. However, it does not scale well to large DUTs, including processors [32, 36, 43], especially the regions of the design that are hard-to-reach. For example, the probability of a random regression technique to generate a test case that triggers a zero flag (indicates that the output is "0") in a 64-bit subtract module is $2^{-64}$. Unfortunately, many hardware security-critical components are inherently hard-to-reach (e.g., access control and password checkers) [19].

Inspired by the success of software fuzzing methods, researchers have investigated hardware fuzzing to significantly increase the exploration of design spaces and accelerate the detection of security vulnerabilities [32, 36, 43, 74]. Unfortunately, software fuzzers cannot be directly applied to the software model of hardware due to their fundamental differences [36, 43, 69, 74]; for instance, hardware does not have an equivalent of a software crash, and software does not have floating wires. Hardware fuzzers outperform traditional hardware verification techniques, such as random regression and
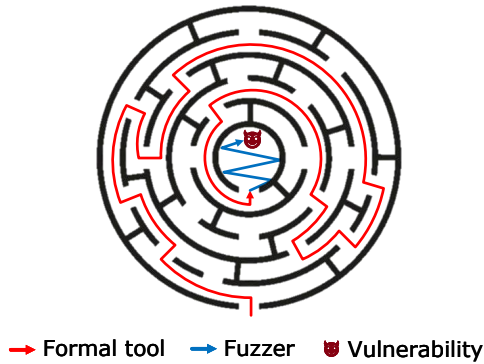
→ Formal tool  → Fuzzer  🐞 Vulnerability

Figure 1: Formal tools and fuzzers catch vulnerabilities in the maze of designs.

formal verification techniques [32, 36, 43, 74], in terms of coverage, scalability, and efficiency in detecting vulnerabilities, and they can fuzz large designs such as processors [32, 36], including `Rocket Core` [11] and `CVA6` [79] [32]. They have found vulnerabilities that lead to privilege escalation and arbitrary code execution attacks [36]. To improve efficiency, these fuzzers use coverage data that succinctly captures different hardware behaviors—finite-state machines (FSMs), branch conditions, statements, multiplexors, etc.—to generate and mutate new test cases.

However, while hardware fuzzing is very promising, it still does not even cover 70% of the hardware design in a practical amount of time. For example, a recent hardware fuzzer, *TheHuzz*, which has higher and faster coverage than random regression techniques and *DIFUZZRTL* [32], has covered only about 63% of the total coverage points in the processors, leaving one-third of the space unexplored for vulnerabilities [36].

The coverage of current hardware fuzzers falls well below industry standards. For instance, Google states that security-critical programs should achieve at least 90% coverage [34]. Achieving 90% coverage is also typical in hardware verification [77]. Faster coverage can promote the decision to tape out and expose unverified design spaces and vulnerabilities early [72]. Unfortunately, none of the existing hardware processor fuzzers meet these criteria.

**Our goals and contributions.** To alleviate the above limitation of hardware fuzzers and inspired by hybrid software fuzzers [70], we aim at making the first step towards building a *hybrid hardware fuzzer* that combines the capabilities of formal verification techniques/tools and fuzzing tools. Figure 1 illustrates the intuition of a **hybrid hardware fuzzer**. The maze represents the entire design space of hardware, and the walls represent the conditions required to reach a design space. Formal tools will lead fuzzers to the hard-to-reach design spaces so that fuzzers can quickly explore them and detect the vulnerability in the hard-to-reach design spaces.

To this end, we developed a new hybrid hardware fuzzer, *HyPFuzz*, which is non-trivial due to the following reasons.

The first challenge is to build an dynamic time scheduling between the formal tool and the fuzzer since static scheduling leads to under- or over-utilization of the capabilities of the formal tool and the fuzzer. The second challenge concerns the selection of coverage points in the DUT to be targeted by the formal tool and the fuzzer, as the former is better at reaching hard-to-reach design spaces while the latter is better at exploring design spaces. The third challenge concerns the incompatibility of formal tools and fuzzing tools: formal tools target assertions about properties of the hardware, whereas fuzzers target coverage points of the hardware. For a seamless integration of formal and fuzzing tools, one needs to convert these assertions into coverage points, and vice-versa.

To solve these challenges: (i) We create a scheduling strategy for fuzzer and formal tool that increases the overall coverage rate of *HyPFuzz* (see Section 4.3). (ii) We propose multiple strategies to select the coverage points for the formal tool and empirically determine the best-performing strategy (see Section 4.4). (iii) We develop a custom *property generator* that converts coverage points to assertions taken by formal tools and a custom *test case converter* that converts Boolean assignments from formal tools into test cases taken by fuzzers, enabling seamless integration of fuzzing and formal techniques for hardware (see Section 4.2).

Consequently, *HyPFuzz* achieves 11.68× faster coverage than the most recently proposed processor fuzzer, *TheHuzz* [36]. It has detected three new vulnerabilities, apart from detecting all the vulnerabilities previously reported. It is also 3.06× faster than *TheHuzz*.

In summary, our main contributions are:

- We present a novel processor fuzzer, *HyPFuzz*, which combines fuzzing and formal verification techniques to verify large-scale processor designs and supports commonly-used hardware description languages (HDLs) like Verilog and SystemVerilog. We use scheduling and selection strategies making *HyPFuzz* fast and efficient for design space exploration and vulnerability detection.

- We evaluate the effectiveness of *HyPFuzz* on five real-world open-source processors from RISC-V instruction set architecture (ISA) [63]—`Rocket Core` [11], `CVA6` [79], and `BOOM` [80]—and OpenRISC ISA [54]—`mor1kx` [53] and `OR1200` [55]— which are widely used as benchmarks in the hardware security community and include all the benchmarks used by *DIFUZZRTL* [32] and *TheHuzz* [36].

- *HyPFuzz* achieves 11.68× faster coverage than the most recent processor fuzzer and 239.93× faster coverage than random regression. It found three new vulnerabilities leading to two common vulnerabilities and exposures (CVE) entries, `CVE-2022-33021` and `CVE-2022-33023`, apart from detecting all the vulnerabilities detected by *TheHuzz*. *HyPFuzz* is 3.06× faster regarding run-time and 3.05× faster regarding the number of instructions than *TheHuzz*.

## 2 Background

We now provide a succinct background on formal verification and hardware fuzzing, which form the basis of *HyPFuzz*.

### 2.1 Formal Verification

Formal verification techniques have shown to be effective at finding subtle vulnerabilities [15], such as side-channel leakage [23, 48, 60, 73], information leakage [20, 22, 62], and concurrency errors [17, 27]. These techniques can find these vulnerabilities because they can exhaustively prove whether a design-under-test (DUT) satisfies specified properties.

Existing hybrid software fuzzers use symbolic execution to generate test cases and explore the hard-to-reach regions of a DUT [25, 26, 57, 58, 70, 81, 82]. This is done by identifying the execution paths a fuzzer cannot reach and generating test cases that force the target DUT to execute these paths [25]. Symbolic execution explores all possible execution paths of a DUT, but it requires a mapping between other coverage metrics to execution paths and is limited by the huge space of execution path of large designs [82].

*HyPFuzz* uses another method as it uses commercial hardware formal tools, like Cadence *JasperGold* [4], to generate such test cases. *JasperGold* requires SystemVerilog Assertion (SVA) properties [28] known as *cover* properties as inputs to enable *HyPFuzz* to generate test cases. The properties proved by such tools mostly fall under two categories: (i) "assert/safety" properties, where one must verify all possible execution paths of the DUT to ensure that the property is not violated; and (ii) "cover/progress/trace-existing" properties, where one must verify there exists a path from the initial state of the DUT to a state where the property is satisfied. The relationship between a *cover* property and an *assert* property can be shown as $cover(p) = assert(\neg p)$, where $p$ represents the expression of a property. Hence, compared to symbolic execution, formal tools such as *JasperGold* that support *cover* property can (i) explore hard-to-reach regions based on various coverage metrics rather than execution paths and (ii) efficiently verify larger DUTs since the tools only need to find the existence of one path that satisfies the property.

On proving a *cover* property, formal tools will return one of three results: (i) *unreachable*, (ii) *reachable*, or (iii) *undetermined* [61]. A property is *unreachable* when there is no execution path from an initial state of the DUT to a state satisfying the property. A *reachable* property will have at least one such path. Formal tools usually output such a path consisting of Boolean assignments for the inputs of the DUT for each clock cycle to satisfy the property. *HyPFuzz* uses such assignments to generate the test cases as seeds for the fuzzer (see Section 4.2). A property is *undetermined* if the formal tool cannot find a path within a given time limit. We account for all these properties while building *HyPFuzz*.

Several commercial formal tools verify hardware DUTs, such as Siemens *Questa* [6], Synopsys *VC Formal* [8], and Cadence *JasperGold*. They operate on hardware designs represented in different hardware description languages (HDLs). For *HyPFuzz*, we currently use the *JasperGold*, which is well-known for its performance and features supported [61] and is also used for the verification of RISC-V processors [1].

Despite their performance, these tools cannot formally verify complete processor designs because the size of the designs and/or complexity are often too big. For instance, *JasperGold* took around eight days to verify 94.51% of the branch coverage points in CVA6 processor [79] (see Section 3.3), motivating the need for techniques such as hardware fuzzing.

*JasperGold* includes SAT/BDD-based formal engines with variations of these algorithms to prove properties [3]. Therefore, *HyPFuzz* could use any other SAT/BDD-based proof engines that support the generation of Boolean assignments for SVA properties to generate test cases.

### 2.2 Hardware Fuzzing

Most hardware fuzzers consist of a *seed corpus*, *mutator*, and *vulnerability detector* [32, 36, 74]. The *seed corpus* is an initial set of input test cases called **seeds** [32]. These input **test cases** are the inputs required to simulate the DUT. The *seed corpus* is either manually crafted or generated randomly [51]. The fuzzer simulates the DUT with these test cases, collects **coverage**, and **mutates** all "interesting" test cases (i.e., test cases that achieve coverage) using its *mutator* to generate new test cases [43]. The *vulnerability detector* reports any vulnerabilities detected during the simulation. The fuzzer simulates these new test cases and repeats the cycle until it achieves the desired coverage. Next, we explain the various components and tasks performed by hardware fuzzers.

**DUT** is a hardware design written in HDLs like Verilog and SystemVerilog [36, 51, 74] or hardware construction languages (HCLs) like Chisel [13, 32, 43]. Hardware fuzzers use simulation tools like *Verilator* [69], Synopsys *VCS* [71], and Siemens *Modelsim* [5] to simulate these DUTs.

**Test cases** of generic hardware fuzzers include data for each input signal of the DUT for each clock cycle [43, 51, 74]. In contrast, fuzzers designed specifically to fuzz processors generate binary executable files as test cases [32, 36].

**Coverage** measures the number of various types of hardware behaviors, such as toggling the select signals of muxes (mux-toggle coverage [43]) and setting registers that drive the selected signals of muxes to different values (control-register coverage [32]) during the simulation. **Coverage points** are assigned to each of these behaviors. For example, branch coverage indicates whether the different paths of a branch statement are covered or not. Whenever a design enters one of the branch paths, its corresponding coverage point is considered **covered**; otherwise, it remains **uncovered**.

The DUT is instrumented to generate coverage during the simulations [43]. Thus, covering all the coverage points in

the DUT is essential to verify all the hardware behaviors. Hardware fuzzers use coverage as **feedback** to determine the interesting test cases [32, 36, 43, 74].

**Mutations** are data manipulation operations, such as bit-flip, byte-flip, clone, and swap, inspired by software fuzzers like the AFL fuzzer [44] [36, 43].

**Vulnerability detection** in hardware fuzzers involves either differential testing or assertion checking. In differential testing, the outputs of the DUT and a golden reference model (GRM), when tested with the same test case, are compared to detect vulnerabilities [32, 36]. In assertion checking, we insert the conditions to trigger the vulnerabilities or assertion properties into the DUT based on its specification and use the violations of these assertions during the simulation to detect vulnerabilities [51, 74]. Note that, unlike software, hardware does not have events like crashes, memory leaks, and buffer overflows to use for vulnerability detection [74].

## 3 Motivation

In this section, we highlight the limitations of existing formal and fuzzing techniques to motivate the need for hybrid hardware fuzzers. To this end, we use a popular, open-sourced RISC-V [63] based processor, `CVA6` [79], as a case study. However, we perform extensive evaluation of *HyPFuzz* on all modules of five different processors (see Section 5).

### 3.1 Case Study on **CVA6** Processor

Consider the Listing 1, which shows the trigger condition of three interrupts in the interrupt handler of `CVA6`. Verifying the correctness of the interrupt handler is critical, as a vulnerable interrupt handler can be exploited for information leakage [18]. To trigger each type of interrupt, the corresponding bits of two control and status registers (CSRs): `mie` and `mip`, need to be enabled (i.e., set to `1'b1`). Thus, the test case should simultaneously consists of instructions that set the bits of both `mie` and `mip` registers. This condition is covered by the *branch coverage* metric, which checks if both directions of the branch (in this case, the *if* statement) are taken [50].

### 3.2 Limitations of Existing Hardware Fuzzers

Hardware fuzzers iteratively perform seed generation and mutation to improve coverage [13, 32, 36, 43, 74]. However, fuzzers still require an exponentially large amount of time to cover some coverage points—whose test cases are hard to generate due to the specific conditions required to trigger them—leaving multiple design spaces unexplored and vulnerabilities undetected.

**Case Study on **CVA6**'s interrupt controller.** We fuzzed the `CVA6` processor with the most recent processor fuzzer, *The-*

Listing 1: Interrupt handler in the `CVA6` processor [79].

```
1   localparam SupervisorIrq = 1;
2   localparam logic [63:0] S_EXT_INTERRUPT = 64'd9;
3   ...
4   // Interrupt handler
5   if (mie[S_TIMER_INTERRUPT] && mip[S_TIMER_INTERRUPT])
6       interrupt_cause = S_TIMER_INTERRUPT; // Supervisor Timer
7   if (mie[S_SW_INTERRUPT] && mip[S_SW_INTERRUPT])
8       interrupt_cause = S_SW_INTERRUPT; // Supervisor Software
9   if (mie[S_EXT_INTERRUPT] && (mip[S_EXT_INTERRUPT] | irq[
        ↪ SupervisorIrq]))
10      interrupt_cause = S_EXT_INTERRUPT; // Supervisor External
```

*Huzz*, for 72 hours, generating more than 200*K* test cases[*]. Unfortunately, *TheHuzz* did not cover any of the branch coverage points of all the three interrupts [36]. We performed further analysis to understand this limitation.

Consider the `S_EXT` interrupt in Line 9 in Listing 1. Triggering this interrupt requires the `S_EXT_INTERRUPT` bit of both the CSRs, `mie` and `mip`, to be enabled. According to the RISC-V instruction set architecture (ISA) emulator, *Spike* [64], there are only four instructions (`CSRRW`, `CSRRWI`, `CSRRS`, and `CSRRSI`) out of the total 1146 RISC-V instructions can modify the values of CSRs, and there are 229 CSRs in total. The probability of generating an instruction that sets the `mie` register is $4/(1146 \times 229) = 1.524 \times 10^{-5}$. As we also need the bit of `mip` CSR to be set, the combined probability of generating such a test case is only $(1.524 \times 10^{-5})^2 = 2.323 \times 10^{-10}$. Though this is the probability of randomly generating a test case, it sheds light on why existing hardware fuzzers [13, 32, 36]—which use the coverage feedback and mutation techniques—could not cover this coverage point after fuzzing for 72 hours. This demonstrates that *existing hardware fuzzers are insufficient to explore hard-to-reach design spaces, leaving vulnerabilities undetected.*

### 3.3 Limitations of Formal Verification

Theoretically, formal tools can use *cover* properties to prove the reachability of all the coverage points in a DUT [28], achieving 100% of the reachable coverage. However, this requires writing and proving *cover* properties for all the coverage points, an error-prone and time-consuming task (as it requires design knowledge and manual effort), especially in large and complex hardware designs like processors with thousands of coverage points [19]. Moreover, since the reachability of a *cover* property only requires the existence of one path, formal tools, like *JasperGold*, do not guarantee to find all the paths with vulnerabilities.

**Case Study on **CVA6**'s interrupt controller.** The `CVA6` processor has $9.53 \times 10^3$ branch coverage points. First, there exists no tool that can convert these branch coverage points into *cover* properties that tools like *JasperGold* [4] can prove. Thus, one has to craft these properties manually. We developed and used a *property generator* that can automatically derive *cover* properties from branch coverage points.

---

[*]For this paper, a *test case* refers to a binary executable.

Listing 2: The Verilog code of CSR reading in `CVA6`.

```
1  if (csr_read) begin
2    unique case (csr_address)
3      // Counters and Timers
4      ML1_ICACHE_MISS, ML1_DCACHE_MISS, MITLB_MISS,
5      MDTLB_MISS, MLOAD, MSTORE, MEXCEPTION,
6      MEXCEPTION_RET, MBRANCH_JUMP, MCALL, MRET,
7      MMIS_PREDICT, MSB_FULL, MIF_EMPTY,
8      MHPM_COUNTER_17,
9      ...
10     MHPM_COUNTER_31: csr_rdata = perf_data_i; => Point
```

Second, but more importantly, we evaluate the time taken by *JasperGold* to prove all these properties and the coverage it achieves. Since the corresponding Boolean assignments generated by *JasperGold* cannot be directly used as test cases for fuzzing a processor, we developed a *test case converter* that can automatically convert these Boolean assignments into the binary executable format. We then simulate these test cases and collect the branch coverage achieved. The time consumption of each point is the summation of formal verification and simulation. To prevent *JasperGold* from spending too much time on the property of one point, we limit the maximal time on each property (see Section 5.1). *JasperGold* took eight days to verify 94.51% of the branch points in the `CVA6` processor as shown in Figure 2. This shows that *using formal tools requires extensive manual labor and has tremendous runtime in verifying all coverage points in the DUT.*

**Case Study on vulnerability detection.** Theoretically, formal tools alone can achieve 100% coverage by proving the *cover* properties of all the coverage points. But, there is still scope for vulnerabilities. For example, consider the vulnerability **V3** found by *HyPFuzz* where the `CVA6` processor returns X-values when accessing unallocated CSRs. Listing 2 is a code segment from the `CVA6` processor that accesses data from the hardware performance counters (HPCs) using the address of the CSRs (`csr_address`). The HPCs are used for anomaly and malicious behavior detection [41, 76], hence verifying their security is essential. Triggering the vulnerability **V3** requires a test case to access the data of the counters among `MHPM_COUNTER_17` to `MHPM_COUNTER_31`. However, *JasperGold* will not always find a path to access the target counters. This is because all counters share the same point to reduce the instrumentation overhead from the branch coverage metric. A *cover* property does not explore all paths under the point, which shows that *using formal tools with standard cover properties is insufficient to detect all vulnerabilities.*

### 3.4 Advantages of a Hybrid Fuzzer

By using both techniques in tandem, hybrid fuzzers overcome the limitations of using fuzzing and formal techniques alone. The fuzzer quickly explores the DUT through the mutation of effective test cases. The formal tool verifies points that the fuzzer is struggling to cover and provides the corresponding test cases as seeds to the fuzzer. The fuzzer then mutates these test cases to explore the target design further.

Listing 3: Instructions covering the point of the `S_EXT_INTERRUPT`.

```
1    ORI X6, X3, h'204; // update reg X6 with s_ext value
2    CSRRS X0, mie, X6; // write the value of X6 to mie
3    CSRRS X0, mip, X6; // write the value of X6 to mip
```

**Case Study on `CVA6`'s interrupt controller.** Since the fuzzer can cover none of the three coverage points in the interrupt handler (see Section 3.2), the hybrid fuzzer uses a formal tool for assistance. Of these three branch coverage points, consider the point of `S_EXT_INTERRUPT` at line 9 in Listing 1 as an example. We first use the conditions for covering this point (see Appendix B) into an SVA *cover* property: cover property (*mie*[*S_EXT_INTERRUPT*] && *mip*[*S_EXT_INTERRUPT*] || *irq*[*SupervisorIrq*]). *JasperGold* then takes only 20 seconds to find a path to this property and dump Boolean assignments as shown in Figure 9.

However, the fuzzer cannot directly use these Boolean assignments because the processors require binary executables as test cases. Hence, we identify instruction-related signals (e.g., the input instruction port of the decoder) from the Boolean assignments and parse their values beginning from the initial state to the state that satisfies the property. Listing 3 shows the extracted sequence of instructions.

Then, the sequence of instructions is converted into a valid executable file. Such files consist of three instruction sequences: `INIT` instructions that initialize registers and memory of the processor, `TEST` instructions that contain the testing instruction sequence, and `EXIT` instructions that handle normal and abnormal (e.g., exception) termination of simulation, as shown in Figure 10. To generate a valid executable file, we create an executable file template with `NOP` instructions as `TEST` instructions, compare and identify the initial memory address of the `TEST` instruction section from the disassembly file, and replaces these `NOP` instructions with the instruction sequence extracted from the Boolean assignments.

The hybrid fuzzer can now use this test case as a seed, mutate it, and generate new test cases that cover the remaining two coverage points in the interrupt handler. For example, a commonly-used mutation technique in existing hardware fuzzers [13, 32, 36], *random-8*, overwrites a random byte with a random value in the instruction [36, 43]. This mutation can easily toggle the other bits in `mie` and `mip` CSRs, covering the other coverage points of the interrupt handler. Therefore, a hybrid fuzzer can *use test cases from formal tools to overcome the limitation of fuzzers, thereby increasing the coverage.*

We built a *property generator* to automatically generate the *cover* properties of coverage points and a *test case converter* (see Section 4.5) to automatically convert Boolean assignments of reachable properties into test cases.

**Case Study on `CVA6`.** Figure 2 shows the branch coverage achieved by the most recent processor fuzzer, *TheHuzz*; the formal tool, *JasperGold*; and our hybrid fuzzer, *HyPFuzz*. *JasperGold* continues to achieve coverage but is slow due to its high run-time to cover each coverage point; it reaches
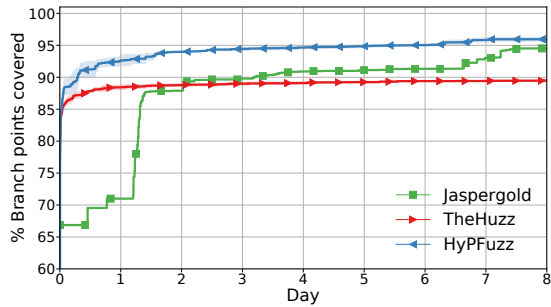
Figure 2: Eight day coverage results of *TheHuzz* [36], *Jasper-Gold* [4], and *HyPFuzz* for the `CVA6` processor [79].



Figure 3: Framework of *HyPFuzz*. The circle and cross represent covered and uncovered points, respectively.

a coverage of 94.51% after running for eight days. On the other hand, even though *TheHuzz* initially achieves faster coverage than *JasperGold*, it fails to achieve the coverage beyond 88%, even after running for 72 hours, as all the remaining coverage points are hard-to-reach. In contrast, *HyPFuzz* achieves 94.78% coverage (6.1% more coverage compared to *TheHuzz*) in 72 hours, and it achieves the 94.51% coverage achieved by *JasperGold* in 50.71 hours (3.71× faster than *JasperGold*). Therefore, a hybrid fuzzer can *use the fuzzer to explore the DUT and overcome the manual and run-time overhead limitations of formal tools, achieving coverage faster.*

## 4 Hybrid Hardware Fuzzing

In this section, we first elaborate on the challenges of building a hybrid fuzzer and how we address them in *HyPFuzz*.

### 4.1 Challenges

**C1. Scheduling:** The speed of the fuzzer varies over time depending on the design-under-test (DUT) and the type of fuzzer used. Similarly, the speed of the formal tool varies from one coverage point to another based on the DUT, the type of formal tool, and the computational resources. Thus, challenge **C1** is to *build a dynamic scheduler between the formal tool and the fuzzer that minimizes the under- and over-utilization of the capabilities of the formal tool and the fuzzer.*

**C2. Selection of coverage points:** Since the coverage point targeted by the formal tool determines the seed of the fuzzer, it also impacts the successive points covered by the fuzzer as it mutates this seed. Thus, the hybrid fuzzer should select the uncovered points that maximize the number of coverage points the fuzzer can uncover, thereby increasing the speed of the fuzzer. However, the current set of uncovered points depends on the DUT and also what points have been covered by the fuzzer in the past. Thus, challenge **C2** is to *build a point selector that maximizes the rate of coverage despite the uneven distribution of uncovered points in the DUT, which also changes with time.*
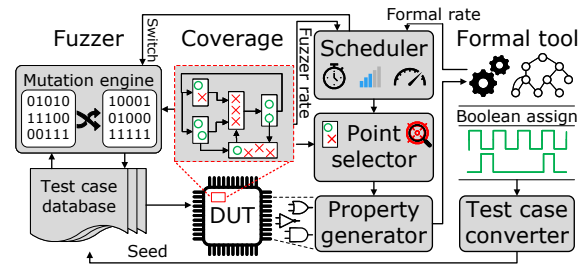
**C3. Seamless integration.** A hybrid fuzzer should seamlessly integrate the formal tool and the fuzzer to be faster and easier to use. However, the inputs and outputs of the fuzzer and formal tool are incompatible. The fuzzer uses test cases as input, while the formal tool generates Boolean assignments for the inputs of the DUT for each clock cycle as the output. Also, the formal tool needs *cover* properties as input while the fuzzer outputs coverage of each point in the DUT. Hence, it is not straightforward to combine a formal tool and a fuzzer. Hence, challenge **C3** is to *seamlessly integrate formal and fuzzing tools to build an automated flow for the hybrid fuzzer.*

We address challenges **C1** and **C2** by building a dynamic *scheduler* and an *uncovered point selector*, respectively. We solve **C3** by building a *property generator* and a *test case converter*, which facilitate seamless integration of the fuzzer with the formal tool.

### 4.2 Framework of *HyPFuzz*

*HyPFuzz* consists of the *scheduler*, *point selector*, *property generator*, and *test case converter*, apart from the fuzzer and the formal tool, as shown in Figure 3. *HyPFuzz* starts by invoking the *point selector* that heuristically selects the uncovered point that the formal tool should verify (see Section 4.4). Then, the *property generator* generates the *cover* property for that point (see Section 4.5). The formal tool proves this property and generates the Boolean assignments for each input of the DUT for every clock cycle required to trigger that uncovered point. The *test case converter* converts the Boolean assignments of instruction signals into a test case, which the fuzzer uses as a seed (see Section 4.5). The fuzzer simulates the DUT with this seed to reach the coverage point. The fuzzer also mutates this seed to cover the neighborhood points. It runs until the *scheduler* stops it. Then, the *point selector* selects the next uncovered point. *HyPFuzz* repeats this process until it achieves the target coverage or hits a timeout.

### 4.3 Scheduling of Fuzzer and Formal Tool

Our *scheduler* uses a dynamic scheduling strategy that switches *HyPFuzz* from fuzzer to formal tool when the rate

of coverage increment of the fuzzer ($r_{fuzz}$) is less than that of the formal tool ($r_{fml}$). The rates reflect their capability to explore the design spaces. On the other hand, the formal tool is running until it generates the Boolean assignments to reach the uncovered point. Next, we formulate the rate of coverage increment of the fuzzer and formal tool, which helps one to schedule them dynamically.

**Formal tool's coverage increment rate ($r_{fml}$)** is the ratio of the number of coverage points verified by the formal tool so far. Since a formal tool will process the points that are hard to be covered by the fuzzer, we can calculate the optimal $r_{fml}$ using $\frac{n}{t_p}$, where $n$ represents the uncovered points selected when switching to the formal tool, and $t_p$ represents the time spent by the formal tool on proving the corresponding properties. However, since we need to calculate $r_{fml}$ before switching to the formal tool, *HyPFuzz* needs to predict the uncovered points selected and the time taken by the formal tool on the corresponding properties. Unfortunately, both predictions are difficult due to the randomness of fuzzing and the low prediction accuracy of the time cost of a formal tool (the accuracy of the state-of-the-art machine learning strategy is 68%, given a property [21]). Therefore, we first calculate the average time $t_{ave}$ spent by a formal tool on properties. We then use $t_{ave}$ to estimate the $r_{fml}$ as $\frac{1}{t_{ave}}$ to reflect how the formal tool will explore the hard-to-reach spaces of the fuzzer in a design.

However, it is infeasible to calculate $t_{ave}$ by proving all uncovered points in the hard-to-reach region. Therefore, we estimate $r_{fml}$ using the moving average, which is widely applied to estimate the underlying trend [33], as

$$ r_{fml} = \frac{|C|}{\sum_{c \in C} t_{fml}(c)}, \tag{1} $$

where $C$ is the set of coverage points verified by the formal tool, and $t_{fml}(c)$ denotes the run-time of the formal tool to generate a test case for the coverage point $c$.

**Fuzzer's coverage increment rate ($r_{fuzz}$)** is the ratio of the number of new coverage points covered by the fuzzer in a rolling window over the run-time of the fuzzer. To prevent under-/over-utilization of a fuzzer, $r_{fuzz}$ reflects how well the fuzzer recently explored the design spaces. Fuzzers usually achieves faster coverage initially and then slow down due to the hard-to-reach spaces in the design [32, 36, 43]. Therefore, if we calculate $r_{fuzz}$ including coverage increment at the beginning, $r_{fuzz}$ will become unnecessarily high and cause over-utilization of the fuzzer, delaying the switching process. However, if we calculate $r_{fuzz}$ using the coverage achieved by the most recent test case, the test case may not achieve new coverage, whereas the upcoming test cases can due to the randomness of the fuzzing process. This causes under-utilization of the fuzzer and hence cannot fuzz around the seed from the formal tool entirely. Therefore, a rolling window is used in this case to compute the instantaneous rate of the fuzzer. Let $K$ denote the set of all the test cases generated by the fuzzer, where

$k_i$ denotes the $i^{th}$ test case generated. Then, the set of test cases in the window $w$ is $K_w = \{k_i \in K \mid |K| - w < i \le |K|\}$. Let $n(k_i)$ and $t_{fuzz}(k_i)$ denote the number of new coverage points covered and the run-time of the fuzzer for each test case, respectively. Thus,

$$ r_{fuzz}(w) = \frac{\sum_{k_i \in K_w} n(k_i)}{\sum_{k_i \in K_w} t_{fuzz}(k_i)} \tag{2} $$

The *scheduler* computes $r_{fuzz}$ and $r_{fml}$ in real-time (see Section 5.1), runs the fuzzer as long as it can cover more points than the formal tool, and then switches to the formal tool when $r_{fuzz} < r_{fml}$.

## 4.4 Selection of Uncovered Points

The hardware design and verification process is modular [12, 13]. Therefore, to be compatible with the existing hardware design verification flow, we develop strategies for the selection of uncovered points at the module level.

We run existing processor fuzzers, including *DIFUZ-ZRTL* [32] and *TheHuzz*, on up to five different processors and analyze their effectiveness at the module level. Based on this study, we make the following observations:

**Observation O1**. The farther the module is from the inputs of the DUT, the harder it is for the fuzzer to trigger the module's components accurately. Thus, the coverage of the fuzzer isproportional to the *distance* of the module from the input.

**Observation O2**. The more the number of uncovered points in the module, the more coverage points the fuzzer can cover if the fuzzer's seed activates the module.

**Observation O3**. The more DUT logic a module drives, the higher the probability of the fuzzer covering new points.

Based on these observations, we have developed three strategies: (i) *BotTop*, (ii) *MaxUncovd*, and (iii) *ModDep*. The strategies include deterministic and non-deterministic operations, which will first select a module and then randomly select an uncovered point inside. We also use *RandSel*, a naive selection strategy that randomly selects the uncovered points. We evaluate these strategies on a comprehensive set of five real-world, open-source processors covering one of the first and most widely used OpenRISC ISA [54] and RISC-V ISA [63], respectively, and select the best strategy for our *point selector* based on empirical results (see Section 5.3). We now explain these three strategies using CVA6 as an example.

### 4.4.1 *BotTop* Strategy

To ensure the efficient usage of the formal tool and fuzzer, the formal tool should target the hard-to-reach coverage points, and the fuzzer should target the remaining points. Based on Observation **O1**, *BotTop* selects modules *deep* in the DUT, i.e., their distance from the DUT's input. We define this distance as
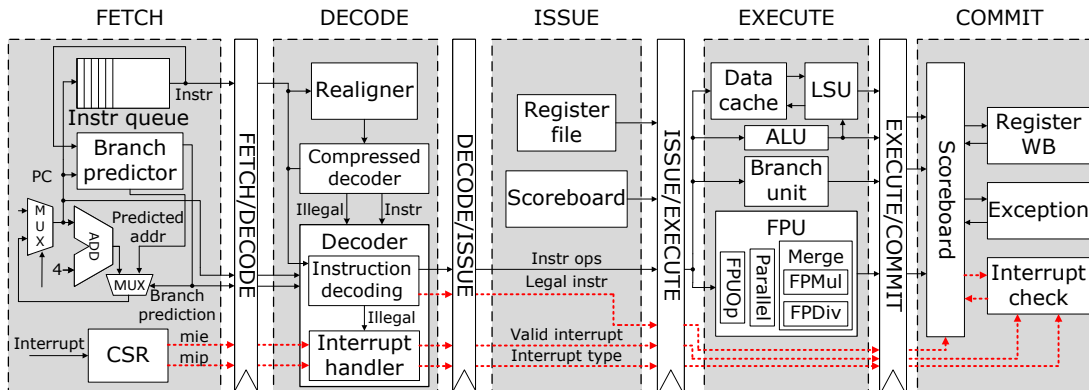
Figure 4: Simplified pipeline of the CVA6 processor [79]. The connectivity in dotted lines is an example to show how a module drives various other modules in the design.

the number of modules between the input of the DUT and the target module. For example, consider the simplified pipeline of the CVA6 processor shown in Figure 4. In this pipeline, the distance of FETCH, DECODE, and EXECUTE is one, and both FPMul and FPDiv have a distance of four. Thus, the *BotTop* strategy assigns the highest priority to FPMul and FPDiv over other modules in the CVA6 processor.

### 4.4.2 *MaxUncovd* Strategy

In a hybrid fuzzer, the seed generated using the formal tool will cover a coverage point in the hard-to-reach design space. On mutating the seed, the fuzzer will explore the design space in the "vicinity" of this covered point. Thus, based on Observation **O2**, having more uncovered points in the "vicinity" will increase the number of coverage points the fuzzer can cover, thereby accelerating *HyPFuzz*. Thus, our second strategy, *MaxUncovd* prioritizes the module with the maximum number of uncovered points over the rest of DUT's modules, irrespective of its distance from the inputs. Unlike the *BotTop* strategy, whose module distance is fixed for a given DUT, the number of uncovered points in a given module decreases as *HyPFuzz* explores more design space over time. Hence, *MaxUncovd* is a dynamic strategy that recomputes the priorities of each module every time the *point selector* is invoked.

For example, in the CVA6 processor, the Decoder and floating point unit (FPU) modules have 381 and 108 branch coverage points, respectively. In the beginning, all the coverage points in the DUT are uncovered. Hence, the Decoder will have more uncovered points; consequently, *MaxUncovd* prioritizes the Decoder over the FPU. However, over time, the number of uncovered points in the Decoder decreases as instructions of all types activate this module.

### 4.4.3 *ModDep* Strategy

Triggering a coverage point first requires activating the modules that drive the logic corresponding to this coverage point.

Thus, based on Observation **O3**, targeting the coverage points from a module with many other modules in their fanout COI will allow the fuzzer to uncover more points. Hence, our *ModDep* strategy prioritizes modules with higher COI over other modules, as shown in Algorithm 2. For example, *ModDep* assigns the highest priority to the CSR module in the CVA6 processor because it has the highest fanout due to driving multiple components: interrupt control logic (shown using red dotted lines in Figure 4), write logic in Register file, and enable logic in Data cache.

## 4.5 Integrating Fuzzer and Formal Tool

Formal tools target the DUT's properties, but fuzzers target the DUT's coverage points. Due to this incompatibility, we cannot directly send the target coverage point to the formal tool to verify or take the Boolean assignments of a proved property as an executable test case for the fuzzer, creating challenge **C3**. To seamlessly integrate the fuzzer and formal tool, we develop a *property generator* and a *test case converter*.

*Property generator* generates the *cover* property for the selected uncovered point. It parses the DUT's logic and identifies the conditions for covering the point. Then, it converts the conditions into a *cover* property and loads it to the formal tool along with the DUT. For example, given an uncovered point of branch coverage, the *property generator* analyzes the dependencies of the branch statement. It then identifies the conditions to cover that point, and the logical conjunction of them will form the expression of the corresponding *cover* property. The *property generator* is compatible to uncovered points of other coverage metrics, as shown in Appendix B.

*Test case converter* As mentioned earlier, the formal tool generates the Boolean assignments of the input signals for each clock cycle to cover a target coverage point. However, the fuzzer has a different input format. For example, processor fuzzers, such as *TheHuzz* and *DIFUZZRTL* [32], use sequences of instructions as test cases to fuzz [32, 36, 43].

**Algorithm 1:** *HyPFuzz*

**Input:** *DUT*: design-under-test;
   *s*: point selection strategy;
   $t_{limit}$: time limit of fuzzing;
   *tc*: target coverage;
**Output:** *n*: total coverage achieved;

1   $t \leftarrow 0$, $r_{fml} \leftarrow 0$
2   Initialize registers and memory in DUT to zero
3   **while** $(n < tc)$ *and* $(t < t_{limit})$ **do**
4      $r_{fml}, seed, t \leftarrow$ SWITCHTOFORMALTOOL
      $(DUT, s, t, r_{fml})$
5      $n, t \leftarrow$ SWITCHTOFUZZER
      $(DUT, t, r_{fml}, seed, t_{limit})$

6   **return** *n*
7   **Function** SWITCHTOFORMALTOOL(*DUT*, *s*, *t*, $r_{fml}$)**:**
    /* select an uncovered point *p* based on
     RandSel, MaxUncovd, BotTop, and
     ModDep strategies      */
8      $p \leftarrow$ SELECTIONSTRATEGY(*DUT*, *s*)
9      $cp_{rop} \leftarrow$ PROPERTYGENERATOR(*DUT*, *p*)
10     $Boolean\_assignment, r_{fml} \leftarrow$
      FORMALTOOL(*DUT*, $cp_{rop}$)
11     $seed \leftarrow$ TESTCASECON-
      VERTER(*DUT*, *Boolean_assignment*)
12     **return** $r_{fml}, seed, t$

13   **Function** SWITCHTOFUZZER(*DUT*, *t*, $r_{fml}$, *seed*,
   $t_{limit}$)**:**
14      **repeat**
15        $n, t, r_{fuzz} \leftarrow$ FUZZER(*DUT*, *seed*, *t*)
16      **until** $(r_{fuzz} < r_{fml})$ *or* $(t \geq t_{limit})$
17      **return** $n, t$

---

The *test case converter* will use the ISA of the DUT to map the Boolean assignments generated by the formal tool to a sequence of instructions. This mapping process uses the ISA of the target processor and is repeated for each clock cycle. The *test case converter* also prepends this test case with another sequences of instructions that initialize and terminate the processor's simulation as the seed.

### 4.6 Putting It All Together

As shown in Algorithm 1, *HyPFuzz* first marks all the coverage points as uncovered. Then, we run a test case where we initialize all the registers and memory values of the DUT to zero using `nop` instructions (Line 2). We then select an uncovered point (*p*) based on the selected strategy (*s*) (Line 8). We then convert this *p* into a corresponding *cover* property and invoke the formal tool to target this property (Line 9). The formal tool returns the Boolean assignment for each signal of the DUT for each clock cycle (Line 10), which is

Table 1: Benchmarks used in our study.

| Processor | # Latches | # Gates | # Branch points | OoO | SIMD | Time limit (sec) |
|---|---|---|---|---|---|---|
| OR1200 [55] | $3.08 \times 10^3$ | $2.48 \times 10^4$ | $7.00 \times 10^2$ | ✗ | ✗ | $4.80 \times 10^2$ |
| mor1kx [53] | $5.29 \times 10^3$ | $4.64 \times 10^4$ | $1.34 \times 10^3$ | ✗ | ✗ | $6.98 \times 10^2$ |
| CVA6 [79] | $2.48 \times 10^4$ | $4.63 \times 10^5$ | $9.53 \times 10^3$ | ✓ | ✗ | $3.80 \times 10^3$ |
| Rocket Core [11] | $1.64 \times 10^5$ | $9.24 \times 10^5$ | $1.33 \times 10^4$ | ✗ | ✗ | $2.71 \times 10^3$ |
| BOOM [80] | $1.99 \times 10^5$ | $1.26 \times 10^6$ | $2.42 \times 10^4$ | ✓ | ✗ | $3.83 \times 10^3$ |

then converted to a sequence of instructions to be used as the seed of the fuzzer (Line 11). We then invoke the fuzzer on the DUT (Line 15). The fuzzer is continued to execute until the coverage rate of the fuzzer is less than that of the formal tool (Line 16); otherwise, we select a new uncovered point for the formal tool to target. This cycle continues until the target coverage is achieved or the time limit is reached.

## 5 Evaluation

We first evaluate how the rate of formal tool ($r_{fml}$) and fuzzer ($r_{fuzz}$) reflect the capability that the formal tool and the fuzzer explore the design spaces. We then evaluate *HyP-Fuzz* on five open-source processors from RISC-V [63] and OpenRISC [54] instruction set architectures (ISAs) to empirically select the best point selection strategy. We then compare *HyPFuzz* with the most recent hardware processor fuzzer [36] regarding the coverage achieved and the vulnerabilities detected. Finally, we investigate the capability of *HyPFuzz* to cover points of different coverage metrics. We ran our experiments on a 32-core, 2.6 GHz Intel Xeon processor with 512 GB of RAM running Cent OS Linux release 7.9.2009.

### 5.1 Evaluation Setup

**Benchmark selection.** Most commercial processors are protected intellectual property (IP) and close-sourced. Thus, we pick the three large (in terms of the number of gates) and widely-used open-sourced processors: `Rocket Core` [11], `BOOM` [80], and `CVA6` [79] from the RISC-V ISA and `OR1200` [55] and `mor1kx` [53] from the OpenRISC ISA as the diverse set of benchmarks to evaluate *HyPFuzz*. Table 1 lists the details of these processors. `CVA6` and `BOOM` processors are complex than the `Rocket Core`, `mor1kx`, and `OR1200`, with advanced micro-architectural features like out-of-order execution (OoO) and single instruction-multiple data (SIMD).
**Evaluation environment.** We use the popular and industry-standard Cadence *JasperGold* [4] and Synopsys *VCS* [9] tools as the formal tool and the simulation tool, respectively. We use the branch coverage generated by *VCS* to evaluate *HyPFuzz* because branch coverage is an important coverage metric in vulnerability detection [50]. Appendix B details how we can evaluate *HyPFuzz* with other coverage metrics. We use *Chip-yard* [10] as a simulation environment for the RISC-V processors. For *HyPFuzz*'s fuzzer, we use the most recent hardware fuzzer for processors, *TheHuzz* [36], as it achieves more coverage than prior hardware fuzzers, e.g., *DIFUZZRTL* [32], and

traditional random regression. To ensure a fair comparison, we constrain the environment of *JasperGold* to be the same as the hardware fuzzer's simulation environment [13, 32, 36, 43]. To prevent *JasperGold* from getting stuck while proving a property, we set a time limit on *JasperGold* to prove each property. We compute this time limit by using the time *JasperGold* spends on 30 random coverage points and applying survival analysis [38] to calculate a time limit large enough to prove over 99% of the points in the design. We set the rolling window size ($w$) as 100 to calculate $r_{fuzz}$ using Equation 2. We ran each experiment for 72 hours and repeated it three times.

## 5.2 Evaluating the Scheduling Strategy

We pick the coverage results of *HyPFuzz* on CVA6 to evaluate the coverage increment rate of the formal tool ($r_{fml}$) and the fuzzer ($r_{fuzz}$). The results are shown in Appendix E.

**Evaluation on $r_{fml}$.** We evaluate how the $r_{fml}$ will quickly converge and reflect how fast the formal tool will explore the hard-to-reach spaces of the fuzzer in a design. The *scheduler* updates $r_{fml}$ every time when *JasperGold* proves the reachability of an uncovered point. As shown in Figure 11, *HyPFuzz* switched to using *JasperGold* more than 300 times. The accumulated $r_{fml}$ only varies during the first 24 switches, after which the mean difference between the final $r_{fml}$ and the accumulated $r_{fml}$ is less than 5%. Hence, compared to the entire experiment, $r_{fml}$ only requires several samples to converge and can reflect the capability of space exploration of *JasperGold* in a design. The time limit on *JasperGold* also helps accelerate the convergence of $r_{fml}$.

**Evaluation on $r_{fuzz}(w)$.** We evaluate how the rolling window size ($w$) affects the accuracy of estimating how well the fuzzer in *HyPFuzz* recently explored the design space. The fuzzer in *HyPFuzz* executes 10 test cases at a time; hence we analyze the $r_{fuzz}(w)$ for different values of $w$ in increments of 10. For each value of $w$, we plot the number of times the *scheduler* will under-utilize the fuzzer and switch to a formal tool, as shown in Figure 12. We consider the fuzzer under-utilized if the coverage of the fuzzer is temporarily stagnated but will cover new points through mutation of test cases if *scheduler* had not switched to the formal tool. We can observe from Figure 12 that when the window is set to 100, $r_{fuzz}(w)$ can capture almost all coverage increments from the following test cases. Hence, we set the experiment $w$ to be 100.

## 5.3 Evaluating Point Selection Strategies

To find the most efficient selection strategy for *HyPFuzz*, we now evaluate the three point selection strategies *BotTop*, *MaxUncovd*, and *ModDep*, along with the *RandSel*, using two metrics: amount of coverage achieved and coverage speed.

**Coverage achieved.** Figure 8 shows the branch coverage achieved by different selection strategies for all the processors. The difference between the coverage achieved by the four selection strategies is less than 0.5% on all the proces-
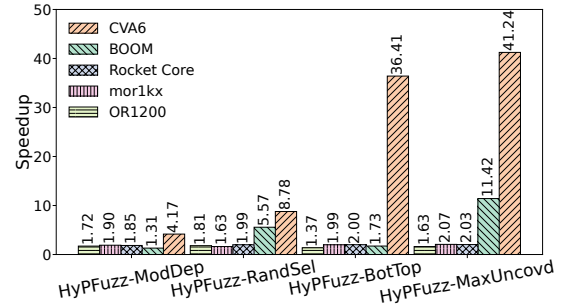


Figure 5: Speedup of *HyPFuzz* over *TheHuzz* [36].

sors except for the CVA6 processor, which has a difference of 1.31%. The difference in the coverage achieved by all the strategies decreases over time, and the coverage achieved will eventually converge. However, they still achieve an average of 1.72% more coverage compared to *TheHuzz* and 4.73% more coverage compared to random regression, across the five processors (see Section 5.4).

**Coverage speed** of the four selection strategies of *HyPFuzz* on the five processors is shown in Figure 5. The *MaxUncovd* selection strategy is the fastest for all processors except for the OR1200, where the *RandSel* strategy performs the best. The reason why *RandSel* strategy is the fastest for the OR1200 processor is because of its smaller size (around 25K gates), and *RandSel* has a faster selection time compared to other strategies. However, this is not the case for the other processors, especially the larger processors, such as CVA6 and BOOM, which have several hundred thousand gates and more microarchitectural features. For these designs, the order of point selection has a more dominant effect on coverage speed than the time taken to select the point. Based on these analyses, we select the *MaxUncovd* strategy as our point selection strategy for *HyPFuzz*. All further evaluations will use the *MaxUncovd* strategy unless stated otherwise.

## 5.4 Coverage Achieved

We now evaluate the capability of random regression, *TheHuzz*, and *HyPFuzz* in achieving coverage. Across the five processors, *HyPFuzz* achieves **4.73%** more coverage than random regression and is **239.93×** faster than random regression after fuzzing for 72 hours, as seen in Figure 8. Also, *HyPFuzz* achieves **1.72%** more coverage than *TheHuzz* and is **11.68×** faster than *TheHuzz* after running for the same 72 hours, as seen in Figure 8 and Figure 5, respectively.

The CVA6 and BOOM processors are complex with advanced OoO and SIMD features. Hence, *HyPFuzz* achieves substantially higher coverage than *TheHuzz* on them. Figure 6 shows that *HyPFuzz* achieves the highest branch coverage improvement of 6.84% on CVA6 compared to *TheHuzz*. Thus, the coverage speed of TheHuzz is less for these processors. However, *HyPFuzz* leverages the efficient *point selector* to generate
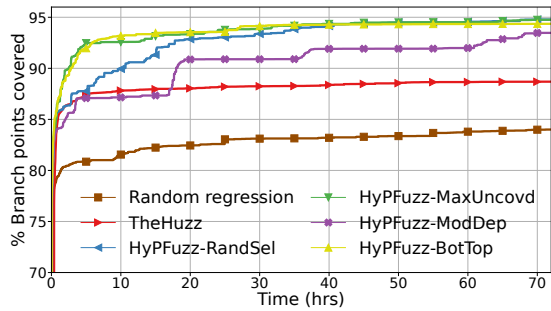
Figure 6: Total branch points covered by random regression, *TheHuzz* [36], and *HyPFuzz* on CVA6 [79].

seeds and maximize the coverage achieved by its fuzzer. This results in a speedup of $41.24\times$ on CVA6 and $11.42\times$ on BOOM compared to *TheHuzz*. On the other hand, the Rocket Core, mor1kx, and OR1200 do not have advanced OoO and SIMD features. However, *HyPFuzz* is still at least $1.5\times$ faster than *TheHuzz*. This is because the dynamic *scheduler* adapts to the complexity of the DUT using the values of $r_{fml}$ and $r_{fuzz}$, scheduling the fuzzer and the formal tool efficiently.

In summary, *HyPFuzz* achieved more coverage and is faster than random regression and *TheHuzz* on all five processors. *HyPFuzz* is faster when fuzzing processors, including the ones with complex micro-architectural features. However, as the complexity and size of the processor grow, the *point selector* and the *scheduler* become more effective, resulting in more speedup achieved by *HyPFuzz*.

## 5.5 Vulnerabilities Detected

While the coverage achieved sheds light on the extent of the DUT verified, it is not a direct measure of a tool's ability to detect vulnerabilities. Hence, we evaluate *HyPFuzz*'s ability to detect vulnerabilities in real-world processors.

**Vulnerability detection strategy** of *HyPFuzz* is the same as that of many existing fuzzers [13, 32, 36]. For the same test case, we output the architecture states of both the DUT and the golden reference model (GRM). The architecture state includes the value of general purpose registers (GPRs), control and status registers (CSRs), instructions committed, and exceptions triggered. Any mismatch between the architecture states indicates a potential vulnerability in the DUT or the GRM. In our experiment, we use the RISC-V ISA emulator, *Spike* [64], as the GRM for Rocket Core, BOOM, and CVA6 processors and the OpenRISC ISA emulator, *or1ksim* [56], for OR1200 and mor1kx processors.

Apart from detecting all the vulnerabilities detected by the most recent fuzzer [36], *HyPFuzz* detected **three new vulnerabilities**, as listed in Table 2.

**Vulnerability V1** is in the memory control unit of CVA6 processor and is similar to an out-of-bounds memory access vulnerability in software programs [66]. According to the

RISC-V specification [63], a processor must raise an exception when operations try to access data at invalid memory addresses. However, CVA6 will not raise exceptions in the same situation. We detected this vulnerability as a mismatch when *Spike* raised an exception for such operations and interrupted the program, whereas CVA6 continued to execute the program. Because operating systems usually use such exceptions to protect isolated executable memory space, missing them can allow an attacker to access data from all of memory (CWE-1252 [37]).

**Vulnerability V2** is located in the decode stage of CVA6 processor and is similar to an undefined behavior vulnerability in a software program [7]. According to the RISC-V specification [63], the decoder should throw an illegal instruction exception when the destination register (rd) of instruction MULH is the same as the first (rs1) or second (rs2) source register. This specification will reduce the utilization of multiplier units and increase the performance of processors. The vulnerability is that the decoder in CVA6 allows the rd of MULH to share the same register as rs1 or rs2. This means the design of CVA6 violates the ISA specification. We detected this vulnerability when *HyPFuzz* generated a test case containing MULH with the same rd, rs1, and rs2. *Spike* threw an illegal instruction exception, whereas CVA6 executed the instruction. This vulnerability can result in a potential performance bottleneck when executing applications with heavy multiplier operations, such as machine learning (CWE-440 [59]).

**Vulnerability V3** is a cross-modular vulnerability where the read logic in the CSR module enables access to undefined hardware performance counters (HPCs), resulting in unknown values when these HPCs are accessed. It is similar to an undefined behavior in a software program [7]. CVA6 has implemented 14 HPCs for recording various hardware behaviors, and its CSR module is responsible for reading the value of an HPC based on requests from the operating system. However, the reading logic in the CSR module enables access to 32 HPCs, which causes X-propagation when reading nonexistent HPCs. We detected this vulnerability when *Spike* returns regular numbers while CVA6 returns X (unknown) values. This vulnerability will cause potential issues during synthesis and fail the functions of HPC (CWE-1281 [24]).

Vulnerabilities **V2** and **V3** resulted in two new common vulnerabilities and exposures (CVE) entries, CVE-2022-33021 and CVE-2022-33023.

**Comparison with *TheHuzz*.** In addition to the three new vulnerabilities, *HyPFuzz* detected all 11 vulnerabilities reported by *TheHuzz* as listed in Table 2. The speedup column shows how fast *HyPFuzz* is compared to *TheHuzz* in terms of both run-time and number of instructions. It can be seen that the speed of vulnerability detection follows the same trend as the coverage speeds. *HyPFuzz* detects vulnerabilities in CVA6 at faster than the vulnerabilities in Rocket Core, mor1kx, and OR1200 processors. However, *TheHuzz* detected some vulnerabilities faster than *HyPFuzz*. This is because the

Table 2: Vulnerabilities detected by *HyPFuzz*. N.A. denotes "Not Applicable."

| Processor | Vulnerability Description | CWE | New? | # Instructions | | | Time (sec) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *TheHuzz* | *HyPFuzz* | Speedup | *TheHuzz* | *HyPFuzz* | Speedup |
| CVA6 [79] | V1: Missing exceptions when accessing invalid addresses. | CWE-1252 | ✓ | *N.A.* | $2.67 \times 10^1$ | *N.A.* | *N.A.* | $6.67 \times 10^1$ | *N.A.* |
| | V2: Incorrect decoding logic for multiplication instructions. | CWE-440 | ✓ | *N.A.* | $9.80 \times 10^4$ | *N.A.* | *N.A.* | $4.20 \times 10^3$ | *N.A.* |
| | V3: Returning X-value when access unallocated CSRs. | CWE-1281 | ✓ | *N.A.* | $2.08 \times 10^3$ | *N.A.* | *N.A.* | $1.70 \times 10^2$ | *N.A..* |
| | V4: Failure to detect cache coherency violation. | CWE-1202 | ✗ | $1.72 \times 10^5$ | $2.15 \times 10^5$ | 0.80× | $6.50 \times 10^3$ | $8.40 \times 10^3$ | 0.77× |
| | V5: Incorrect decoding logic for the FENCE.I instruction. | CWE-440 | ✗ | $1.36 \times 10^4$ | $1.08 \times 10^4$ | 1.26× | $1.68 \times 10^3$ | $3.08 \times 10^2$ | 5.45× |
| | V6: Incorrect exception type in instruction queue. | CWE-1202 | ✗ | $4.02 \times 10^4$ | $7.40 \times 10^3$ | 5.43× | $2.54 \times 10^3$ | $2.92 \times 10^2$ | 8.69× |
| | V7: Missing exceptions for some *illegal* instructions. | CWE-1242 | ✗ | $1.81 \times 10^6$ | $1.43 \times 10^5$ | 12.66× | $5.67 \times 10^4$ | $6.41 \times 10^3$ | 8.85× |
| Rocket Core [11] | V8: Instruction commit count not increased when EBREAK. | CWE-1201 | ✗ | $7.76 \times 10^2$ | $1.07 \times 10^2$ | 7.25× | $1.19 \times 10^2$ | $6.67 \times 10^1$ | 1.78× |
| mor1kx [53] | V9: Incorrect implementation of the carry flag generation. | CWE-1201 | ✗ | 20 | 20 | 1.00× | 10 | 10 | 1.00× |
| | V10: Missing access checking for privileged register. | CWE-1262 | ✗ | $4.46 \times 10^5$ | $2.30 \times 10^5$ | 1.94× | $1.97 \times 10^4$ | $8.20 \times 10^3$ | 2.40× |
| | V11: Incomplete implementation of the EEAR register write logic. | CWE-1199 | ✗ | $1.12 \times 10^5$ | $2.13 \times 10^5$ | 0.53× | $4.89 \times 10^3$ | $7.60 \times 10^3$ | 0.64× |
| OR1200 [55] | V12: Incorrect forwarding logic for the GPR0. | CWE-1281 | ✗ | $1.05 \times 10^3$ | $1.23 \times 10^3$ | 0.85× | $1.12 \times 10^2$ | 60 | 1.87× |
| | V13: Incorrect overflow logic for MSB & MAC instructions. | CWE-1201 | ✗ | 47 | 73 | 0.64× | $1.25 \times 10^1$ | $5.05 \times 10^1$ | 0.25× |
| | V14: Incorrect generation of overflow flag. | CWE-1201 | ✗ | $2.21 \times 10^4$ | $1.78 \times 10^4$ | 1.24× | $1.34 \times 10^3$ | $6.72 \times 10^2$ | 1.99× |

number of instructions required to trigger them is too low (< 1000 instructions) (i.e., measure the coverage and mutate test cases). For example, V9 and V13 require less than one hundred instructions to trigger. On an average, *HyPFuzz* detects vulnerabilities **3.06×** faster than *TheHuzz* in terms of run time and uses **3.05×** fewer number of instructions to trigger them.

## 5.6 Evaluation on Compatibility of *HyPFuzz*

In this section, we demonstrate the compatibility of *HyPFuzz* in achieving coverage with different coverage metrics such as *condition* and *finite-state machine* (FSM) coverage. *HyPFuzz* uses its *property generator* to generate the *cover* properties for the points of them as discussed in Appendix B. Figure 7 shows the coverage achieved by *HyPFuzz* and *TheHuzz* on CVA6 processor when run for 72 hours and repeated three times. *HyPFuzz* achieves an average of 15.95% more *condition* coverage and 33.76% more *FSM* coverage compared to *TheHuzz* hence demonstrating that *HyPFuzz* is capable of improving coverage for different types of coverage metrics.

## 6 Related Works

**Hardware fuzzers.** *RFUZZ* [43] uses mux-toggle coverage as feedback to fuzz hardware and was one of the first attempts towards creating hardware fuzzers. The mux-toggle coverage covers the activity in the select signals of muxes in the
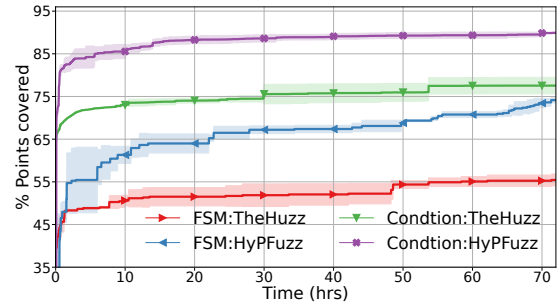


Figure 7: Total condition points and FSM points covered by *TheHuzz* [36] and *HyPFuzz*.

DUT. *DirectFuzz* [13] leverages the mux-toggle coverage and allocates more mutation energy to test cases that achieve coverage closed to a manually selected model, hence achieving faster coverage. However, the mux-toggle metric does not scale well to large designs like BOOM and CVA6 due to instrumentation overhead. *DIFUZZRTL* [32] addresses the overhead limitation of *RFUZZ* by developing a new coverage metric, control-register coverage. It uses all possible combinations of values of all the registers driving the selection logic of MUX. However, this coverage metric does not assign coverage points to many registers as well as combinational logic, thereby missing many security-critical vulnerabilities [36]. Therefore, *TheHuzz* [36] uses code coverage metrics (branch,

condition coverage, etc.) as feedback information to guide the fuzzer. *TheHuzz* cannot verify the DUT efficiently as it is increasingly difficult to cover the coverage points in hard-to-reach regions of hardware (see Section 3.2). The coverage achieved by this processor is only 63%. ***Fuzzing hardware like software*** [74], unlike other hardware fuzzers, first translates a hardware design into a software model, then uses the coverage-guided software fuzzers to fuzz hardware. However, the software model does not support all the hardware constructs like latches and floating wires [36].

In summary, existing hardware fuzzers suffer from one or more of the following limitations: they (i) cannot scale to large designs, (ii) cannot capture all hardware behavior in RTL, thereby missing vulnerabilities, (iii) are not designed to fuzz the entire DUT, or (iv) do not support all the constructs like latches and floating wires.

**Hybrid techniques.** Existing software hybrid techniques combine fuzzers with symbolic execution [25,26,57,58,70,81,82]. Fuzzers leverage symbolic execution's constraint-solving capabilities to explore *deep* execution paths. Meanwhile, symbolic execution utilizes concrete test cases generated by fuzzers to mitigate scalability issues [35]. Therefore, most hybrid fuzzers run symbolic execution in parallel to keep calculating paths covered by test cases and also use symbolic execution to generate new test cases when the original program terminates normally (halt) or abnormally (e.g., crash) [25, 26, 57, 58, 81]. ***Driller*** [70] invokes symbolic execution when the fuzzer cannot achieve coverage after a pre-defined time threshold. Moreover, *Driller* will select an unexplored path close to the paths covered by test cases. ***DART*** [25] applies a depth-first search along the path tree, and ***SAGE*** [26] selects paths under the paths explored by previous inputs with maximal coverage increment. ***Pak's*** strategy [57] profiles unexplored paths and assigns difficult ones with higher priority to be selected, and ***Digfuzz*** [81] applies Monte Carlo methods [65] to quantify the difficulty of unexplored paths with probability.

Compared to software hybrid fuzzers, *HyPFuzz*: (i) generates SVA properties and invokes formal tools, ensuring compatibility with various coverage metrics for different security and verification purposes rather than just path, (ii) does not require formal tools to check the test cases generated by the fuzzers, utilizing formal tools only when necessary, (iii) develops multiple uncovered point selection strategies based on computer architecture and hardware design routines [29,67].

Apart from these fuzzing techniques, researchers have combined various formal verification techniques with either random regression [30,40] or fuzzing [45] in the context of test generation, targeting faults in hardware rather than security vulnerabilities, which is the focus of *HyPFuzz*. Also, none of these techniques scale to large, real-world designs.

In contrast, *HyPFuzz*: (i) is scalable to large real-world processors, (ii) captures all activity in the hardware details, such as FSMs and combinational logic, (iii) fuzzes the entire

processor, (iv) uses industry-standard hardware simulators that support all hardware constructs and has the ability to detect vulnerabilities.

## 7 Discussion

**Utilization of point selection strategies.** *HyPFuzz* specifically adopts the *MaxUncovd* strategy to select uncovered points since it shows the highest coverage speed compared to the other strategies. However, we can modify *HyPFuzz* to switch from them dynamically. We can assign weight to each strategy as the probability of it being selected by *HyPFuzz*. We then use evolutionary algorithms, such as particle swarm optimization algorithm [47, 68], to dynamically update the weights for fast and high coverage.

**FPGA emulations** is $10\times$ faster than hardware simulation [32, 43]. *HyPFuzz* uses hardware simulators to instrument and collect the coverage data. Thus, currently, it does not readily support FPGA emulations. One can provide FPGA emulation support for *HyPFuzz* by first instrumenting the DUT using existing tools such as Verific [75] or by modifying intermediate representation (IR) compilers [13, 32, 43]. The instrumented DUT will then be emulated on the FPGA while the formal tool and the rest of the components of *HyPFuzz* run on the host machine. Thus, one can use *HyPFuzz* to fuzz designs emulated on an FPGA.

## 8 Conclusion

Existing hardware fuzzers do not fuzz the hard-to-reach parts of the processor, thereby missing security vulnerabilities. *HyPFuzz* tackles this challenge by using a formal tool to help fuzz the hard-to-reach parts. We fuzzed five open-source processors, including a million-gate `BOOM` processor. *HyPFuzz* found three new memory and undefined behavior-related vulnerabilities and detected all the existing vulnerabilities $3.06\times$ faster than the most recent processor fuzzer. It also achieved $239.93\times$ faster coverage than random regression and $11.68\times$ faster coverage than the most recent processor fuzzer.

**Responsible disclosure**. We responsibly disclosed the vulnerabilities to the designers.

## 9 Acknowledgement

## References

[1] Accelerating exhaustive and complete verification of RISC-V processors, 2021. `https://riscv.org/news/2021/08/accelerating-exhaustive-and-complete-verification-of-risc-v-processors-ashish-darbari-axiomise/`. Last accessed on 09/30/2022.

[2] Prologue: The 2020 Wilson Research Group Functional Verification Study. `https://blogs.sw.siemens.com/verificationhorizons/2020/11/05/part-1-the-2020-wilson-research-group-functional-verification-study/`, 2020. Last accessed on 02/13/2022.

[3] Jasper Engine Selection Guide. `https://www.cadence.com/en_US/home/support.html`, 2022. Last accessed on 01/11/2023.

[4] Jasper RTL Apps. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html`, 2022. Last accessed on 02/11/2022.

[5] ModelSim. `https://eda.sw.siemens.com/en-US/ic/modelsim/`, 2022. Last accessed on 05/02/2022.

[6] Questa Formal Verification Apps. `https://eda.sw.siemens.com/en-US/ic/verification-and-validation/formal-verification/`, 2022. Last accessed on 09/16/2022.

[7] Undefined behavior. `https://en.cppreference.com/w/cpp/language/ub`, 2022. Last accessed on 09/30/2022.

[8] VC Formal. `https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html`, 2022. Last accessed on 02/11/2022.

[9] VCS. `https://www.synopsys.com/verification/simulation/vcs.html`, 2022. Last accessed on 05/02/2022.

[10] A. Amid, D. Biancolin, et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[11] K. Asanović, R. Avizienis, et al. The Rocket Chip Generator. (UCB/EECS-2016-17), Apr 2016.

[12] B. Bailey. Incremental Design Breakdown. `https://github.com/openrisc/mor1kx`, 2022. Last accessed on 04/08/2022.

[13] S. Canakci, L. Delshadtehrani, et al. Directfuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing. *Design Automation Conference*, 2021.

[14] E. Clarke, O. Grumberg, et al. Progress on the State Explosion Problem in Model Checking. *Informatics*, pages 176–194, 2001.

[15] E. M. Clarke, T. A. Henzinger, et al. Handbook of Model Checking. 10, 2018.

[16] D. Cyrluk, S. Rajan, et al. Effective Theorem Proving for Hardware Verification. *International Conference on Theorem Provers in Circuit Design*, 1994.

[17] S. Das, C. Karfa, et al. Formal Modeling of Network-on-Chip Using CFSM and Its Application in Detecting Deadlock. *IEEE Transactions on Very Large Scale Integration Systems*, 28(4):1016–1029, 2020.

[18] R. De Clercq, F. Piessens, et al. Secure Interrupts on Low-End Microcontrollers. *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 147–152, 2014.

[19] G. Dessouky, D. Gens, et al. HardFails: Insights into Software-Exploitable Hardware Bugs. *USENIX Security Symposium*, pages 213–230, 2019.

[20] C. Deutschbein, A. Meza, et al. Toward Hardware Security Property Generation at Scale. *IEEE Security and Privacy*, (01):2–10, 2022.

[21] E. El Mandouh and A. G. Wassal. Estimation of Formal Verification Cost Using Regression Machine Learning. *IEEE International High Level Design Validation and Test Workshop*, pages 121–127, 2016.

[22] H. Eldib, C. Wang, et al. Formal Verification of Software Countermeasures against Side-Channel Attacks. *ACM Transactions on Software Engineering and Methodology*, 24(2):1–24, 2014.

[23] M. R. Fadiheh, J. Müller, et al. A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors. *ACM/IEEE Design Automation Conference*, pages 1–6, 2020.

[24] N. Fern. CWE-1281. `https://cwe.mitre.org/data/definitions/1281.html`, 2020. Last accessed on 09/12/2022.

[25] P. Godefroid, N. Klarlund, et al. DART: Directed Automated Random Testing. *ACM conference on Programming language design and implementation*, 2005.

[26] P. Godefroid, M. Y. Levin, et al. Automated Whitebox Fuzz Testing. *NDSS*, 8:151–166, 2008.

[27] I. Graja, S. Kallel, et al. A comprehensive survey on modeling of cyber-physical systems. *Concurrency and Computation: Practice and Experience*, 32(15), 2020.

[28] S. L. W. Group. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017*, 2018.

[29] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. 2011.

[30] P.-H. Ho, T. Shiple, et al. Smart Simulation Using Collaborative Formal and Simulation Engines. *International Conference on Computer-Aided Design.*, 2000.

[31] W. Hu, A. Ardeshiricham, et al. Hardware Information Flow Tracking. *ACM Computing Surveys*, 2021.

[32] J. Hur, S. Song, et al. DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs. *IEEE Symposium on Security and Privacy*, pages 1286–1303, 2021.

[33] R. J. Hyndman. Moving Averages. 2011.

[34] M. Ivanković, G. Petrović, et al. Code Coverage at Google. *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019.

[35] I. B. Kadron, Y. Noller, et al. Fuzzing, Symbolic Execution, and Expert Guidance for Better Testing. *IEEE Software*, 2023.

[36] R. Kande, A. Crump, et al. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *USENIX Security Symposium*, pages 3219–3236, 2022.

[37] A. Kanuparthi, H. Khattri, et al. CWE-1252. `https://cwe.mitre.org/data/definitions/1252.html`, 2022. Last accessed on 09/12/2022.

[38] D. G. Kleinbaum and M. Klein. Survival Analysis a Self-Learning Text. 1996.

[39] P. Kocher, J. Horn, et al. Spectre Attacks: Exploiting Speculative Execution. *IEEE Symposium on Security and Privacy*, 2019.

[40] A. Kolbi, J. Kukula, et al. Symbolic RTL Simulation. *IEEE/ACM Design Automation Conference*, 2001.

[41] P. Krishnamurthy, R. Karri, et al. Anomaly Detection in Real-Time Multi-Threaded Processes Using Hardware Performance Counters. *IEEE Transactions on Information Forensics and Security*, 15:666–680, 2019.

[42] Y. Kukimoto. Introduction to Formal Verification. `https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html`, 2011. Last accessed on 09/12/2022.

[43] K. Laeufer, J. Koenig, et al. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. *IEEE International Conference on Computer-Aided Design*, 2018.

[44] lcamtuf. American Fuzzy Lop (AFL) Fuzzer. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2014. Last accessed on 02/07/2022.

[45] T. Li, H. Zou, et al. Symbolic Simulation Enhanced Coverage-Directed Fuzz Testing of RTL Design. *IEEE International Symposium on Circuits and Systems*, 2021.

[46] M. Lipp, M. Schwarz, et al. Meltdown: Reading Kernel Memory from User Space. *USENIX Security*, 2018.

[47] C. Lyu, S. Ji, et al. MOPT: Optimized Mutation Scheduling for Fuzzers. *USENIX Security Symposium*, 2019.

[48] Y. A. Manerkar, D. Lustig, et al. RTLCheck: Verifying the Memory Consistency of RTL Designs. *IEEE/ACM International Symposium on Microarchitecture*, 2017.

[49] MITRE. CWE VIEW: Hardware Design. `https://cwe.mitre.org/data/definitions/1194.html`, 2019. Last accessed on 05/02/2022.

[50] A. Mockus, N. Nagappan, et al. Test Coverage and Post-Verification Defects: A Multiple Case Study. *IEEE/ACM International Symposium on Empirical Software Engineering and Measurement*, 2009.

[51] S. K. Muduli, G. Takhar, et al. HyperFuzzing for SoC Security Validation. *ACM/IEEE International Conference on Computer-Aided Design*, pages 1–9, 2020.

[52] Y. Naveh, M. Rimon, et al. Constraint-Based Random Stimuli Generation for Hardware Verification. *AI magazine*, 28(3):13–13, 2007.

[53] OpenRISC. mor1kx source code. `https://github.com/openrisc/mor1kx`, 2020. Last accessed on 04/08/2021.

[54] OpenRISC. OpenRISC Homepage. `https://openrisc.io/`, 2020. Last accessed on 04/08/2021.

[55] OpenRISC. or1200 source code. `https://github.com/openrisc/or1200`, 2020. Last accessed on 04/08/2021.

[56] OpenRISC. Or1ksim Source Code. `https://github.com/openrisc/or1ksim`, 2020. Last accessed on 04/08/2021.

[57] B. S. Pak. Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. *School of Computer Science Carnegie Mellon University*, 2012.

[58] V.-T. Pham, M. Böhme, et al. Model-Based Whitebox Fuzzing for Program Binaries. *International Conference on Automated Software Engineering*, 2016.

[59] PLOVER. CWE-440. https://cwe.mitre.org/data/definitions/440.html, 2022. Last accessed on 09/12/2022.

[60] H. Ponce-de León and J. Kinder. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. *IEEE Symposium on Security and Privacy*, 2022.

[61] R. J. Punnoose, R. C. Armstrong, et al. Survey of Existing Tools for Formal Verification. 2014.

[62] J. Rajendran, A. M. Dhandayuthapany, et al. Formal Security Verification of Third Party Intellectual Property Cores For Information Leakage. *IEEE/ACM International Conference on VLSI Design and International Conference on Embedded Systems*, 2016.

[63] RISC-V. RISC-V Webpage. https://riscv.org/, 2021. Last accessed on 04/08/2021.

[64] RISC-V. SPIKE Source Code. https://github.com/riscv/riscv-isa-sim, 2021. Last accessed on 05/12/2022.

[65] C. P. Robert. Monte Carlo Methods in Statistics. *arXiv:0909.0389*, 2009.

[66] K. Serebryany, D. Bruening, et al. AddressSanitizer: A Fast Address Sanity Checker. *USENIX Annual Technical Conference*, pages 309–318, 2012.

[67] J. P. Shen and M. H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. 2013.

[68] Y. Shi and R. Eberhart. A Modified Particle Swarm Optimizer. *IEEE international conference on evolutionary computation proceedings.*, 1998.

[69] W. Snyder. Verilator. https://www.veripool.org/wiki/verilator, 2021. Last accessed on 04/08/2021.

[70] N. Stephens, J. Grosen, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *NDSS*, 2016.

[71] Synopsys. Synopsys Webpage. https://www.synopsys.com/, 2021. Last accessed on 04/08/2021.

[72] Synopsys. Accelerating Verification Shift Left with Intelligent Coverage Optimization. https://www.synopsys.com/cgi-bin/verification/dsdla/pdfr1.cgi?file=ico-wp.pdf, 2022. Last accessed on 02/18/2023.

[73] C. Trippel, D. Lustig, et al. Checkmate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. *IEEE International Symposium on Microarchitecture*, 2018.

[74] T. Trippel, K. G. Shin, et al. Fuzzing Hardware Like Software. *USENIX Security Symposium*, 2022.

[75] Verific. Verific Design Automation. https://www.verific.com/, 2022. Last accessed on 04/08/2021.

[76] X. Wang and R. Karri. Numchecker: Detecting Kernel Control-flow Modifying Rootkits by Using Hardware Performance Counters. *Design Automation Conference*, 2013.

[77] B. Wile, J. Goss, et al. Comprehensive Functional Verification: The Complete Industry Cycle. 2005.

[78] H. Witharana, Y. Lyu, et al. A Survey on Assertion-based Hardware Verification. *ACM Computing Surveys*, 2022.

[79] 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. *IEEE Transactions on Very Large Scale Integration Systems*, 2019.

[80] J. Zhao, B. Korpan, et al. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *4th Workshop on Computer Architecture Research with RISC-V*, 2020.

[81] L. Zhao, Y. Duan, et al. Send Hardest Problems My Way: Probabilistic path prioritization for hybrid fuzzing. *NDSS*, 2019.

[82] X. Zhu, S. Wen, et al. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

# Appendix

# A   Total branch coverage achieved by random regression, *TheHuzz* [36], and *HyPFuzz*

Figure 8 shows the branch coverage achieved by random regression, *TheHuzz* [36], and different point selection strategies of *HyPFuzz*. Across the five processors, *HyPFuzz* achieves **4.73%** more coverage than random regression after fuzzing for 72 hours, as seen in Figure 8. Also, *HyPFuzz* achieves **1.72%** more coverage than *TheHuzz* after running for the same 72 hours. More details are discussed in Section 5.4.
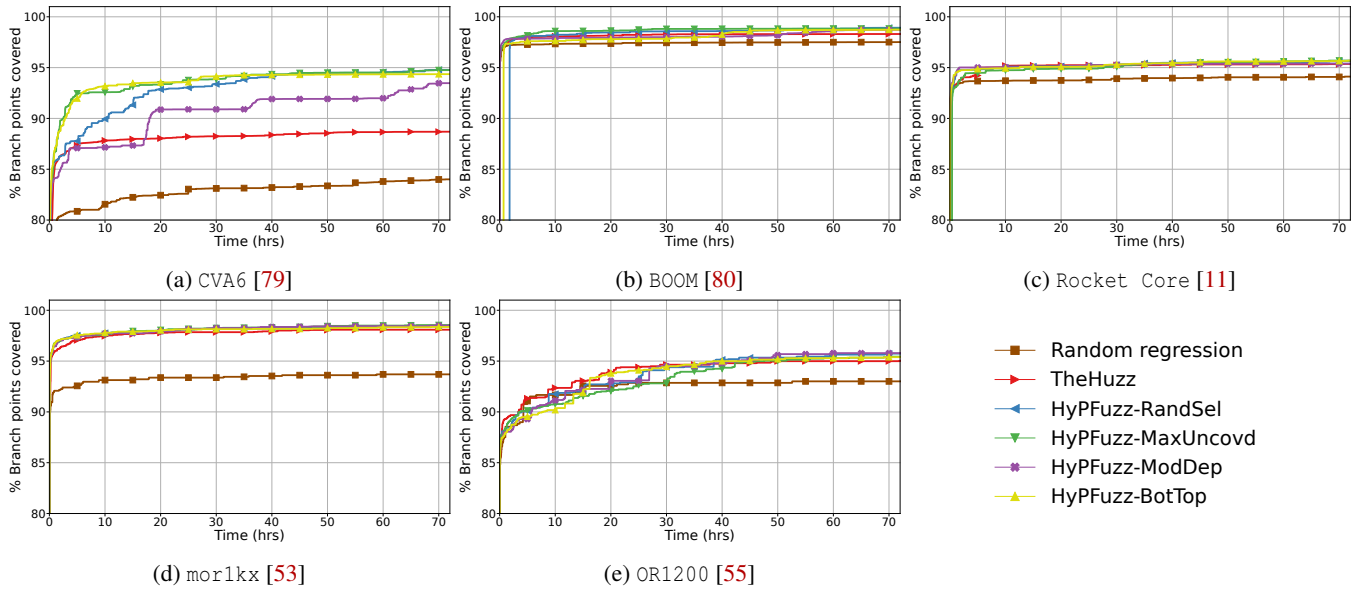
Figure 8: Total branch points covered by random regression, *TheHuzz* [36], and *HyPFuzz*.

Listing 4: Code snippet showing different coverage metrics.

```verilog
module code_cov_example (input a,b,c,output d);
    reg [1:0] state_d,state_q;
    always@(*)begin
        case (start_q) begin
            IDLE: begin // FSM
                if (a || b && c) begin // branch, condition
                    state_d = FINISH;
                end else begin
                    state_d = a & c; // expression
                ...
            FINISH: begin // FSM
                d = 1'b1; // toggle
```

## B  The *Property Generator* for Different Coverage Metrics

*HyPFuzz* currently uses the branch, condition, and finite-state-machine (FSM) coverage metrics for evaluation to demonstrate the compatibility of *HyPFuzz* with different coverage metrics. Our *property generator* is similarly compatible with other coverage metrics also, as we can translate the coverage points to *cover* properties needed by the formal tools, such as *JasperGold* [4]. Usually, the coverage metrics are reported as either a signal name or Boolean expression. Hence, it is always possible to translate these metrics into SystemVerilog Assertions (SVA) *cover* properties because the *cover property* is any legal, temporal logic (TL) expression [28] with the form: *cover property* $< TL - expression >$, where $< TL - expression >$ can be temporal in general. We use Listing 4 as an example to show how to generate SVA *cover properties* for different code coverage metrics.

**Branch coverage** allocates coverage points following the branch statement (i.e., two coverage points, one for *if* branch statement in Line 6 taken and another one for *if* not taken). Following the branch statement tree, the conditions of cov-

ering the point for taking the *if* branch statement will be $(start\_q == IDLE)$ and $(a||b\&\&c)$. And, the condition to check the branch not taken is $(start\_q == IDLE)$ and $not(a||b\&\&c)$. The *cover* properties for the two points can be specified in the SVA format as:

*cover property* $((start\_q == IDLE)\&\&(a||b\&\&c))$
*cover property* $((start\_q == IDLE)\&\&!(a||b\&\&c))$

**Condition coverage** allocates coverage points for all possible combinations of values for the signals in a branch statement (i.e., three 1-bit signals in the *if* branch statement in Line 6 lead to eight condition coverage points.) The *cover* properties:

*cover property* $(a == 1'b0\&\&b == 1'b0\&\&c == 1'b0)$

$$\vdots$$

*cover property* $(a == 1'b1\&\&b == 1'b1\&\&c == 1'b1)$

**Expression coverage** allocates coverage points for all possible combinations of values for the signals in an assignment (i.e., two 1-bit signals in Line 9 lead to four expression coverage points). The *cover* properties:

*cover property* $(a == 1'b0\&\&c == 1'b0)$

$$\vdots$$

*cover property* $(a == 1'b1\&\&c == 1'b1)$

**Finite-state machine (FSM) coverage** allocates two sets of coverage points to check the FSM in a module. The first set of point checks if state registers have reached all possible state values. The second set of point captures all state transitions in the FSM. Lines 5 and 11 show the two state values and at least one state transition of the *state_q* register. The *cover* properties in SVA format:

(1) checking FSM states:

*cover property* ($state\_q == IDLE$)

$\vdots$

*cover property* ($state\_q == FINISH$)

(2) checking state transitions:

*cover property*

($state\_q == IDLE$ ##1 $state\_q == FINISH$)

**Toggle coverage** entails verifying whether specific bits have been flipped during simulation or not. Line 12 contains a toggle coverage point that verifies if the 1-bit output signal $d$ has been flipped from 0 to 1. The *cover* property:

*cover property* ($d == 1'b0 \mapsto d == 1'b1$)

## C   Case study example for Boolean assignments and Test case converter

Figure 9 shows the Boolean assignments output by *Jasper-Gold* after proving a *cover* property. Figure 10 shows the template disassembly file used to determine the address of the *TEST* instructions, the binary file template, and the valid binary file generated by the *test case converter* to cover the S_EXT interrupt branch point. More details are in Section 3.4.
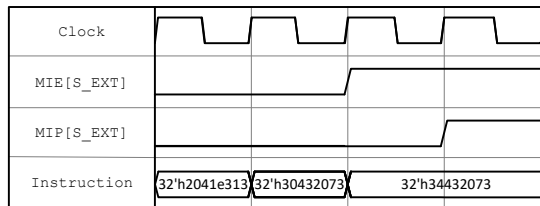


Figure 9: Boolean assignments from *JasperGold* [4].



Figure 10: Test case conversion.

## D   *ModDep* algorithm

Algorithm 2 is the method to calculate the fanout COI of each module. The module with the highest fanout COI will be selected first. *HyPFuzz* will prioritize modules based on the dependence before running the experiment. Hence, unlike

the *MaxUncovd* strategy, the priority of module dependence will not change. More details are discussed in Section 4.4.3.

---

**Algorithm 2:** *ModDep* strategy

**Input:** $M = \{m_0, m_1, ..., m_n\}$: a set of modules;
**Output:** $M'$, ($|M'| = |M|$): ordered set of modules based on *ModDep* strategy;

1  $M' \leftarrow \emptyset$
2  $|COI| \leftarrow |M|$ // store COI of the modules
3  **for** $j \leftarrow 0...(|M| - 1)$ **do**
4     $coi_j \leftarrow 0$
5     **for** $output \in$ GETOUTPUTS($m_j$) **do**
6        $coi_j \leftarrow coi_j +$ MEASURECOI($output$)

   /* sort the module set based on the fanout COI of the modules          */
7  $M' \leftarrow$ SORT($M, COI, MaxToMin$)
8  **return** $M'$

---

## E   Evaluation results of $r_{fuzz}$ and $r_{fml}$

Figure 11 and Figure 12 show the evaluation results of coverage increment rate of formal tool ($r_{fml}$) and fuzzer ($r_{fuzz}$) respectively on CVA6 processor. Figure 11 evaluates the $r_{fml}$ overtime, and Figure 12 evaluates the number of underutilization of fuzzers when the window size ($w$) is insufficient. More details are discussed in Section 5.2.



Figure 11: The change of $r_{fml}$ overtime.
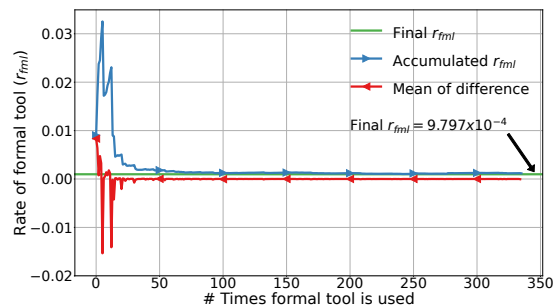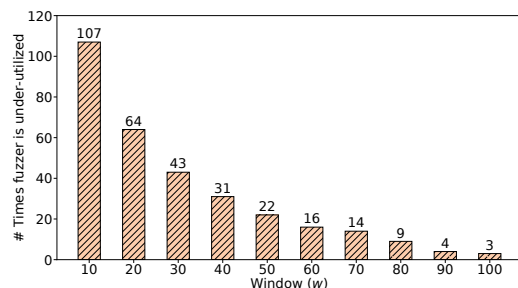


Figure 12: Sampling results of underutilization of fuzzer.