Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection

Matej Bölcskei
ETH Zurich

Flavien Solt
ETH Zurich

Katharina Ceesay-Seitz
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

Hardware fuzzing has recently gained momentum with many discovered bugs in open-source RISC-V CPU designs. Comparing the effectiveness of different hardware fuzzers, however, remains a challenge: each fuzzer optimizes for a different metric and is demonstrated on different CPU designs. Furthermore, the number of newly-discovered bugs is not an appropriate metric since finding new bugs becomes increasingly more difficult as designs mature. We argue that a corpus of automatically injectable bugs will help compare hardware fuzzers to better understand their strengths and weaknesses. Through a large-scale study of 177 software-observable bugs in open-source RISC-V CPUs, we discover that CPU bugs can be modelled by manipulating conditional statements or signal drivers. Based on this observation, we design ENCARSIA, a framework that automatically transforms the intermediate representation of a given CPU design to inject bugs that are equivalent to incorrect conditions or assignments at the HDL level. To ensure that an injected bug has an observable architectural effect, we leverage formal methods to prove the existence of an architectural deviation due to the bug-specific transformation. We evaluate ENCARSIA by injecting bugs into three open-source RISC-V CPUs, fuzzing these CPUs with recently-proposed CPU fuzzers, and comparing their bug-finding performance. Our experiments reveal key insights into the limitations of existing hardware fuzzers, including their inability to cover large sections of the designs under test, ineffective coverage metrics, and bug detection mechanisms that often miss bugs or produce false positives, highlighting the urgent need to reassess current approaches.

1 Introduction

To satisfy the ever-increasing demand for diverse hardware targeting wide ranges of applications from secure Internet of Things (IoT) devices to high-performance computing (HPC) systems, new hardware is being developed at an unprecedented pace. This momentum has created a high demand for usable and effective tools and methodologies to verify the correctness and security of these hardware designs at all stages of their development.

Hardware fuzzing is a recent and popular response to this pressing demand [5,6,9,14,22,26,29,30,32,42,45,48]. Given its scalability, ease of adoption, and demonstrated ability to find bugs in real-world designs, hardware fuzzing has become an essential tool in the hardware validation toolbox. However, hardware fuzzing research often makes conflicting claims as to the underlying mechanism behind their effectiveness, revealing a profound lack of understanding about what truly drives fuzzing performance.

Fuzzer evaluation. We posit that this confusion stems from the lack of a standardized framework for evaluating hardware fuzzers. Instead, existing work relies on either some coverage that they achieve during the fuzzing process, or the number of new naturally-occurring bugs they find in real-world designs. Coverage is known to be a poor proxy for bug-finding ability in software [28], and the discovery of new naturally-occurring bugs is a noisy metric that is hard to compare across different fuzzers due to diverse tested hardware designs and versions. In particular, a fuzzer that finds bugs in a given design may find fewer new bugs due to the increasing maturity of the design, even if it is capable of finding more complex bugs. Inserting bugs manually is a labor-intensive option [16, 40] that has to be repeated per design, in particular if the bug-inserted designs are not shared with the community. Automatic injection of bugs is a promising alternative.

Encarsia. We introduce ENCARSIA[†] , the first framework for automatically injecting realistic bugs into arbitrary RTL designs. To guide the design of ENCARSIA, we first conduct a comprehensive survey of bugs reported in four popular open-source RISC-V CPUs of various complexities and design paradigms: Ibex [33], CVA6 [20], Rocket [1] and BOOM [4]. We find that all these bugs resemble two simple syntactic

[†]Encarsia is a genus of tiny parasitic wasps belonging to the family Aphelinidae. These wasps are commonly used in biological control programs to manage pest populations, particularly whiteflies and scale insects.

transformations: mix-ups of signals or logic expressions and errors in conditional statements. Based on this observation, ENCARSIA applies these syntactic transformations to automatically inject diverse and realistic bugs in a given HDL design. However, such transformations may not necessarily be reachable or result in an architecturally-visible effect. To certify that the injected bugs are observable and have a functional effect, we rely on formal verification.

We evaluate ENCARSIA on three RISC-V CPUs, namely Ibex, Rocket and BOOM, and evaluate three state-of-the-art hardware fuzzers: DifuzzRTL [26], ProcessorFuzz [6], and Cascade [42]. We find that these fuzzers are able to find respectively 41.67%, 41.67%, and 40% of the bugs injected by ENCARSIA in 24 hours of fuzzing. ENCARSIA reveals shortcomings in both the design and evaluation of these fuzzers: (1) existing fuzzers fail to reach large sections of the designs due to limitations in their test generation mechanism, (2) for parts of the designs that the fuzzers do reach, existing coverage metrics fail at effectively guiding the fuzzers to exercise those areas sufficiently, and (3) current fuzzers' bug filtering mechanisms result in both false positives and false negatives. Based on these insights, we propose future research directions for improving hardware fuzzing. For example, fuzzers might benefit from higher-level coverage metrics, either at the architectural level or at the microarchitectural level, given that current fuzzers seem neither able to reach good ISA coverage, nor to explore all microarchitectural corner cases.

Contributions. We make the following contributions:

- We survey the bugs reported in four popular open-source RISC-V CPUs: Ibex, CVA6, Rocket and BOOM.
- We design and implement ENCARSIA based on a new model for automatically injecting realistic bugs into RTL designs based on multiplexer tree corruptions and wire connection mix-ups, and a formal methodology to ensure that injected bugs are architecturally observable.
- We implement and evaluate ENCARSIA by injecting bugs in three popular open-source RISC-V CPUs and assessing three state-of-the-art hardware fuzzers. Our assessment provides insights into the limitations of existing hardware fuzzers and proposes future research directions.

Open sourcing. ENCARSIA is open source and we have put particular attention in making it user-friendly and well-documented to facilitate its adoption by the community. Additional information can be found at the following URL: https://comsec.ethz.ch/encarsia.

2 Background

We provide the necessary background on hardware design representations (Section 2.1), hardware fuzzing (Section 2.2) and manual bug insertion (Section 2.3).

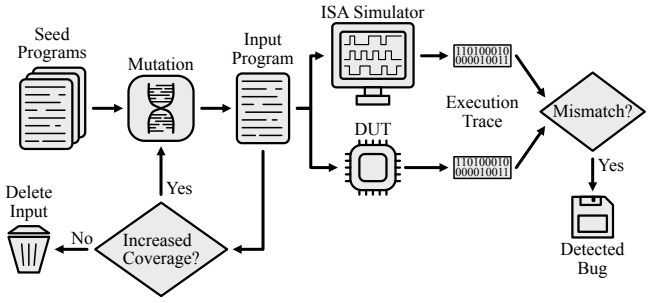

Figure 1: Typical hardware fuzzer workflow.

2.1 Hardware representations

Hardware designs are represented at various levels of abstraction throughout their development. During the design phase, they are typically described using Hardware Description Languages (HDLs) such as Verilog or VHDL, often at a high level of abstraction, like the behavioral or register-transfer level (RTL). HDLs usually provide multiple ways of expressing the same concept for the convenience of the designer. For instance, conditional assignments of signals can be represented using *if* or *case* statements, or ternary operators. Automated tools convert HDLs into an Intermediate Representation (IR) that abstracts away the syntactic sugar [46]. Various processing is then performed on this IR, before converting it back into HDL or another format suitable for the next step in the hardware design flow.

For example, the front-end of the Yosys [46] open-source synthesizer converts HDL designs into RTL Intermediate Language (RTLIL) representations. In RTLIL, Yosys models hardware designs as networks of *macro-cells*, representing functional components such as logic gates or arithmetic units, interconnected by wires. Furthermore, Yosys models Verilog signal assignments [27] as *connections* between a *wire* and it's *driver* signal that supplies it with a logic value. Driver signals can be constants, wires, or combinations thereof. Yosys transforms conditional assignments into multiplexer trees where the assigned signal is connected to the output, the results of each branch are connected to the inputs, and the conditions controlling the assignment are connected to the select signals. Yosys then applies further transformations to the RTLIL representation through procedures known as *passes* (e.g., to implement optimizations).

2.2 Hardware fuzzing

Hardware fuzzers generate randomized inputs in an attempt to trigger and observe bugs in designs under test (Figure 1). They can target CPUs [5, 6, 9, 14, 22, 26, 29, 30, 42, 48] or more generic hardware [32, 45]. CPU fuzzers leverage the well-defined behavior of a CPU, as specified by the ISA, to test meaningful variations of valid instruction sequences, essentially programs, rather than relying on random binary data that rarely triggers CPU functionality. Typically, fuzzers gen-

Table 1: Evaluation methodologies adopted by some recent hardware fuzzers. Cov.: coverage evaluation. Nat.: discovery of natural bugs. Synth.: discovery of synthetic bugs.

Fuzzer	Cov.	Nat.	Synth.
RFUZZ [32]	✓	✗	✗
DifuzzRTL [26]	✓	✓	✓
DirectFuzz [9]	✓	✗	✗
Trippel et al. [45]	✓	✗	✓
TheHuzz [30]	✓	✓	✗
ProcessorFuzz [10]	✓	✓	✗
MorFuzz [48]	✓	✓	✗
Cascade [42]	✓	✓	✗

erate new programs through mutation, such as flipping bits, swapping operands, or rearranging instructions to explore new CPU behaviors. To guide this process, fuzzers use coverage feedback, which tracks the parts of the ISA or CPU exercised by each test. Programs are executed on the CPU, and if they increase coverage, they are retained for further mutation, otherwise, they are discarded. Several coverage metrics have been proposed, such as multiplexer control coverage [32], control register coverage [26], and aggregated simulator-reported coverage [30].

CPU fuzzers detect bugs by comparing a CPU's behavior to a reference model such as an Instruction Set Architecture (ISA) simulator [26, 30, 42]. These fuzzers often advertise the number of new bugs they find in real-world designs to demonstrate their effectiveness [5, 6, 9, 14, 22, 26, 29, 30, 42, 48], yet this metric does not easily lend itself to comparison across different fuzzers due to the diversity of tested hardware designs and versions. Furthermore, bugs become harder to find as designs undergo more testing and become more mature. Table 1 summarizes evaluation methodologies of some recent hardware fuzzers. We distinguish natural bugs found in real-world designs from synthetic bugs that are manually inserted.

2.3 Manual bug insertion

To inject bugs into hardware designs, researchers have been so far relying on manual insertion, in two distinct contexts. The first context is the insertion of specific example bugs to demonstrate the effectiveness of one specific fuzzer at detecting them [24–26, 45]. These ad-hoc bugs are few (not more than 7) and might be biased, given that they are built specifically to illustrate a given fuzzer's effectiveness. In particular, these very few bugs are never later used by others for comparison. The second context is the manual insertion of some dozens of bugs to organize competitions [16]. To inject realistic and diverse bugs, this laborious manual process relied on industry partnerships and strong familiarity with the target design. Consequently, the design used for competitions has been CVA6 for years [49] and these manually-injected bugs are few and generally not disclosed to the public [40].

3 Overview of Challenges

We provide an overview of the challenges that guided the design of ENCARSIA. First of all, it is necessary to understand the properties of natural bugs, since evaluating fuzzers on synthetic bug corpora must reflect the fuzzer's ability to detect bugs that occur in real designs.

CHALLENGE 1.
Understand and unify the characteristics of natural bugs.

In Section 4, we conduct a comprehensive survey of 1672 pull requests across four popular RISC-V CPUs of various complexities and from different maintainers. We find that bugs with highly-complex effects can often be traced back to simple code modifications. We further observe that two types of transformations are sufficient to capture them. The first transformation involves signal mix-ups, such as the use of incorrect operators in expressions or assignments. The second transformation pertains to broken conditionals, which encompass incorrect *if* conditions or the absence of a *default case* in a *switch* statement, among other issues.

The second challenge concerns the translation of these source-level observations to the practical injection of bugs.

CHALLENGE 2.
Automatically and realistically transform RTL designs.

In Section 5, we present ENCARSIA, a fully automatic and easy to use bug injection framework based on the observations made in our survey. We leverage Yosys' intermediate representation (IR) to manipulate wire connections and multiplexer trees that respectively represent expression operands and conditionals. Concretely, ENCARSIA takes an RTL design, injects realistic bugs in the IR provided by Yosys, and outputs the modified design.

The third challenge concerns the quality of injected bugs.

CHALLENGE 3.
Ensure that the injected design transformations are architecturally observable.

In Section 6, we employ formal verification to ensure that the injected transformations have an architecturally observable effect. We provide two setups: one more performance-oriented using the closed source JasperGold and one more accessible to the community using Yosys. Aware of the injected transformations, ENCARSIA adopts a two-step assume-guarantee approach to guide the formal verification process of Yosys. To ensure that one bug will not hide another, we limit the injection to a single bug per design at a time.

Overcoming these challenges, we provide the first open synthetic corpus of CPU bugs in Section 7 and derive new insights about hardware fuzzing in Section 8.

Table 2: Overview of surveyed CPUs, detailing data width (DW), pipeline stages (St), out-of-order execution (OoO) capability, hardware description language (HDL) used, as well as wire and cell counts.

Design	DW	St	OoO	HDL	Wires	Cells
Ibex [41]	32	3	N	SV	21.6k	24.5k
CVA6 [49]	64	6	N	SV	903k	945k
Rocket [2]	64	5	N	Chisel	495k	781k
BOOM [13]	64	10	Y	Chisel	880k	1320k

Table 3: Summary of manually analyzed PRs, with bugs identified as causing software-observable ISA deviations.

	Pull Requests (PRs)			Observable
Design	Pre-filtered	Oldest	Newest	bugs
Ibex	42	PR #31	PR #343	31
CVA6	41	PR #27	PR #225	33
Rocket	50	PR #13	PR #592	49
BOOM	41	PR #7	PR #512	46
Total	174			159

4 Understanding the Causes of Bugs

To understand the characteristics of natural bugs found in real-world CPUs, we survey bugs in four popular open-source designs. We also survey bugs manually crafted for the HACK@EVENT competitions [16]. We focus on bug patches, as they provide root-cause information.

4.1 Collecting natural and synthetic bugs

We describe our methodology for collecting bugs from open-source designs and manually assembled corpora.

4.1.1 Open-source designs

We collect bugs from four open-source CPUs of various complexities and design paradigms (Ibex [33], CVA6 [20], Rocket [1], BOOM [4]) based on their public issue trackers, summarized in Table 2. The brief description of the bugs' symptoms provided in the issue reports is often insufficient to determine root causes and filter out non-observable bugs. Hence, we focus on Pull Requests (PRs) and filter them for actual bug fixes. Using GitHub's REST API [19], we collect the first 100 PRs per design containing the keyword 'fix' and filter them to include only those that modify design source files. We opt for keyword-based filtering over more principled options, such as GitHub labels, due to their inconsistent usage. For instance, only 4 out of 1529 CVA6 PRs are labeled as bugs. Keyword filtering yields a substantial bug dataset and proves unbiased by capturing 20 of 27 labeled bugs in BOOM. Some PRs may still be false positives, e.g., performance improvements, while others address multiple bugs, with up to 65 commits. Therefore, we manually examine the textual report

```
1 assign illegal_csr_insn_o =
2     illegal_csr | illegal_csr_priv;
3 assign csr_we_int  =
4 --  csr_wreq & ~illegal_csr_priv & ...;
5 ++  csr_wreq & ~illegal_csr_insn_o & ...;
```

Listing 1: Example of a signal mix-up in PR #399 of the Ibex CPU. The designer mistakenly uses a partial result of an access check instead of the final result, allowing unauthorized access to debug CSRs.

and the HDL-level fix (Verilog or Chisel) of each commit in the PRs and determine the architecturally-observable bugs. We analyzed a total of 310 code changes. Table 3 indicates the number of pre-filtered PRs, the range of PR numbers (#) and the number of observable bugs.

Manually assembled corpora. For evaluating verification methodologies Dessouky et al. [16] manually compile a set of hardware bugs, which they claim to be uniquely challenging for state-of-the-art verification techniques. While the original corpus presented in the paper provided only brief descriptions of the bugs, later iterations of the competition included some bug fixes. The HACK@DAC 2019 corpus featured 4 such bug fixes, while the HACK@DAC 2021 corpus expanded this by 26. To ensure that ENCARSIA is generic enough to also model these selected bugs, we include them in our survey.

4.2 Bug classification

Our goal is to identify recurring structural patterns among the bugs, enabling their automatic injection through simple circuit transformations. In an initial exploratory phase of our survey, we observed that bugs affect a wide range of components and HDL constructs. Yet, despite this diversity, we found that the affected HDL constructs fall into two groups: wrong logic in signal drivers and mistakes in conditionals. This led us to define the following categories.

Signal mix-ups. Signal mix-ups encompass confusion of signals, functions, or other elements in assignments or expressions. They usually arise from human errors due to signals having similar names, types, or purpose. An example of a signal mix-up observed during the initial exploratory phase (Ibex PR #399) is shown in Listing 1. In this example, the designers mistakenly used a partial result of an access check instead of the final result, allowing unauthorized access to the debug CSRs.

Broken conditionals. Broken conditionals refer to incorrect conditional statements, for example mishandling of exceptional cases like special values in floating-point arithmetic, or signal updates in wrong design states. These usually arise from designers either forgetting an exceptional case or specifying wrong conditional expressions, or generally making algorithmic mistakes. A simplified example of a broken conditional observed during the initial exploratory phase (Ibex PR #277) is shown in Listing 2. In this example, the designers

```
  always_comb begin
    csr_rdata_int = '0;
    illegal_csr   = 1'b0;
    unique case (csr_addr_i)
      CSR_MSTATUS: csr_rdata_int = mstatus_q
      CSR_DCSR: begin
        csr_rdata_int = dcsr_q;
++      illegal_csr = ~debug_mode_i;
      end
      default: begin
        illegal_csr = 1'b1;
      end
    endcase
  end
```

Listing 2: Example of a broken conditional derived from pull request #277 of the Ibex CPU. The designer forgot to include a check to ensure the CPU is in debug mode upon access to debug CSRs, which allows unintended access.

forgot a check to ensure that the CPU is in debug mode upon access to debug CSRs, which allows users unintended access.

4.2.1 Classification

We manually classify each bug fix identified in our survey into the two categories: *signal mix-ups* or *broken conditionals*. If they contain multiple modifications, including both types, we classify them as both. Listing 3, which resembles an excerpt from a simplified cipher block chaining cryptographic module, illustrates our classification approach. Any erroneous assignment of a signal, function, or other entity in assignments or expressions (outside conditions) is classified as a signal mix-up. Bug M1 in Listing 3 is a name mix-up in a continuous assignment, where one of the operands of its assigned logic expression is incorrectly chosen (like Rocket PR #349). Bug M2 shows a more complex case of a mix-up of a whole logic expression (state == ENCRYPT && enable_i) with a single signal (enable_i) (like BOOM PR #437). Bug M3 shows a mix-up in a clocked assignment and within a conditional statement (like Ibex PR #286).

We classify bugs affecting conditional assignments as broken conditionals when they involve missing cases or incorrect conditions. Bug C1 of Listing 3 is an example where a nested conditional is forgotten (like CVA6 PR #138). This implies that the condition for the resulting assignment, in this case of `next_state`, is too loose. Similar bugs might, for example, create access control vulnerabilities. Bug C2 exemplifies a missing case in a switch statement (like BOOM PR #173). This bug exemplifies forgetting a certain edge case such as a reset for a register, a default case, or an overflow check.

Some bugs, like bug M4, are signal mix-ups in the body of a conditional statement (like Ibex PR #169). To avoid overlaps between the two categories, we classify bugs that concern solely the conditionally assigned values, but do not modify any conditions, as signal mix-ups.

```
module cbc (
  input  logic clk,
  input  logic reset,
  input  logic valid_i,
  input  logic enable_i,
  input  block_t plaintx_i,
  output block_t ciphertx_o
);
  state_t state, next_state;
  block_t prev_cipher_blk;

  // bug M1: wrong continuous operand choice
--assign cipher = plaintx_i ^ state;
++assign cipher = plaintx_i ^ prev_cipher_blk;

  // bug M2: forgotten operand
  assign cipher_valid = enable_i;
++assign cipher_valid = enable_i &&
++                      (state == ENCRYPT);

  // States
  always_ff @(posedge clk, posedge reset) begin
    if (reset) begin
      state <= IDLE;
      prev_cipher_blk <= '0;
    end else begin
      state <= next_state;
      // bug M3: wrong register assignment
--    prev_cipher_blk <= plaintx_i;
++    prev_cipher_blk <= cipher;

      ciphertx_o <= cipher_valid ? cipher : '0;
      ...
    end
  end

  // Next state logic
  always_comb begin
    next_state = state;
    case (state)
      IDLE: begin
        // bug C1: forgotten conditional
++      if (valid_i) begin
          next_state = ENCRYPT;
++      end
      end
      ENCRYPT:
        // bug M4: wrong state transition
--      next_state = IDLE;
++      next_state = ENCRYPT_FINAL;
      // bug C2: missing case
++    ENCRYPT_FINAL: begin
++      if (!valid_i) begin
++        next_state = IDLE;
++      end
++    end
    endcase
  end
  ...
endmodule
```

Listing 3: Simplified cipher design with six bugs. Bugs M1-M4 are mix-ups, corresponding to wrong operand choices, with different contexts and sensitivities (continuous or clocked assignment). Bugs C1-C2 are broken conditionals.

Table 4: Number of bug fixes per design and category.

Source	Conditionals	Mix-ups	Total
Ibex [41]	17	14	31
CVA6 [49]	22	11	33
Rocket [2]	8	41	49
BOOM [13]	13	33	46
HACK@DAC19	2	2	4
HACK@DAC21	4	10	14
Total	66	111	177

Results. Table 4 summarizes the results of our survey, confirming that all identified observable bugs indeed do fall into one of the two categories. Of the 177 relevant bugs found in our survey, 111 (63%) were signal mix-ups, 66 (37%) were broken conditionals. Appendix A provides some examples per category and CPU.

Security implications. We further studied whether the bugs could have security implications by comparing its effects to those of hardware bugs that have previously been assigned CVE numbers [10, 12, 26, 42, 48]. In conclusion, all architecturally observable bugs could impact security in one way or another (with different severities), because attackers may deliberately create conditions where the bug leads to an attacker-chosen architectural effect.

5 Transforming Hardware

Given our newfound knowledge of bugs, we proceed to the design of ENCARSIA. We implement ENCARSIA as a series of netlist transformation passes in the Yosys Open SYnthesis Suite [46]. We start by injecting signal mix-ups through swaps on signal connections (Section 5.1), and then reproduce broken conditionals by corrupting multiplexer trees (Section 5.2). For each bug category, we describe the affected hardware structures in the intermediate representation of Yosys and how to model the bugs within this representation.

5.1 Signal mix-ups

As discussed in Section 4, signal mix-ups are erroneous uses of signals, functions, or other entities in assignments or expressions. We now explain how ENCARSIA injects signal mix-up transformations in assignments and expressions.

Assignments. In the intermediate representation of Yosys, assignments, known as *connections*, are pairs of *wires* with their *driver* signals. For instance, the Verilog statement `assign sig_a = sig_b;` is represented as `(sig_a, sig_b)`. We inject signal mix-ups by replacing driver signals (the right-hand side of assignments) with other random signals present in the same RTL module. This resembles replacing a driver with the logic expression, constant or signal that is assigned to the new driver and therefore models arbitrary logic bugs. We only replace drivers to avoid introducing statically detectable bugs such as multi-driven wires. ENCARSIA randomly selects two such connections and replaces the driver of the first connection with that of the second. When the newly connected signals differ in width, ENCARSIA truncates the wider signal to match the width of the narrower one. If the new driver is a constant smaller than the driven wire, we sign-extend it. This is consistent with the implicit Verilog behavior for mismatching signal widths [27].

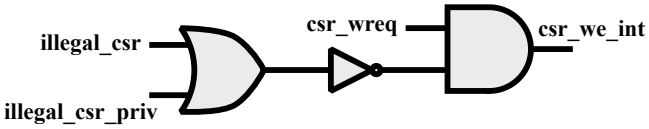

Figure 2: Intermediate representation of logical expressions in Yosys, as derived from Listing 1.

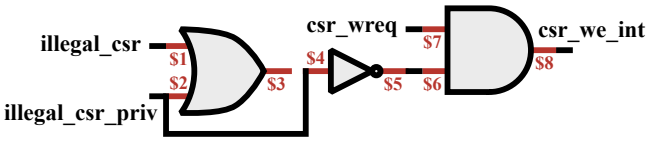

Figure 3: Injecting a signal mix-up into a logical expression (Figure 2). Intermediate wires are shown in red. After injection, `csr_we_int` will be erroneously high if `csr_wreq` and `illegal_csr` are both high.

Logical expressions. Expressions are translated into a series of logical cells interconnected by signals (Figure 2). Mix-ups in expression operands therefore correspond to a wrong signal being connected to a cell port. Each cell keeps its own internal dictionary that maps ports to the connected signals. To unify the injection of mix-ups in simple assignments and logic expressions, we insert an intermediate wire between each cell port and the signal originally connected to it (Figure 3). This way, injecting a mix-up at the intermediate wire is equivalent to injecting a mix-up at the cell port. Intermediate wires are later cleaned up using a standard optimization pass.

5.2 Broken conditionals

Broken conditionals refer to issues arising from improper handling of exceptional cases or, e.g., operations allowed only in a specific privilege mode. Yosys represents conditionals as multiplexer trees, a hierarchical arrangement of multiplexers where each level progressively narrows down the set of possible expressions assigned to the signal at the root of the tree (Figure 4). At each level of the multiplexer tree, the select signals, derived from the conditions of the assignment, determine the appropriate branch to follow, ultimately selecting the correct expression to assign. In this representation, broken conditionals correspond to a potential branch missing from the multiplexer tree, or an incorrectly derived select signal that

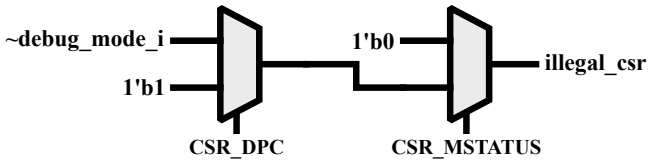

Figure 4: Intermediate representation of conditional assignments in Yosys, as derived from Listing 2. CSR_DPC and CSR_MSTATUS are select signals, illegal_csr is the root of the tree.

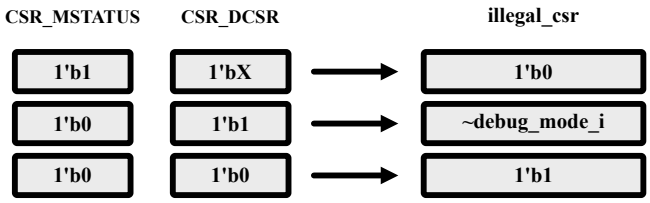

Figure 5: ENCARSIA's table representation of conditional assignments. The columns on the left of the arrow represent select signal combinations, the column on the right contains the expression that is selected for the multiplexer tree root.

does not match the condition of the assignment. Hence, this abstraction allows handling complex conditional assignments with a unified injection approach.

Table representation. ENCARSIA abstracts these multiplexer trees into a table representation (Figure 5). The table maps specific select signal combinations to the corresponding expression chosen in that case. This abstract representation allows ENCARSIA to inject realistic complex transformations into conditional assignments by performing simple operations on the table, such as removing or adding cases, thus avoiding the need to modify the circuit following each operation.

Table construction. ENCARSIA iterates over all wires in the circuit to identify multiplexer trees. It marks as multiplexer roots those wires that are driven by a multiplexer but do not drive any other multiplexer. These wires correspond to signals that are conditionally assigned with different expressions.

To construct the table, ENCARSIA performs a depth-first search of the multiplexer tree, starting at the root. During this traversal, ENCARSIA records the select signal values required to follow a given path. Every time ENCARSIA reaches an input signal that is not itself an output of a multiplexer, it discovers a new potential expression to be assigned to the root. Thus, it adds a new row to the table with the discovered input signal and the recorded select signal values, expanding the table's columns as necessary and filling existing rows with X (i.e., "don't care" value) for the newly added columns. This process is repeated for each path in the multiplexer tree, resulting in a table that contains all possible conditionally chosen expressions (right most column in Figure 5) with their specific select signal combinations (left columns in Figure 5).

Table transformations. Transformations of the tables can represent realistic subtle mistakes that designers make, according to our survey. Removing a single row from the table can model a designer forgetting to consider a special case. Turning a select signal into an X (known as "don't care" value) can model cases where designers place insufficient constraints on a case. Conversely, turning a "don't care" select signal into a specific value can model bugs where constraints are placed on cases mistakenly. These transformations cover the entirety of the bugs in conditional logic that we revealed in the survey.

Transformed circuit construction. Once manipulated, the tables must be translated from their abstract representation back into standard circuitry. This allows the circuit to be simulated, instrumented, or otherwise analyzed by fuzzers (or other testing tools). We achieve this by translating the table into a Yosys internal *pmux* cell, which is a many-input multiplexer that utilizes one-hot encoding (where only one bit is high per possible select value) [27]. Each row in the table represents a possible case in the conditional assignment. We want an entry's assignment expression to propagate exactly when the select signal matches the corresponding select values in the table. Thus, we connect the expression to one of the inputs of the *pmux*. The corresponding one-hot select bit is created by adding an *eq* cell that compares the select signal combination from the table to the actual select signal value. The output of the *pmux* cell is then connected to the root wire that was originally driven by the multiplexer tree. The Yosys backend for translating RTLIL back into Verilog then translates the `pmux` cell into a process resembling the original conditional statement.

6 Verification of Observability

The transformations outlined in Section 5 do not necessarily introduce observable bugs, as they do not consider the higher-level semantics of the modified circuitry. Consider a CPU employing triple modular redundancy to safeguard against single event upsets. A transformation that corrupts the majority voting mechanism to select the minority value instead of the majority is undetectable by current CPU fuzzers. This is because they do not model single event upsets in the simulation, so redundant circuits always produce identical outputs and reach the correct consensus. Including these transformations in our evaluation corpus would skew the results by making fuzzers search for innocuous transformations.

Miter circuits. To address this challenge, we leverage miter circuits [3], which provide the original and transformed Design Under Test (DUT) with the same inputs at all times, and monitor for differences in the observable outputs, e.g., the architectural registers. Given that the two versions of the DUT only differ in the transformation performed by ENCARSIA, any difference in the observable elements proves that this transformation has an observable effect. This approach allows

```
property propagated;
  host_observables != reference_observables;
endproperty
c_propagated: cover property (propagated);

p_boot: assume property (in_boot_addr_i == 0);
p_hart: assume property (in_hart_id_i == 0);
```

Listing 4: SVAs for proving the observability of injected bugs.

us to define a bug observation in a more formal and precise manner using SystemVerilog Assertions (SVAs), specifically as discrepancies in the values of visible elements between the two DUTs (`c_propagated` in Listing 4). Formal tools like JasperGold [8] can effectively explore all possible states and transitions to find cover traces or disprove the coverability of such design properties.

Two-step proof approach. Model checking is expensive in terms of computations and memory. Its performance can be boosted by providing invariants [11, 17] or partitioning the problem using assume-guarantee reasoning [15]. Generally, automatically deriving invariants is hard [17, 47]. However, when transforming the CPU, ENCARSIA is aware of one key information: the trigger. For example, in a multiplexer tree transformation, we know that the transformation can only affect the design's output once the modified rows of the corresponding table are selected. Hence, to reduce the search space for the formal verifier, ENCARSIA adopts a two-step assume-guarantee approach. First, we instruct the SAT solver to find inputs that satisfy the trigger condition, using a property similar to `c_propagated` but on the mixed-up signal instead of the architecturally visible signal. Second, if the first step succeeded, we constrain the SAT solver to preserve the triggering sequence and instruct it to propagate the discrepancy to a user-defined list of architecturally-visible signals.

Formal setup. To limit the search space and avoid false positives due to uninitialized states, we constrain the formal setup to resemble the initialization done by fuzzers. We initially reset the CPUs and configure several critical control and status registers to predefined values. For example, we set the FS field of the `mstatus` register to 1 to enable floating-point instructions. We model these configurations with initial value abstractions and constraints for the first clock cycle to accelerate verification. We also seek to prevent the verifier from using debug or testing features that are typically encapsulated within an SoC and therefore inaccessible to fuzzers. To this end, we restrict the considered input behavior using SVA assume statements. For example, we limit Ibex to a single hardware thread (`p_hart` in Listing 4).

Formal verification. For verification, we instruct Jasper to cover the aforementioned `c_propagated` property, which proves that the transformation can provoke an architecturally observable deviation from the original design. On the other hand, if the property is not covered, the transformation is deemed ineffective and excluded from the evaluation.

Table 5: Number of transformations (#Transf.) and average time required for a transformation (Avg. T.).

Design	Mix-ups		Conditionals	
	#Transf.	Avg. T.	#Transf.	Avg. T.
Ibex [41]	1210	213 ms	1111	88 ms
Rocket [2]	931	134 ms	718	65 ms
BOOM [13]	1230	886 ms	982	407 ms

7 Evaluation

We first generate a fuzzer evaluation corpus, EnCorpus and evaluate ENCARSIA in terms of injection and verification performance (Section 7.1). Next, we detail our methodology for evaluating fuzzers with EnCorpus and validate it using an example fuzzer and target design [41] (Section 7.2). We then evaluate three open-source hardware fuzzers using EnCorpus and conclude with an extensive discussion of the insights gained from this evaluation and concrete recommendations for future research into hardware fuzzing (Section 8).

Evaluation setting. We evaluate ENCARSIA on a machine equipped with two AMD EPYC 7H12 processors, each running at 2.6 GHz (256 logical cores in total) and 1 TB of DRAM. The injection is implemented as a series of passes in the Yosys Open SYnthesis Suite [46] (v0.37, commit `a5c7f69e`). For verification, we utilized JasperGold [8] (version Jasper Apps 2022.09p001, 64-bit) running its four default engines in parallel across four cores. Time measurements obtained from parallel testing are reported as the aggregate of the durations recorded by each individual core.

We evaluated DifuzzRTL [26] (commit `d2dc9f82`), Processorfuzz [10] (commit `de6b7ef5`) and Cascade [42] (commit `e916c7e0`) with their default parameters. We chose these three fuzzers as they are among the few open-source fuzzers. Additionally, they represent different approaches to CPU fuzzing: DifuzzRTL and ProcessFuzz are guided by distinct coverage metrics, while Cascade is black-box. They also have different algorithms for generating instructions sequences. We intend to uncover the impact of these design decisions.

7.1 EnCorpus

We evaluate ENCARSIA by generating EnCorpus, a unified corpus of 90 bugs for evaluating hardware fuzzers.

Injection. We measure the performance of ENCARSIA by injecting bugs into Ibex, Rocket, and BOOM. We injected approximately 1'000 transformations per design and category and summarize the results in Table 5. We observe that even for a complex design like BOOM, ENCARSIA requires on average less than a second for performing each transformation.

Another important aspect of the injection process is the diversity of transformations that can be applied to the design. We have identified in Section 4 that bugs can be modeled by

Table 6: Verification performance of ENCARSIA. We report the number of transformations that we tried to verify (#Transf.), the verification success rate (Succ. %), and the average time taken to verify a single bug (Avg. T.).

Design	Mix-ups			Conditionals		
	#Transf.	Succ. %	Avg. T.	#Transf.	Succ. %	Avg. T.
Ibex [41]	116	45 %	211 s	1065	4.5 %	116 s
Rocket [2]	242	44 %	194 s	396	8.1 %	117 s
BOOM [13]	276	24 %	258 s	982	2.3 %	195 s

two types of transformations: signal mix-ups and broken conditionals. We injected 3371 signal mix-ups that impact 2482 signals spread across 121 distinct modules. We injected 2811 broken conditionals that impact 202 multiplexer trees spread across 37 distinct modules. This underlines the diversity of locations that can be affected by ENCARSIA.

Verification. We evaluate the verification performance by measuring both the failure rate and the time required to verify observability. We run JasperGold for the transformations and set a timeout of 40 core minutes. The verification is considered successful if JasperGold successfully covers the property shown in Listing 4. The non-successful proofs timed out.

Table 6 summarizes the results. The percentage of mix-ups successfully verified by Jasper lies between 23% and 45% and decreases with increasing design complexity. The percentage of verified conditionals is consistently low across the designs, i.e., between 2.3% and 8.1%. While a larger time-out may increase this number, we trade-off for overall verification time, since, as we will see in the following, the verified bugs are diverse enough to evaluate differences between fuzzers.

7.2 Fuzzer evaluation methodology

We present our fuzzer evaluation methodology on bugs injected by ENCARSIA, followed by a case study on Ibex [41], only supported by Cascade.

Attribution. A key challenge in evaluating fuzzers is attributing deviations from the expected behaviour to the specific bugs responsible for them [31]. When a single designs is affected by multiple bugs without an accurate attribution, a fuzzer that repeatedly triggers the same bug might appear more effective than another fuzzer that identifies multiple distinct bugs within the same design. Although various automated techniques for crashing input deduplication have been proposed in software, they have generally proven to be ineffective [31]. Additionally, no such methods have been proposed for hardware. As a result, recent studies on hardware fuzzing have relied on manual examination of flagged traces [10, 26, 30, 42, 48]. To eliminate this labor-intensive process, we inject one bug per design at a time.

Uninterrupted fuzzing. To provide a measure of difficulty for a fuzzer to detect a bug, we do not interrupt the fuzzing campaign once a bug is detected. This allows us to measure the number of times a bug is detected by a fuzzer, i.e., the number of fuzzer-provided inputs that trigger the bug.

Table 7: Cascade's bug detection performance on Ibex. The table shows whether each bug was detected (✓) or not (✗).

Bug	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
Mix-ups	✓	✗	✗	✓	✓	✗	✗	✗	
Cond.	✗	✗	✗	✗	✗	✗	✗	✗	
Bug	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**Total**	
Mix-ups	✓	✓	✓	✗	✓	✓	✓	9/15	
Cond.	✓	✗	✗	✗	✗	✗	✗	1/15	

Eliminating false positives. The underlying designs might be affected by natural bugs. To avoid imprecisions in the survey, we filter them out by discarding fuzzing inputs that signal a bug in both the non-injected and the injected version. In total, 1% of inputs produced false positives for DifuzzRTL and were hence filtered out.

Results on Ibex. We first apply our evaluation methodology on Cascade using 15 (verified) bugs per category injected into Ibex. Each bug was tested by Cascade for the duration of 24 hours on a single core. The results of the evaluation are presented in Table 7. Cascade successfully identified 9 out of the 15 signal mix-ups and 1 out of the 15 broken conditionals.

Some bugs were harder to detect than others. Over 24 hours, the easiest bugs were detected by 99 % of the programs, while the hardest by only 0.02 % (6 out of 30'239 programs). This matches the results of the original Cascade evaluation, where most bugs were found in the first 10 core minutes, while some required 17 core hours [42]. This corroborates the representativity of the bugs injected by ENCARSIA. If Cascade was 10x slower, it could have missed some bugs over 24 hours. Does this imply that higher performance would lead to detection of even more bugs? In Section 8, we evaluate factors that affect fuzzers' bug detection ability.

8 Insights into Hardware Fuzzing

We explore various aspects of hardware fuzzing to understand the key factors that determine whether a bug will be detected.

Table 8: Bugs detected by DifuzzRTL (DF) and ProcessorFuzz (PF) with coverage disabled.

Bug	Rocket				BOOM			
	Mix-ups		Cond.		Mix-ups		Cond.	
	DF	PF	DF	PF	DF	PF	DF	PF
1	✗	✗	✗	✗	✓	✓	✓	✓
2	✓	✓	✗	✗	✓	✓	✓	✓
3	✓	✓	✗	✗	✗	✗	✗	✗
4	✓	✓	✗	✗	✓	✓	✗	✗
5	✗	✗	✗	✗	✓	✓	✗	✗
6	✗	✗	✓	✓	✗	✗	✗	✗
7	✗	✗	✓	✓	✗	✗	✓	✓
8	✗	✗	✗	✗	✓	✓	✗	✗
9	✗	✗	✗	✗	✗	✗	✗	✗
10	✗	✗	✓	✓	✓	✓	✗	✗
11	✗	✗	✗	✗	✓	✓	✓	✓
12	✗	✗	✓	✓	✓	✓	✓	✓
13	✓	✓	✗	✗	✗	✗	✗	✗
14	✗	✗	✗	✗	✗	✗	✗	✗
15	✓	✓	✓	✓	✓	✓	✓	✓
Tot.	5	5	5	5	9	9	6	6

Table 9: Bugs detected by DifuzzRTL (DF) and ProcessorFuzz (PF) with coverage enabled.

Bug	Rocket				BOOM			
	Mix-ups		Cond.		Mix-ups		Cond.	
	DF	PF	DF	PF	DF	PF	DF	PF
1	✗	✗	✗	✗	✓	✓	✓	✓
2	✓	✓	✗	✗	✓	✓	✓	✓
3	✓	✓	✗	✗	✗	✗	✗	✗
4	✓	✓	✗	✗	✓	✓	✗	✗
5	✗	✗	✗	✗	✓	✓	✗	✗
6	✗	✗	✓	✓	✗	✗	✗	✗
7	✗	✗	✓	✓	✗	✗	✓	✓
8	✗	✗	✗	✗	✓	✓	✗	✗
9	✗	✗	✗	✗	✗	✗	✗	✗
10	✗	✗	✓	✓	✓	✓	✗	✗
11	✗	✗	✗	✗	✓	✓	✓	✓
12	✗	✗	✓	✓	✓	✓	✓	✓
13	✓	✓	✗	✗	✗	✗	✗	✗
14	✗	✗	✗	✗	✗	✗	✗	✗
15	✓	✓	✓	✓	✓	✓	✓	✓
Tot.	5	5	5	5	9	9	6	6

8.1 Granularity of differential fuzzing

Differential fuzzers such as DifuzzRTL [26], ProcessorFuzz [10], TheHuzz [30] and HyPFuzz [14] adopt various policies to detect divergences between the design under test and a golden model. We do not have access to TheHuzz and HyPFuzz, and the authors did not respond to our questions. We are therefore constrained to exclude these two fuzzers from our evaluation. DifuzzRTL and ProcessorFuzz are built on a similar instruction generation infrastructure and thus allow for a fair comparison. DifuzzRTL places a read function at the end of each program that reads the register file, CSRs and some other architecturally visible elements to be used for the comparison. ProcessorFuzz continuously logs the values of several manually defined internal signals, with the intuition that a bug symptom might later be shadowed. We evaluate detection granularity on Rocket and BOOM, which are the only designs supported by both fuzzers. We will evaluate the fuzzers on 15 bugs per category on each design, for 24 hours each, on a single core.

Results. We evaluate DifuzzRTL and ProcessorFuzz on the aforementioned corpus of bugs with coverage initially disabled to prevent any coverage-induced bias. Table 8 summarizes the results of the evaluation and shows that both fuzzers found exactly the same set of bugs (5/15 mix-ups, 5/15 conditionals). This leads us to the first insight of this evaluation:

Insight 1. Instruction-granular bug detection mechanisms do not demonstrate greater potential for detecting bugs.

8.2 Coverage metrics

Recently, several papers proposed using hardware-specific coverage metrics to guide fuzzers towards uncovering new bugs. The generic hardware fuzzer RFUZZ [32] observes *multiplexer select coverage* to guide the fuzzer to explore new paths through the design. DifuzzRTL [26] found that multiplexer select coverage is flawed because multiplexers are not clock sensitive, resulting in the capture of irrelevant intermediate signals. Hence, DifuzzRTL relies on *control register coverage*, which tracks the states of clocked registers driving the select signals. ProcessorFuzz [10] suggests that much of these registers are datapath-related and that the state of the CPU is better captured by monitoring CPU specific control and status registers (CSRs). The other family of coverage-guided CPU fuzzers [14, 30], which are not open-source, relies on a sum of simulator-provided coverage metrics. In this experiment, we evaluate whether the two state-of-the-art coverage metrics introduced by DifuzzRTL and ProcessorFuzz effectively help fuzzers discover new bugs.

Results. We evaluate DifuzzRTL and ProcessorFuzz on the same set of bugs, this time with coverage enabled. We do not evaluate RFUZZ, which is by construction not able to detect bugs [32]. Table 9 shows that both fuzzers performed equally well, detecting exactly the same set of bugs as when coverage is disabled (Table 8). This leads us to the second insight:

Insight 2. The hardware-specific structural coverage metrics, advertised as central by many fuzzers, are of little help in detecting bugs.

Table 10: Bugs detected by DifuzzRTL (DF) with coverage enabled and Cascade (CA). The fuzzers identify different bugs, indicating that seed programs influence RTL bug detection.

Bug	Rocket				BOOM			
	Mix-ups		Cond.		Mix-ups		Cond.	
	DF	CA	DF	CA	DF	CA	DF	CA
1	✗	✓	✗	✗	✓	✓	✓	✓
2	✓	✓	✗	✓	✓	✓	✓	✓
3	✓	✓	✗	✗	✗	✗	✗	✗
4	✓	✓	✗	✗	✓	✗	✗	✗
5	✗	✗	✗	✗	✓	✗	✗	✗
6	✗	✓	✓	✓	✗	✓	✗	✗
7	✗	✗	✓	✗	✗	✓	✓	✓
8	✗	✗	✗	✗	✓	✓	✗	✗
9	✗	✗	✗	✗	✗	✓	✗	✗
10	✗	✗	✓	✗	✓	✓	✗	✗
11	✗	✗	✗	✗	✓	✓	✓	✓
12	✗	✗	✓	✓	✓	✗	✓	✓
13	✓	✓	✗	✗	✗	✗	✗	✗
14	✗	✗	✗	✗	✗	✓	✗	✗
15	✓	✓	✓	✓	✓	✓	✓	✗
Tot.	5	7	5	4	9	10	6	5

8.3 Importance of the seeds

DifuzzRTL and ProcessorFuzz adopt different coverage feedback mechanisms, but they rely on the same seed programs [10]. On the other hand, Cascade is a fuzzer that builds programs explicitly from the bottom up and claims to generate valid complex programs [42]. Hence, Cascade can be seen as a seed generator, with seeds distinct from those of DifuzzRTL and ProcessorFuzz. In this experiment, we evaluate whether differences in seed programs affect bug detection ability.

Results. We compare Cascade and DifuzzRTL on the previous set of bugs. Results, summarized in Table 10, show that different seed programs result in the discovery of different bugs per design, as opposed to the feedback mechanism, which corroborates our finding from Section 8.2. While both fuzzers exhibit comparable performance on Rocket (Cascade: 11/30, DifuzzRTL: 10/30) and BOOM (Cascade: 15/30, DifuzzRTL: 15/30), there is a notable difference in the bugs that they detect. Cascade detects 6 bugs that DifuzzRTL does not, while DifuzzRTL detects 5 bugs that Cascade does not.

Insight 3. The fuzzer seeds are a key factor that determines which bugs will eventually be detected.

Figure 6 summarizes the time to bug detection for the bugs that were detected by all fuzzers. On these bugs, Cascade, which operates in a black-box manner, is at least one order of magnitude faster than DifuzzzRTL and ProcessorFuzz when

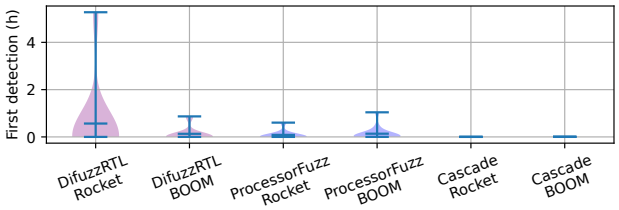

Figure 6: Time to bug detection for bugs that were detected by all fuzzers. The horizontal bars indicate the average.

it comes to detecting the same bugs. This corroborates that not only structural coverage does not appear to help finding bugs, but also that coverage feedback strategies tend to slow down bug discovery. Given that fuzzing campaigns are usually time-bounded, this can ultimately lead to missed bugs.

Insight 4. Fuzzing that relies on state-of-the-art structural coverage feedback is slower at finding the same bugs as black-box fuzzing.

8.4 Case studies

We closely examine a selection of the injected bugs to gain further insights into hardware fuzzing.

BOOM mix-up 9. In the CVA6 PR #27, the multiplier was not flushed due to missing handshake signals. Similarly, the signal mix-up 9 injected by ENCARSIA into BOOM affects the multiplier handshake signals, causing it to drop multiplication requests. This bug only requires the fuzzer to issue a multiplication instruction to trigger, hence is relatively easily detectable. Nevertheless, DifuzzRTL fails to detect this bug due to an error that prevents it from using instructions in the RISC-V M extension when testing CPUs implementing the RISC-V G subset (extensions I, M, A, F, and D). While fuzzers detect many bugs in the components that they test, most of the undetected bugs reside in components completely untested by the fuzzers, like e.g. in the handling of compressed instructions (like Ibex PR #48).

Insight 5. Fuzzers often do not test the entire ISA.

BOOM mix-up 3. This mix-up introduced a bug that only gets triggered when the result of a floating-point division equals zero. Cascade does not detect it because it excludes f.div/f.sqrt instructions for some designs to filter out known unfixed natural bugs to avoid re-detections. While such filtering is desirable to continue the fuzzing campaign until the bug has been fixed by the designers, it highlights that the filtering must be more targeted towards the specific bug, rather than excluding an instruction type [12]. A natural example that might be filtered out is Rocket PR #574. While DifuzzRTL and ProcessorFuzz test these instructions, they do not seem to provide sufficiently interesting input values.

ProcessorFuzz introduces an instruction-granular bug detection mechanism over DifuzzRTL. It then builds known bug filtering on top of this new mechanism, ignoring bugs previously recognized in the design. However, this system is exclusively integrated within the new detection mechanism. Since the original detection mechanism remains active and unmodified by the authors, it is possible for a bug to be identified by the new system, categorized as a known bug, and then rediscovered by the old system, resulting in a false positive. These false positives often terminate test executions early and therefore might mask new bugs.

Insight 6. Current filtering mechanisms for known bugs broadly exclude test scenarios, hindering the discovery of further bugs.

Rocket conditional 7. The CSRRS instruction in RISC-V reads a CSR, performs a bitwise OR operation with the value stored in the source register, and writes the result back to the CSR. However, it includes a special case where, if the source register is the zero register, the instruction should avoid any side effects of writing to the CSR. Broken conditional 7 in Rocket removes this special case for the FCSR register (like Ibex PR #53). While Cascade does access FCSR, it only does so via the CSRRW instruction, which includes no such special case. As a result, Cascade fails to detect this bug.

Insight 7. Values seemingly coming from the same source according to the ISA can be produced by different components. Fuzzers often fail to cover these various cases.

BOOM mix-up 13. Signal mix-up 13 injected by ENCARSIA into BOOM affects issue scheduling logic. However, due to this injected bug, if an issue queue is full and the youngest instruction is a variable-latency instruction, then a following instruction dependent on the result of the variable-latency instruction might be scheduled before it. Therefore, following instructions read stale values. Similarly, conditional 3 only gets triggered by sequences of independent floating-point instructions that fill up a specific buffer. None of the three fuzzers under study detected these bugs. PR #289 and #295 in BOOM fixed bugs that only appeared when the reorder buffer was full. This highlights the importance of testing all possible microarchitectural paths through the design.

Insight 8. Fuzzers struggle to detect subtle bugs that arise from corner cases in microarchitectural structures.

Interestingly, this case study also exposes strong similarities between ENCARSIA-injected bugs and natural bugs.

8.5 Recommendations for fuzzing

Our findings demonstrate the need for more principled approaches in the development of hardware fuzzers. We provide guidelines to help shape future hardware fuzzers.

Structural coverage: Coverage-guided fuzzers must rigorously evaluate the benefits of the underlying coverage metric in discovering bugs.

Most coverage-guided fuzzer proposals do not evaluate the effectiveness of their coverage metric in detecting new bugs [14, 30, 48]. This impedes distinguishing the effectiveness of the coverage metric from the seeds and instruction generation scheme, which we have shown to be a key component. ENCARSIA will provide a valuable tool for evaluating this effectiveness by correlating the use of the feedback from a given coverage metric with the actual bugs found.

ISA-based functional coverage: Fuzzers must maximize ISA-based functional coverage.

In Section 8.3, we have shown that the quality of the seed programs is a key factor in the effectiveness of a fuzzer. In Section 8.4, we have shown that bugs are often missed because the fuzzer does not test the corresponding design functionality. Hence, maximizing ISA-based functional coverage [27] of the seed programs to test all instruction variations with various input values is a crucial prerequisite for effective fuzzing. If fuzzers filter for known bugs, they need to restrict the filtering to specific scenarios only, rather than excluding instructions, as this might mask unknown bugs. Unfortunately, many fuzzers [10, 26, 30, 32] put a strong emphasis on purely structural coverage feedback to the detriment of taking the functional coverage into account.

Microarchitectural coverage: Fuzzers might consider feedback from microarchitectural components.

In Section 8.4, we have shown that bugs can arise from corner cases in some microarchitectural structures. The existing coverage-guided fuzzers consider low-level structural coverage, often related to multiplexer signals [26, 32, 48] or toggle coverage [14, 30]. There is no evidence that coverage feedback from such low-level metrics will translate into covering these interesting microarchitectural corner cases. One natural way to improve bug detection might be to consider functional coverage feedback from microarchitectural components, such as the issue queue, the reorder buffer, or the branch predictor, to guide the fuzzer towards corresponding corner cases. Hence, an interesting future work direction is to define a generic coverage metric that takes these common microarchitectural components into account.

Black-box microarchitectural coverage model: Synthetic bugs can aid in developing generic functional coverage models of microarchitectural elements.

Black-box CPU fuzzing is on the rise because some CPUs are closed-source [23, 36, 37, 44] and for performance reasons [42]. Black-box fuzzers cannot benefit from direct feedback from microarchitectural components. However, they can use black-box functional coverage models [36] described as traces that are expected to exercise CPU internal corner cases. We envision that a large-scale learning campaign based on ENCARSIA-injected bugs and their verification traces could generate a black-box functional microarchitectural coverage model that could guide black-box fuzzers.

Breadth: Combining multiple state-of-the-art fuzzers leads to a broader range of tested inputs.

While Cascade claims to be superior to DifuzzRTL in terms of coverage and new bugs found, ENCARSIA clearly shows that DifuzzRTL can find bugs that Cascade cannot. A simple, considerable improvement is to build a mixed fuzzer that spends 50% of its time running Cascade and 50% running DifuzzRTL. This mixed fuzzer indeed finds 21/30 bugs of the study, 6 more than Cascade and 5 more than DifuzzRTL. Importantly, traditional evaluations in terms of coverage and new bugs found would not have revealed this, and Cascade, released after DifuzzRTL would appear to be objectively better than the mixed fuzzer, while in practice, it can find fewer bugs. One practical inconvenience of a mixed fuzzer is that it requires the instrumentations and probing mechanisms of both fuzzers, which is mostly an engineering challenge, but which also conditions the practical adoption of the fuzzer. Since Cascade boasts a low-effort adoption without any instrumentation and a bug discovery based on non-termination, which is easy to implement in practice, a more practical approach might be to extend Cascade to generate broader programs.

9 Discussion

We discuss innocuous transformations, compatibility with fuzzers that rely on HDL source code, specialization of our verification framework, and automatic generation of additional bug-injecting transformations.

Correctness-preserving transformations. ENCARSIA leverages formal methods to ensure that the transformations cause architecturally observable deviations. Yet not all observable transformations violate the specification, i.e., the ISA. For instance, a transformation that unconditionally adds one clock cycle of latency to all instructions is architecturally observable but does not violate the RISC-V specification, provided it does not violate the timing requirements of the memory hierarchy or other critical components. When manually analyzing many bugs generated by ENCARSIA in Section 7, we observed no such occurrence in practice. If such a case would occur, its impact would be limited to a slight reduction in the apparent performance of the fuzzers, and would affect all fuzzers uniformly. We encourage further work that would rule out such correctness-preserving transformations completely.

Changes at language level. To transform the design, ENCARSIA operates on the Yosys intermediate representation. This process vastly transforms the syntax of the source code, for example, by converting complex constructs into simpler ones. This means that potential fuzzing techniques that rely on the source code for input generation or coverage measurement might be affected by the transformation. Since Yosys does not yet support all SystemVerilog features, assertions are not preserved in the generated code, which might affect assertion-guided fuzzers.

Specialization. ENCARSIA relies on formal verification to ensure that the injected bugs are architecturally observable. In fact, the verification framework can be expanded to include additional properties. For example, it can ensure triggerability within a specific privilege level or other special configurations with tolerated deviations from the base specification [44]. ENCARSIA can also focus on injecting bugs into a given specific HDL module or set of modules. This permits the evaluation of fuzzers on more specific functionalities of the design. This is particularly useful for specialized fuzzers and for evaluating blind spots of a given fuzzer.

Automatically Derived Transformations. Deriving our bug injection transformations required considerable effort, including manual analysis of numerous pull requests. This raises the question of whether such transformations can be derived automatically and what their potential applications might be. Since our current transformations are highly generic and already cover all bugs identified in our survey, any additional transformations would essentially be subsets of these. However, focusing on specific bug types or scenarios could enable more detailed analysis that may not be possible with general transformations, making it a valuable direction for future research.

10 Related Work

We first discuss contemporary hardware fuzzer evaluations based on coverage and natural bugs. We then discuss manual bug insertions, bug surveys and software bug injection.

Fuzzer evalution via coverage. Coverage is typically involved in fuzzer evaluations. RFUZZ [32] compares with random testing by measuring multiplexer select coverage. DifuzzRTL [26] measured its own coverage when turning on or off coverage feedback. Some fuzzers compare with other fuzzers on a new coverage metric [14, 30], or compare with other fuzzers on their original coverage metrics [10, 42, 48]. No work has ever shown any correlation between coverage and bug discovery abilities in hardware fuzzing.

Fuzzer evaluation via natural bug discovery. Natural bugs are often used to evaluate hardware fuzzers [10, 26, 30, 42, 48]. While intuitive, this approach for comparison has several

drawbacks. First, natural bugs are increasingly harder to find as designs mature. Second, fuzzers are rarely evaluated on natural bugs found by older fuzzers. Finally, the lack of ground truth makes it impossible to derive fundamental performance metrics such as miss rates. ENCARSIA provides a systematic alternative for more fairly comparing hardware fuzzers.

Manually Assembled Corpora. Manually inserted bugs are occasionally involved in fuzzer evaluations. DifuzzRTL [26] uses one synthetic RTL module with one manually inserted bug for comparison with RFUZZ. The inserted bug is tailored to DifuzzRTL's coverage metric and might not be realistic.

Manually inserting realistic bugs into real-world designs is complex and demands a deep understanding of the design and the nature of bugs themselves. The HACK@EVENT competition series [16] is a rigorous attempt to create a manually compiled corpus of bugs. Its reliance on industry partners has led to a lack of transparency and reluctance to release the bugs to the public [40]. Of the ten competitions held over the past five years, the corpus has only been released for three. Additionally, among the 170 bugs released as part of these three corpora, only 28 included detailed descriptions or fixes. 26 bugs from the 2021 corpus closely resemble or are identical to those from the 2019 corpus. The HACK@EVENT competition has struggled to meet the demand for new bugs and has continued using the same CPU, CVA6, for years.

Survey of hardware bugs. A few past studies have focused on the analysis of real-world bugs, but on different aspects. RemembERR [43] is a database that classifies around 2'500 errata from recent x86 CPUs, The black-box nature of the entries classified in this database, however, hampers the understanding of structural properties of the bugs. Ma et al. [35] performed a review of 68 bugs in reconfigurable designs and produce tools to help developers simulate and deploy reconfigurable designs as well as a manually-created testbed of 20 well-documented bugs specific to FPGA designs [34].

Bug injection in software. Previous research has examined software bug injection. LAVA [18] proposed to cause segmentation faults if user-provided data matches specific arbitrary key values. The realism of these bugs has been many times questioned [7, 21, 31], and LAVA leaves artifacts that can be detected by white-box fuzzers [7]. Apocalypse [39] replaces LAVA's key equality condition with finite state machines to implement conditions under which a bug is triggered. SemSeed [38] uses machine learning to train on a corpus of real-world bugs and uses this model to adapt the learned patterns to the local context of the target program. FixReverter [50] derives a formal grammar from the real-world bugs and uses this grammar to generate new bugs. None of these approaches is directly applicable to hardware fuzzing, as software and hardware bugs are fundamentally different. ENCARSIA is the first systematic approach for injecting bugs into hardware.

11 Conclusion

We presented ENCARSIA, the first framework that automatically injects realistic bugs into RTL designs to support the evaluation of fuzzers. Based on ENCARSIA, we provide EnCorpus, a unified evaluation corpus made of 90 synthetic bugs verified to have an architecturally visible effect. To design ENCARSIA, we first conducted a comprehensive survey of bugs fixed in four popular open-source RISC-V CPUs of various complexities and design paradigms and found that these bugs can be modeled by wire connection mix-ups and errors in conditional statements. ENCARSIA leverages the Yosys intermediate representation to automatically inject transformations that reflect these representative bugs and employs formal verification to check whether these injected bugs are architecturally visible. We evaluate ENCARSIA using three RISC-V CPUs and three state-of-the-art CPU fuzzers. Based on this evaluation, we provide insights into the shortcomings of current fuzzers and suggest future directions for hardware fuzzer development, particularly regarding structural and functional coverage, and breadth of the input space.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

12 Ethics Considerations

In conducting this research, we took several ethical precautions to ensure that our work did not pose any risks to real-world systems, individuals, or intellectual property. The following sections detail our approach to injected bugs, potential misuse, data collection, and intellectual property compliance.

Injected bugs. In this paper, we intentionally inject bugs into RISC-V CPUs to assess the effectiveness of different bug detection techniques. These bugs are introduced solely in local instances of the devices and do not affect any public implementations of the CPUs. Consequently, our research does not create any security vulnerabilities in real-world systems, whether active or otherwise, and therefore, no disclosure process is required.

Risks of misuse. Risks of Misuse: While ENCARSIA could potentially be exploited to introduce bugs into proprietary hardware designs, such misuse would necessitate specific access to the design files. Given the requirement for this level of access, we assess the likelihood of direct misuse of our work to be low.

Data collection. The research did not involve any human subjects. No personal data was processed or analyzed. All

data collected as part of this study was sourced from public GitHub repositories.

Intellectual property. All tools used in this study, both open-source and commercial, were used in full compliance with their respective licenses.

13 Open Science

We make the artifacts of our study publicly available to the research community via https://github.com/comsec-group/encarsia and https://doi.org/10.5281/zenodo.14664723.

13.1 Artifact contents

These artifacts are mainly made of the following components:

- The detailed results of our RISC-V CPU survey.
- The source code of ENCARSIA, our custom-developed bug injection and verification tool.
- Two versions of ENCARSIA's verification setup: one based on proprietary JasperGold (to the extent of what our license agreements allow to share) and another based on the open-source Yosys.
- The EnCorpus set of CPU bugs.
- Reproducible evaluations of DifuzzRTL, Processorfuzz, and Cascade against the bug set.

13.2 Reproducibility

To facilitate reproducibility of our study, we release the complete evaluation setup as a Docker container. The container contains all scripts and configurations necessary to replicate our experiments. Specifically, it contains the Python script used to execute all tools within ENCARSIA in the correct sequence to generate a set of verified bugs. It then runs multiple instances of selected fuzzers in parallel to evaluate them across these bugs, significantly reducing the wallclock duration of the evaluation. The container also includes the setup details for operating all three fuzzers (DifuzzRTL, Processorfuzz, Cascade) on each core (Ibex, Rocket, BOOM), covering everything from the device source code with fixed bugs and the surrounding System-on-Chip to the fuzzer and core configurations. We believe that these artifacts enables other researchers to verify, reproduce, and extend our study.

13.3 Usability in future research

To let other researchers use our artifacts and better evaluate further bug detection research, we provide detailed guidelines on how to inject CPU bugs into arbitrary RISC-V CPUs.

References

[1] CHIPS Alliance. Rocket Chip Generator. https://github.com/chipsalliance/rocket-chip. Accessed: 2024-08-30.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *Tech. Rep. UCB/EECS-2016-17*, 2016.

[3] Armin Biere, Matti Järvisalo, Marijn JH Heule, and Norbert Manthey. Equivalence checking of hwmcc 2012 circuits. In *Proceedings of SAT Competition*, 2013.

[4] RISC-V BOOM. The Berkeley Out-of-Order RISC-V Processor. https://github.com/riscv-boom/riscv-boom. Accessed: 2024-08-30.

[5] Niklas Bruns, Vladimir Herdt, Daniel Große, and Rolf Drechsler. Efficient cross-level processor verification using coverage-guided fuzzing. In *VLSI*, 2022.

[6] Niklas Bruns, Vladimir Herdt, Eyck Jentzsch, and Rolf Drechsler. Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging. In *IEEE DATE*, 2022.

[7] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. Evaluating synthetic bugs. In *ACM ASIACCS*, 2021.

[8] Cadence. Jasper RTL Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-verification-platform.html. Accessed: 2024-08-08.

[9] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *ACM/IEEE DAC*, 2021.

[10] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *HOST*, 2023.

[11] Katharina Ceesay-Seitz, Sarath Kundumattathil Mohanan, Hamza Boukabache, and Daniel Perrin. Formal property verification of the digital section of an ultra-low current digitizer asic. In *accellera DVCON EUROPE*, 2021.

[12] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. μcfi: Formal verification of microarchitectural control-flow integrity. In *ACM CCS*, 2024.

[13] Christopher Celio, David A Patterson, and Krste Asanovic. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. *Tech. Rep. UCB/EECS-2015-167*, 2015.

[14] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hypfuzz: Formal-assisted processor fuzzing. In *USENIX Security*, 2023.

[15] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Handbook of Model Checking*. Springer International Publishing, 2018.

[16] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights into Software-Exploitable hardware bugs. In *USENIX Security*, 2019.

[17] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W Fletcher. ConjunCT: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *IEEE S&P*, 2024.

[18] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE S&P*, 2016.

[19] GitHub. GitHub REST API documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28. Accessed: 2024-08-18.

[20] OpenHW Group. CVA6 RISC-V CPU. https://github.com/openhwgroup/cva6/. Accessed: 2024-08-30.

[21] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In *POMACS*, 2020.

[22] Vladimir Herdt, Daniel Große, Eyck Jentzsch, and Rolf Drechsler. Efficient cross-level testing for processor verification: A risc-v case-study. In *FDL*, 2020.

[23] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. Speculation at fault: Modeling and testing microarchitectural leakage of {CPU} exceptions. In *USENIX Security*, 2023.

[24] Muhammad Monir Hossain, Nusrat Farzana Dipu, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. Taintfuzzer: Soc security verification using taint inference-enabled fuzzing. In *IEEE/ACM ICCAD*, 2023.

[25] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing. In *IEEE DATE*, 2023.

[26] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE S&P*, 2021.

[27] IEEE. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, 2024.

[28] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ACM/IEEE ICSE*, 2014.

[29] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *IEEE/ACM MICRO*, 2021.

[30] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *USENIX Security*, 2022.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM CCS*, 2018.

[32] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *IEEE/ACM ICCAD*, 2018.

[33] lowRISC. Ibex RISC-V Core. https://github.com/lowRISC/ibex. Accessed: 2024-08-30.

[34] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. Reproducible Hardware Bugs. https://github.com/efeslab/hardware-bugbase. Accessed: 2024-09-02.

[35] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. Debugging in the brave new world of reconfigurable hardware. In *ASPLOS*, 2022.

[36] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing black-box cpus against speculation contracts. In *ASPLOS*, 2022.

[37] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *IEEE S&P*, 2023.

[38] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *ACM ESEC/FSE*, 2021.

[39] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *ACM ESEC/FSE*, 2018.

[40] Ahmad-Reza Sadeghi, Jeyavijayan Rajendran, and Rahul Kande. Organizing the world's largest hardware security competition: challenges, opportunities, and lessons learned. In *GLSVLSI*, 2021.

[41] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *PATMOS*, 2017.

[42] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. In *USENIX Security*, 2024.

[43] Flavien Solt, Patrick Jattke, and Kaveh Razavi. Rememberr: Leveraging microprocessor errata for design testing and validation. In *IEEE/ACM MICRO*, 2022.

[44] Fabian Thomas, Lorenz Hetterich, Ruiyi Zhang, Daniel Weber, Lukas Gerlach, and Michael Schwarz. RISCVuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing. https://ghostwriteattack.com/riscvuzz.pdf. Accessed: 2024-09-02.

[45] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *USENIX Security*, 2022.

[46] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Austrochip*, 2013.

[47] Jiahui Xu and Lana Josipović. Automatic inductive invariant generation for scalable dataflow circuit verification. In *IEEE/ACM ICCAD*, 2023.

[48] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *USENIX Security*, 2023.

[49] Florian Zaruba and Luca 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.

[50] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. {FIXREVERTER}: A realistic bug injection methodology for benchmarking fuzz testing. In *USENIX Security*, 2022.

A Survey Overview

In this Appendix, Table 11 provides an overview of the latest two analyzed PRs per CPU and per category. We provide the detailed survey results as part of the Artifacts of this paper.

Table 11: The latest two analyzed PRs per CPU and category. (M = Mix-up, C = Broken Conditional)

Design	PR#	Bug Description	Type
Ibex	343	Wrong physical memory protection address matching	M
	332	Interrupt handler returned to misaligned address	M
	277	Missing access checks on debug CSRs	C
	272	DRET (debug return) executes outside debug mode	C
CVA6	206	Mixed up array index between instruction and data cache	M
	191	Race condition in data cache miss handler	M
	191	Race condition in data cache miss handler	C
	189	Wrong calculations due to floating point wrapper state machine	C
Rocket	592	Wrong signal order on bus protocol conversion	M
	589	Wrong address and data generated on bus	M
	574	Values changed when registers spilled to memory and read back	C
	440	Wrong data cache probe acknowledgement data	C
BOOM	451	A mathematical util function performed a wrong computation	M
	437	Illegal instruction dispatched into the LSU	M
	448	Wrong address used for data cache releases	C
	405	Tag update hazard for cache refills	C

USENIX Security '25 Artifact Appendix: Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection

Matej Bölcskei
ETH Zurich

Flavien Solt
ETH Zurich

Katharina Ceesay-Seitz
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

We provide the source code of ENCARSIA along with a Docker-based evaluation setup to facilitate reproducing the results presented in our accompanying paper and to support ENCARSIA's deployment for evaluating future CPU fuzzers. These artifacts demonstrate that ENCARSIA is a fully functional tool capable of injecting bugs into CPUs, formally verifying their architectural observability, and leveraging the generated bugs to assess fuzzers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

ENCARSIA is a bug injection and fuzzer benchmarking tool that does not pose any security risks, as it is not designed to attack the evaluation system. However, its high degree of parallelism, combined with the significant computational cost of formal verification and RTL simulation, can overwhelm system resources and potentially cause crashes.

A.2.2 How to access

Our artifacts are accessible for permanent access on Zenodo at `https://doi.org/10.5281/zenodo.14664723`. Alternatively, they can be accessed via GitHub at `https://github.com/comsec-group/encarsia`.

A.2.3 Hardware dependencies

ENCARSIA runs on any standard hardware, but requires substantial computing resources to handle the demands of formal verification, RTL simulation, and to take full advantage of parallelization for accelerating the experiments. We therefore recommend a machine with at least 32 CPU cores, 256 GB of main memory, and 512 GB of disk space. For detailed requirements of each experiment, refer to their descriptions in Section A.4.2.

A.2.4 Software dependencies

The following software dependencies are required for running ENCARSIA:

- **`docker`**: to run the experiments in the provided Docker environment.
- **`make`**: for automating tasks like building the Docker image.
- **`tar`**: to extract the EnCorpus dataset, which is provided as a .tar.gz archive.
- **`python3`**: for parsing the results of our bug survey.

If you prefer to run the experiments outside the Docker setup, the Dockerfile can serve as a reference for the necessary software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Clone the artifact repository using the GitHub link provided in Section A.2.2, or download it from the permanent access link to our artifacts on Zenodo. After cloning the repository, navigate to the root directory of the repository and run `make pull` to fetch the pre-built Docker image. Alternatively, you can run `make build` to build the Docker image locally.

After obtaining the Docker image, use `make run` to start a container from the image. Note the container ID displayed in the terminal output. You can use it later to restart and attach to the container for further experiments with `docker start <container_id> && docker attach <container_id>`.

A.3.2 Basic Test

Make sure the Docker container is running and attached. Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H ibex rocket boom -p 30` to confirm that ENCARSIA is functioning correctly and parsing the EnCorpus dataset as expected.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Bugs in open-source CPUs often stem from two simple syntactic transformations: mix-ups of signals or logic expressions and errors in conditional statements. This is proven by the bug survey **(E1)** described in Section 4 of the paper whose results are reported in Table 4.

(C2): ENCARSIA can rapidly inject a large number of diverse, realistic bugs into CPUs of varying complexity and design paradigms. This is proven by the injection experiment **(E2)** described in Section 7.1 of the paper whose results are reported in Table 5.

(C3): ENCARSIA can formally prove the architectural observability of bugs within a practical timeframe using computational resources typical for hardware design and verification. This is demonstrated by experiment **(E3)**, which follows a similar approach to the verification experiment described in Section 7.1 of the paper whose results are reported in Table 6. To make our verification framework more accessible, we provide a fully open-source version based on Yosys.

(C4): Instruction-granular bug detection mechanisms do not demonstrate greater potential for detecting bugs. This is demonstrated by the instruction-granular bug detection evaluation **(E4)** described in Section 8.1 of the paper whose results are reported in Table 8.

(C5): The hardware-specific structural coverage metrics, advertised as central by many fuzzers, are of little help in detecting bugs. This is demonstrated by the coverage metrics evaluation **(E5)** described in Section 8.2 of the paper whose results are reported in Table 9.

(C6): The fuzzer seeds are a key factor that determines which bugs will eventually be detected. This is demonstrated by the seed program experiment **(E6)** described in Section 8.3 of the paper whose results are reported in Table 10.

A.4.2 Experiments

All experiments are intended to be run within the provided Docker environment to ensure consistency and reproducibility. Before running each experiment, ensure that your system has sufficient computing resources available to meet the requirements specified for the experiment.

(E1): [Survey (Section 4)] [5 human-minutes]: The results of our bug survey are available in `survey/classification/synthetic.json` and `survey/classification/natural.json` within the main artifacts repository (not Docker image).
Execution: Use `survey/classification/plot.py` to display the results of the survey.
Results: The results of the survey confirm that all identified observable bugs indeed do fall into one of the two categories.

(E2): [Injection (Section 7.1)] [5 human-minutes + 20 compute-minutes + 100GB disk + 8GB memory]: This experiment exactly replicates the injection experiment described in Section 7.1 of the paper.
Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/Injection -H ibex rocket -p 30`. Optionally, add `boom` to the `-H` option to inject bugs into BOOM, but note that this requires up to 512GB of disk space.
Results: This experiment injects around 1000 Signal Mix-ups and 1000 Broken Conditionals per CPU. The resulting `host.v` files and injection logs can be found in the experiment directory at `/encarsia-meta/out/Injection`. A summary of the injection results, similar to Table 5 in the paper, is printed to the terminal. We expect the summary table to closely match the one presented in the paper.

(E3): [Bug Verification] [5 human-minutes + 1 compute-hour + 6GB disk + per-process memory (device dependent: 4 GB for Ibex, 8 GB for Rocket, 32 GB for BOOM)]: This experiment closely replicates the verification experiment outlined in Section 7.1 of the paper, with the main difference being our use of a fully open-source verification setup based on Yosys. Additionally, we limit the experiment to bugs from the EnCorpus bug set to reduce the duration of the experiment.
Execution: Assess your system's computing resources to determine the number of parallel processes it can support for this experiment. Then, navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H ibex rocket boom -p NUM_PROC -Y`, replacing `NUM_PROC` with the number of parallel processes. You can also execute the experiment on one device at a time by running `python encarsia.py -d out/EnCorpus -H DEVICE -p NUM_PROC -Y` for each of the three devices. This ensures optimal memory usage, despite varying memory requirements across devices.
Results: This experiment generates formal proofs of architectural observability for the EnCorpus bugs using the Yosys setup. The resulting verification log (`yosys_verify.log`) and proof of observability (`yosys_proof.S`) can be found in the EnCorpus experiment directory at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the verification results, similar to Table 6 in the paper, is printed directly to the terminal. Note that EnCorpus was verified using the JasperGold setup, which is more robust and powerful. As a result, not all EnCorpus bugs are expected to verify successfully using Yosys. We nevertheless expect Yosys to verify most of the EnCorpus bugs in the simpler CPUs (Ibex and Rocket) and a smaller subset in the more complex BOOM. Furthermore, we expect the average verification time per bug to be similar to the values

reported in Table 6 of the paper.

(E4): [Instruction-granular bug detection evaluation] [5 human-minutes + 10 compute-hours + 100GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.1 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F no_cov_difuzzrtl no_cov_processorfuzz`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 8 in the paper, is printed directly to the terminal.

We expect the results to match those presented in the paper, except for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14. These bugs are detected by DifuzzRTL and Processorfuzz in the Docker setup, but remain undetected in the bare-metal setup used to generate the data presented in the paper despite several fuzzing re-runs. We suspect this discrepancy stems from differing versions of dependencies, such as Spike or Verilator, used internally by the fuzzers. However, the lack of transparency regarding which tools are used and their specific versions makes it difficult to determine the exact cause. Despite this discrepancy, we firmly believe that the major claims of the paper remain valid.

(E5): [Coverage metrics evaluation] [5 human-minutes + 10 compute-hours + 100GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.2 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F difuzzrtl processorfuzz`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 9 in the paper, is printed directly to the terminal.

As in the previous experiment, we observe discrepancies between the Docker and bare-metal setups for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14.

(E6): [Seed program evaluation] [5 human-minutes + 90 compute-minutes + 10GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.3 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Note that this experiment reuses the results of the DifuzzRTL evaluation from experiment **(E5)**, if available, which reduces computation time and disk space requirements. If the results of experiment **(E5)** are not available, the requirements are approximately half those of experiment **(E5)**.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F difuzzrtl cascade`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 10 in the paper, is printed directly to the terminal.

As in the previous experiments, we observe discrepancies between the Docker and bare-metal setups for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14 on DifuzzRTL. However, the results for Cascade exactly match those reported in the paper.

A.5 Troubleshooting Guide

- **OOM Errors:** Ensure your system meets the minimum memory requirements specified for the experiment by reducing the number of parallel processes (`-p NUM_PROC`).
- **Running Out of Disk Space:** Verify that your system has sufficient disk space before starting an experiment. Clean up old experiment outputs if necessary.
- **Badly Terminated Parallel Experiments:** If an experiment is terminated ungracefully, some processes may remain running in the background. Use `ps` to identify and terminate any processes stuck in an infinite loop.

A.6 Notes on Reusability

ENCARSIA can be easily extended to support additional CPUs by creating a new `EncarsiaConfig` instance in `/encarsia-meta/config.py`. For more information, see the README in the ENCARSIA repository.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.