

SoK: You Find What You Seek

Rethinking Oracles, Guidance, and Input Generation in Hardware Fuzzing

G Abarajithan
UC San Diego

Zhenghua Ma
UC San Diego

Cristian Tirelli
UC San Diego

Andres Meza
UC San Diego

Francesco Restuccia
UC San Diego

Cynthia Sturton
*University of North Carolina
at Chapel Hill*

Ryan Kastner
UC San Diego

Abstract

Hardware fuzzing is an active area in security verification research, yet its industrial adoption remains in its early stages. This SoK examines which lessons from software fuzzing carry over to hardware and where unique approaches are needed. By analyzing 52 fuzzers across RTL/IP, CPU, NoC, and SoC designs, we introduce an analytical framework that frames verification as a bounded search. This search is defined by its objective, oracle, guidance, input generation, target abstraction, and budget. Consequently, a campaign only uncovers failures it can effectively reach, recognize, and prioritize before exhausting its resources. We distinguish two roles for hardware fuzzing: (1) augmenting constrained-random verification (CRV) via feedback-guided coverage and (2) directed adversarial testing based on threat models and security specifications. Through our framework, we identify what each campaign can observe and generate, providing a basis for assessing the evidence behind reported results. Our analysis suggests that mainstream adoption of hardware fuzzing will require reusable interfaces, target-specific verification assets, reproducible evaluations, and transparent reporting of cost and user effort.

1 Introduction

Fuzzing is an established technique in software testing. Google’s OSS-Fuzz has helped identify and fix over 13,000 vulnerabilities and 50,000 bugs across 1,000 open-source projects [23]. This success has motivated attempts to adopt fuzzing for hardware verification, especially in academic research. Hardware fuzzing promises to automate verification and expose corner cases that constrained-random verification (CRV) and ad-hoc directed adversarial testing may miss in security-critical designs. Despite this academic interest, hardware fuzzing has yet to be widely adopted in industrial hardware verification [43].

Software fuzzing began with automated random-input testing of UNIX utilities in 1990 [49]. Over three decades, it

matured through advances in failure detection, execution feedback, structured input generation, harnesses, and continuous fuzzing infrastructure [44]. Modern software fuzzing combines these components into workflows that can be deployed and maintained across many targets [17, 23].

Hardware fuzzing faces its own challenges as hardware executes in parallel and bugs rarely crash a register-transfer-level (RTL) simulation. Useful inputs also depend on whether the target is a CPU, an intellectual property (IP) block, or a full system-on-chip (SoC). Oracles, guidance, and input generation from software fuzzers therefore do not always transfer. Hardware fuzzing should adopt design patterns proven sustainable in software while adapting them to hardware verification objectives and existing verification assets.

In dynamic hardware security verification, hardware fuzzing can serve two objectives: (1) improving CRV by using feedback to steer test generation toward uncovered behaviors in a functional verification plan, and (2) performing directed adversarial testing by searching for an execution that violates a property derived from a security threat model. In either role, the campaign must align its checks, progress signal, and generated actions with the verification plan, threat model or security specification.

Analyzing 52 hardware fuzzers across RTL/IP, CPU, network-on-chip (NoC), and SoC targets, this SoK develops a framework to ask (1) where fuzzing complements established verification, (2) what each fuzzer can recognize, reach, and prioritize, and (3) when hardware fuzzers can be compared, reproduced, and deployed. Our contributions include:

- We develop an analytical framework that treats hardware fuzzing as bounded search. It separates the oracle that recognizes failures, the input generator that determines reachability, and the guidance that prioritizes executions within a finite budget. It identifies the subcomponents, design choices, and tradeoffs in this decomposition.
- We analyze and compare 52 hardware fuzzers across IP, CPU, NoC, and SoC abstraction levels, considering their failure oracles, structural, functional, and directed

guidance, input representations, mutation and scheduling policies, and solver-assisted target-to-input generation.

- We examine hardware-fuzzing guidance and identify a general mismatch between commonly used feedback signals and the functional and security behaviors encoded by verification plans and threat models.
- We compare evaluation, reproducibility, and deployment requirements. Drawing on software fuzzing and recent hardware-fuzzer evaluations [3], we identify conditions for fair comparison and mainstream adoption, including consistent metrics, versioned designs under test (DUTs), documented bug corpora, shared verification infrastructure, component ablations, and open artifacts.

2 The Framework: You Find What You Seek

We define fuzzing as guided verification, where observations from an execution influence how inputs are generated for future executions. As shown in Figure 1, our framework begins with a stated **verification objective**, which is the behavior, requirement, verification-plan item, or security property that a campaign seeks to exercise or falsify. The same DUT can be tested in campaigns with different verification objectives. The verification objective drives every other choice in the campaign.

2.1 Core Components of a Fuzzer

We analyze a verification campaign through three components. The oracle determines which failures can be recognized, input generation determines which behavior is tested, and guidance determines how observations affect the subsequent input generation. Together, these three determine what bugs or failures the fuzzer can find within a finite budget.

Oracle determines whether an observed execution violates a checked specification. It connects observations from the DUT to the design specification codified by the verification objective. Some oracles check only the DUT’s externally visible outputs. Others also inspect internal state or sequences of events across multiple clock cycles. In hardware verification, every class of failures we intend to find in a campaign must be explicitly defined in the oracle. Anything not checked remains invisible.

Guidance determines which executions receive attention under a finite campaign budget. The observation from an execution is turned into a **feedback** signal, which is used to guide the input generation. The feedback can be either **coverage** over a set of observations, or a scalar function (**score**) computed from the observations. The fuzzer generates inputs with the goal of either maximizing coverage or optimizing the

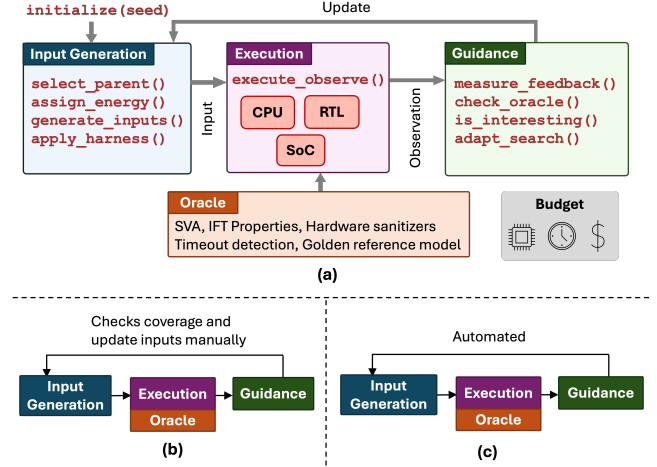


Figure 1: (a) Proposed framework: hardware fuzzing as budget-bounded search over input generation, execution, oracle, and guidance, with subcomponents from Algorithm 1. (b) CRV relies on engineers to manually inspect coverage and update input generation; (c) fuzzing automates this verification loop through feedback-guided input generation.

score function. For performance reasons, the feedback used for guidance is a lossy summary of a larger execution. Useful guidance preserves distinctions that matter to the verification objective while remaining inexpensive enough to collect throughout the campaign. If it rewards behavior unrelated to the objective, the fuzzer may make measurable “progress” without ever generating an input that will expose a failure.

Input generation determines which tests can be expressed and which behaviors are practically observable. It includes the input representation, seed corpus (initial inputs), generation and mutation mechanisms, and the policies that allocate generation effort. Expressibility and reachability limit a campaign in different ways. A representation may permit a test while the available seeds and transformations make it extremely unlikely to appear within the budget. Conversely, a generator may produce a diverse set of tests efficiently while accidentally excluding a specific test required to expose a failure. The **harness** maps a candidate from a fuzzer into an executable input to the DUT while preserving the timing, protocol, and system interactions needed by the campaign. If the input language or harness cannot express a triggering action, guidance cannot help to reach it.

2.2 Target Abstraction and Budget

The target abstraction constrains every component of the framework. Whether the DUT is treated as an IP, CPU, or SoC changes what observations can be used for guidance, which input representation is useful, which checks are practical,

and how much each execution costs. It also changes how the three components interact. Limited visibility constrains both the oracle and the feedback, while limited controllability constrains the harness and input representation. Execution cost limits how many tests the campaign can run.

A finite **budget** forces tradeoffs among these choices. A test that is technically reachable may still be irrelevant in practice if generating or executing it uses up the budget before the violation can be observed. In contrast, a more expensive feedback mechanism may increase per-execution time while reducing the number of executions needed to find a failure, potentially reducing total campaign time. Therefore, time, compute, memory, and human effort can all form part of this budget.

Together, the oracle, guidance, and input generation determine which failures a campaign can find within its budget. This is what we mean by *you find what you seek*.

3 Background

The framework in Section 2 describes fuzzing in terms of its objective, oracle, guidance, input generation, target abstraction, and budget. This section applies that vocabulary to software fuzzing, a generalized hardware-fuzzing loop, and established dynamic hardware verification practices, including CRV and property-driven security verification.

Hardware verification utilizes both static and dynamic techniques. Formal verification, a static technique, is considered the gold standard in verification, but it does not scale for complex designs such as large SoCs and CPUs. Dynamic verification executes concrete tests against a design in simulation, emulation, or physical hardware and observes the resulting traces. CRV is the most common form of dynamic hardware verification, in which inputs are generated randomly within constraints. In contrast, hardware fuzzing is a newer approach that uses observations from earlier executions to guide the generation of future test inputs. While this SoK focuses on dynamic campaigns, we include static analysis techniques when they are used to guide hardware fuzzing.

3.1 From Software to Hardware Fuzzing

Fuzzing is a self-guided verification technique in which inputs are generated based on feedback from prior executions. Early software fuzzers did not use such guidance [49], but it is a defining feature of contemporary fuzzers [17, 44]. A fuzzer can use this feedback to seek new coverage, reduce a distance or cost function, or reach an intermediate state associated with a target property. AFL++ is a representative software fuzzer, and its mutational loop roughly resembles Algorithm 1 [17]. Consider AFL++ fuzzing a PDF reader. It starts from a small corpus of PDF files, selects one as a parent, mutates its bytes, and uses a harness to invoke the reader. Lightweight instrumentation records the control-flow

Algorithm 1: Feedback-guided Hardware Fuzzing as pseudocode. Individual fuzzers may omit or combine operations and do not necessarily implement every function shown.

```

1 seed: initial set of candidate inputs
2 corpus, feedback_history, metadata = initialize(seed)
3 fail_history = {}
4 while campaign budget remains
5     parent = select_parent(corpus, metadata)
6     energy = assign_energy(parent, metadata)
7     candidates = generate_inputs(parent, energy)
8     for candidate ∈ candidates
9         test_vector = apply_harness(candidate)
10        observation = execute_observe(test_vector)
11        feedback = measure_feedback(observation)
12        failure = check_oracle(observation)
13        if failure
14            fail_history.update(candidate, failure, feedback)
15        if is_interesting(feedback)
16            feedback_history.update(feedback)
17            corpus.append(candidate, observation)
18            metadata.update(parent, candidate, observation)
19    adapt_search(corpus, feedback_history, metadata)

```

edges and bucketed edge counts activated by the execution. A candidate enters the corpus if it activates a new edge or a new count bucket for a known edge. AFL++ also records execution time, input size, path frequency, and favored status to prioritize parents and assign mutation effort.

However, software fuzzing techniques do not directly transfer to hardware. While software fuzzers can still encounter silent failures and structured inputs, these failures take different forms in hardware, as incorrect DUT behavior generally does not crash the simulator, and bugs may span numerous signals and clock cycles. These differences affect: (i) the harness, which must translate a candidate into structured input over multiple clock cycles; (ii) the guidance, which must summarize relevant hardware behavior; and (iii) the oracle, which must apply explicit design- or security-specific checks.

3.2 Anatomy of a Hardware Fuzzer

Algorithm 1 provides a high-level description of a hardware fuzzer and a common vocabulary for this SoK, while Figure 1 shows how its components interact. The loop selects a retained input, generates and executes candidates, measures progress, saves failures, retains interesting inputs, and repeats. Retained inputs form the **corpus**. The selected input is the **parent**, and a newly generated input is a **candidate**. A **harness** maps the candidate to executable DUT stimulus. **Feedback** summarizes progress and guides future exploration, while the **oracle** defined in Section 2 independently checks whether the execution exposes a failure.

initialize() builds the corpus from external, random, or solver-derived seeds. **select_parent()** may traverse the corpus, sample uniformly or by weight, or rank inputs by target distance. **assign_energy()** fixes or adjusts mutation effort

using metadata such as coverage, execution cost, distance, rarity, or past reward. **generate_inputs()** performs byte, bit, instruction, transaction, grammar-aware, structure-aware, or adversarial-action mutation. It may also combine parents or add solver-derived or language-model-generated candidates.

apply_harness() preserves the structure required by the target abstraction. RTL/IP fuzzers map bits or bytes to pins over several cycles. SoC/NoC fuzzers produce protocol transactions, firmware actions, or adversarial tampering, while CPU fuzzers produce instructions, initial memory, and interrupts. **execute_observe()** runs simulation or emulation on a field programmable gate array (FPGA). Fuzzers may use checkpoints, perform soft resets, or compare paired executions.

measure_feedback() derives coverage or a scalar score from the observation. Coverage may track control-flow-graph (CFG) edges, mux toggles, control registers, taint states, information-flow paths, or NoC transactions, while scores may estimate distance to a vulnerability condition. **check_oracle()** detects crashes, hangs, assertion or hyper-property violations, differential mismatches, timing differences, taint leaks, and sanitizer findings. **is_interesting()** compares feedback with campaign history and retains candidates that expand coverage or improve the best-known target distance, vulnerability cost, or another score. Some fuzzers perform no adaptation, while others use **adapt_search()** to respond to stagnation by changing mutations, combining inputs, resetting state, or invoking symbolic or formal generation.

3.3 CRV vs. Hardware Fuzzing

Hardware security verification combines functional verification of security-critical components with requirements derived from a threat model. Functional plans encode expected behavior through tests, covergroups, scoreboards, assertions, and reference models. Threat models identify assets, adversary capabilities, trust boundaries, and permitted behavior, then derive requirements for isolation, confidentiality, integrity, and availability. Both are needed because functional bugs can become vulnerabilities, while functional correctness alone does not establish resistance to an adversary.

CRV is the established approach for exercising functional verification plans and is typically implemented with standardized universal verification methodology (UVM) in SystemVerilog. Algorithm 2 distills the verification environment for OpenTitan’s `ac_range_check` IP [52]. OpenTitan is a root of trust widely adopted in consumer electronics whose per-IP environments include test plans, constrained sequences, drivers, monitors, scoreboards, assertions, and coverage models [53]. The IP enforces a Register Access Control List (RACL) role policy by checking TL-UL requests against configurable address ranges and read, write, and execute permissions [51]. Algorithm 2 contrasts its constrained input generation, functional coverage, and oracle with the hardware-fuzzing loop in Algorithm 1. CRV samples engineer-defined constraints and

Algorithm 2: Constrained-Random Verification of an OpenTitan Access-Control IP as pseudocode. Engineer-defined constraints generate structured TL-UL requests, functional coverage measures completeness, and a golden reference model, scoreboard, and assertions form the oracle.

```

1 ac = ConfigGen_AccessControl()
2 ac.ranges.constraint({valid address ranges})
3 ac.permissions.constraint(...)
4 gen = InputGen_TLUL()
5 gen.addr.constraint(ac.ranges, {higher weights
   around selected range})
6 cov=Coverage(ac.ranges,ac.permissions,ac.decisions)
7 cov.bins = cross(ranges, permissions, decisions)
8 cov.bins.illegal_bins({excluded bins})
9 orc = Oracle(GRM(),Monitor(),Scoreboard(),Asserts())

10 while total campaign budget remains
11   seed = next_seed()
12   dut.config = ac.randomize(seed, permissions)
13   sequence = gen.randomize(seed,addr,data,command)
14   for tlul_req ∈ sequence
15     observation = execute_observe(tlul_req)
16     expected = orc.grm(tlul_req, dut.config)
17     cov.sample(tlul_req, expected, observation)
18     orc.run_scoreboard(expected, observation)
19     orc.assert({completes without timeout})
20     orc.assert({denied read ⇒ data = 0})
21   //manually performed by engineers: {
22     inspect coverage for gaps
23     update constraints, tests, gaps
24   }
```

relies on engineers to revise campaigns, while fuzzing uses feedback from earlier executions to guide later inputs.

Random inputs in CRV remain structured and need not follow uniform distributions. In this example, constraints ensure valid access-control ranges and permissions. For each seed, `randomize` produces a configuration and a TL-UL sequence that preserves request structure across signals and cycles while varying `addr`, `data`, and `command`. The address constraint assigns 98% of its weight to a selected range and nearby addresses, with 2% assigned to the extremes of the address space. Enabled read and write permissions receive twice the weight of disabled permissions.

Functional coverage records a `cross` (combination) of ranges, permissions, and decisions, capturing interactions that coverage of each attribute alone would miss. It excludes unnecessary combinations to make coverage meaningful, and `cov.sample` records the observed interactions. Engineers inspect remaining gaps after a campaign and add tests, revise constraints, or document unreachable cases. The monitor, golden reference model, scoreboard, and assertions form the oracle. The monitor observes each transaction, the model computes the expected response, and the scoreboard compares and records expected and observed behavior. Assertions check same-cycle or temporal relationships. In this example, they detect hangs and data leakage from unauthorized reads.

Functional coverage measures campaign completeness, while the scoreboard and assertions detect failures.

At its simplest, CRV resembles early software fuzzing, which generated random inputs and treated crashes or hangs as failures without using execution feedback to guide later inputs [44, 49]. Industrial CRV is far more mature than early software fuzzing, but its feedback loop remains manual. Modern fuzzers automate this loop through guided input generation, which we use to distinguish hardware fuzzing from CRV. This boundary is shifting. A technique in a 2024 Synopsys patent guides constraint solving using coverage history, while Cadence Xcelium ML and Siemens Questa One Sim CX infer relationships between stimulus and coverage to accelerate coverage closure [2, 6, 63]. CRV and fuzzing lie on a continuum. Their convergence motivates hardware fuzzing to combine lessons from software fuzzing with mature CRV practices under hardware-specific objectives and constraints. This operational view provides context for the Oracle, Guidance, and Input Generation sections that follow.

4 Oracle

Hardware fuzzers differ not only in how they generate inputs and guide exploration, but also in how they determine that an execution is erroneous. At its core, an *oracle* is the mechanism that determines whether the DUT deviates from an expected behavior. The expected behavior may be represented explicitly as a property assertion or implicitly through a reference model or a comparison between executions. To formalize failure detection across these different fuzzing setups, we use a unified vocabulary based on three concepts: *property*, *assertion*, and *failure*. A *property* is a computable expression that encodes a particular expected behavior. An *assertion* is a mechanism that evaluates whether a property holds and reports a failure when it does not. We use *failure* to denote the detection of a deviation between the implemented and expected behavior, regardless of whether that deviation is identified through an assertion, a reference-model mismatch, a differential comparison, or another checking mechanism. This practical definition captures the failure-detection mechanisms used by hardware fuzzers in our survey without restricting the oracle to a particular formal representation of expected behavior.

4.1 Oracle Mechanisms and Blind Spots

The way expected behavior is represented determines what an oracle can observe and, consequently, which classes of erroneous behavior a fuzzer can detect. For example, an architectural reference model can identify deviations from the architectural specification of a processor, but may not expose microarchitectural or information-flow violations that do not manifest in architectural state [19, 37, 48]. Other fuzzers therefore rely on specialized models, comparisons between

multiple executions, or explicitly specified properties and verification monitors. Across the surveyed fuzzers, these mechanisms can be organized into four broad oracle families, summarized in Table 1. The first uses an architectural reference model to compare the DUT against an ISA-level representation of expected behavior. The second uses specialized models to capture microarchitectural, security, timing, or information-flow behavior that is not represented by architectural state. The third identifies failures through differential or relational comparisons between executions or states within the DUT. The fourth relies on explicitly specified properties and verification infrastructure, including assertions, monitors, scoreboards, and sanitizers. These families describe how expected behavior is represented and are not mutually exclusive; an oracle may combine several complementary mechanisms.

For each executed test, the oracle determines whether the observed behavior violates the specification. Assertions and protocol monitors enforce local, temporal, and interface rules; timeout checks detect hangs; and scoreboards and reference models detect departures from expected execution [67]. Sanitizers look for recurring weakness patterns [34]. Hyperproperties can encode confidentiality, integrity, and non-interference requirements; hardware fuzzers can check them through self-composition, miter, or information-flow tracking mechanisms [46, 50]. Any failure undetected by the oracle remains invisible to the verification campaign [14, 34].

The oracle must likewise match the target and threat model. An IP may use local assertions, protocol checks, or a block-level reference model; a CPU may use architectural differential testing; and an SoC may require properties spanning firmware, interconnects, peripherals, and security state [30, 34, 50]. A local assertion does not establish an SoC-wide security invariant [34].

4.2 Failure Validation and Triage

An oracle report identifies a potential failure, not necessarily a confirmed bug or its root cause [68, 77]. When a failure occurs, the input sequence that generated it must be analyzed alongside the DUT’s RTL, specification, and properties, to determine what went wrong and where a fix must be applied [46]. This analysis becomes more difficult when a campaign continues after the first failure and produces many reports, including duplicate manifestations of the same bug [68]. Therefore, the reports must be reproduced, minimized, grouped, and validated before they become useful findings [64, 68].

Assigning a relative priority to validated failures requires assessing their impact on functionality, security, or safety and the scope of the repair [46]. Their frequency, reproducibility, and analysis effort provide additional operational criteria. Fuzzers may assist this process using information they already collect. Automated input reduction can preserve the bug-triggering state while making long programs tractable to analyze, and signatures derived from execution traces can

Table 1: Oracle mechanisms across the surveyed fuzzers.

Oracle mechanism	Expected behavior / comparison	Fuzzers
Reference-model comparison (21)	DUT architectural, functional, or transaction-level behavior checked against a separate reference model or model-derived expected result.	[8–11, 22, 24, 25, 30, 31, 33, 40–42, 57, 64, 67–70, 72, 75]
Specialized semantic / cost model (8)	A domain-specific execution, leakage, information-flow, contract, or vulnerability-cost model supplies the detection semantics.	[13, 15, 19, 20, 27, 28, 37, 71]
Differential / relational (11)	Paired executions, implementations, states, timing observations, or information-flow views are compared; ordinary DUT-versus-golden functional checking is listed above.	[4, 19, 29, 37, 50, 55, 58, 62, 66, 71, 74]
Property / runtime checker (13)	An assertion, contract, security/protocol predicate, scoreboard, generated sanitizer rule, or expected-termination check evaluates each fuzzed execution.	[12, 18, 19, 34, 40, 42, 47, 48, 50, 59, 64, 65, 67]

Scope note. The four rows classify implemented failure-detection mechanisms and contain 53 non-exclusive placements covering 44 of the 52 surveyed frameworks. BugsBunny and RFUZZ delegate relevance or failure detection to a user-defined target or external checker [38, 54]. FMTC, TargetFuzz, DirectFuzz, FuSS, ProFuzz, and RTLfuzzLab report reachability or coverage but do not implement a pass/fail oracle [7, 16, 32, 39, 60, 61].

group duplicate reports [64, 68]. Future hardware-fuzzing papers should report the manual effort spent, the triage or severity-scoring method employed, and the post-processing needed to make the fuzzer’s output useful.

Google’s OSS-Fuzz project shows that the fuzzing loop is only one part of deploying fuzzing at scale. Its largely automatic failure detection reports crashes, sanitizer findings, timeouts, and out-of-memory executions, while ClusterFuzz provides continuous fuzzing infrastructure [23]. These reusable partial oracles require little target-specific specification. Target project maintainers normally maintain recommended seed corpora and dictionaries where applicable [23].

An oracle only detects failures that are both observable and explicitly checked. Its findings should be validated and prioritized before being treated as bugs.

5 Guidance

Guidance determines how a fuzzer spends a finite campaign budget. Feedback from earlier executions may cause the fuzzer to retain a candidate, prioritize a parent, assign more mutation effort, or change its search strategy. Since guidance influences input generation, the boundary between the two can be blurred. We therefore use *guidance* for what behavior receives attention and *input generation* for which candidates can be produced and how they are applied to the DUT.

A dynamic campaign executes only a small fraction of possible tests. Therefore, coverage is used to assess completeness and guide exploration, but it remains a lossy proxy for progress rather than a verification result. CRV uses structural coverage as a sanity check and relies primarily on functional coverage from the verification plan, whose covergroups can encode temporal, protocol-specific, and cross-event behavior. NoCFuzzer illustrates the tradeoff: CRV closed an easy router covergroup faster, whereas fuzzing closed harder code and functional targets that CRV did not reach within the budget [42]. Guidance is beneficial only when it reduces search effort enough to offset the cost of feedback.

Table 2 summarizes execution-derived guidance across the surveyed fuzzers. The *source* records what is observed, while the *type* records whether those observations become coverage or a numeric score. The same feedback type can carry different meanings depending on whether it comes from implementation structure, a selected region, a verification plan, or a threat model. The fuzzers in the *Other* row do not adapt their search using feedback from earlier DUT executions. Instead, their user-selected targets, static or formal analysis, execution models, historical data, or construction rules shape input generation [20, 37, 41, 60, 64, 66].

AFL helped make feedback-guided mutational fuzzing practical and widely adopted by popularizing approximate CFG-edge coverage and bucketed hit counts as inexpensive search proxies [17, 44]. Hardware can reuse this design pattern, but the useful feedback changes with the abstraction level. Fine-grained signals such as mux toggles may guide a fuzzer on an isolated IP but become noisy or expensive at CPU and SoC scale. At those levels, guidance may track control and status register (CSR) transitions, instruction sequences, NoC transactions, or functional covergroups [8, 42, 50].

5.1 Source and Types of Feedback

The most common source of feedback is the implementation of the design itself. Line, branch, condition, toggle, mux, and register-state coverage can be derived automatically from the RTL or netlist of the design. They require less target-specific user effort than a functional coverage model or a security property, but show only that a test exercised new logic. For example, the select pins of four 2:1 muxes can go from 0000 $\rightarrow$ 1111 $\rightarrow$ 0000 and achieve 100% mux-toggle coverage in a single, short trace. Yet four independent select pins create up to $2^{4 \times 10} \approx 1$ trillion traces over ten cycles, and mux-toggle coverage preserves neither event order nor protocol context.

Coverage feedback is constructed by choosing observable values and defining coverage items over them. Instrumentation monitors those values, and the fuzzer retains a candidate when it contributes an item absent from the accumulated history. RFUZZ’s mux-select activity, NoCFuzzer’s UVM covergroups, and HyperFuzzing’s instruction, NoC-access, and fault-read events all use novelty, but preserve different

Table 2: Guidance across 52 hardware fuzzers. The *type* of feedback is either Coverage over a set of source observations to be expanded, or a numeric Score function to be optimized. Counts are non-exclusive and fuzzers appear in every applicable row.

Source	Type	Details of the guidance mechanism
Implementation structure / state (32)	C (24)	RTL or Verilator control flow [13, 18, 47, 48, 59, 67]; mux toggle [16, 38]; full-mux toggle [16, 39]; statement and expression [33]; line [16, 42]; branch [4, 10, 32, 33, 42]; condition and FSM [4, 10, 33]; toggle [32, 33, 72]; VCS code-coverage points [31, 34]; per-cycle control-register state [30, 54, 55, 70, 75]; architectural CSR transitions [8]; netlist cell toggle and expression [62]
	S (8)	structural-coverage-derived genetic-algorithm fitness [40]; particle-swarm optimization [9]; multi-armed and contextual-bandit reward [11, 22]; language-model (LM) reinforcement learning (RL) [57, 68]; deep-RL reward [24]; and LM tuning [69]
Selected region of design (6)	C (4)	target nets [61]; target-module mux activity [7]; changes on signal close to property [12]; ancestor-register state coverage [65]
User-defined behavior (3)	S (2)	distance from covered muxes to the target module sets energy [7]; target-signal dependency distance sets seed priority [54]
	C (1)	UVM functional covergroups for NoC [42]
Threat model or security property (12)	S (2)	temporal density of an annotated event assigns mutation effort [65]; packet waiting time [42]
	C (6)	adversary-action bigrams and fault-read events [50]; secret-propagation taint matrix [71]; self-composition divergence [19]; toggles on statically found leakage paths within speculative window [58]; taint propagation [13]; opcode/rollback-reason pair [29]
Program (1)	S (6)	vulnerability cost function [15, 28]; taint and target-output proximity cost [27]; transient-window size, after a new trigger cause–opcode pair [29]; request interval at a contention point [74]; rarity-weighted self-composition-deviation prioritization [19]
	S (1)	deviation of executed opcode frequencies from the expected distribution rebalances opcode selection in later rounds [25]
Other (6)	-	No execution feedback; generation uses static/formal analysis, execution models, historical data, or rules. [20, 37, 41, 60, 64, 66]

information about an execution [38, 42, 50].

Not all hardware fuzzers are coverage-guided. Directed fuzzers instead compute a numeric score such as target distance, security cost, estimated influence on an asset, or contention interval [15, 27, 28, 54, 74]. NoCFuzzer prefers inputs that increase packet waiting time and move toward starvation, while TaintFuzzer’s cost includes inferred input influence on outputs and proximity to asset leakage [27, 42]. Such scores can measure directed progress, but they still remain lossy.

Coverage and score-based feedback can coexist. Bugs-Bunny first uses register coverage to discard uninteresting tests and computes target distance for survivors [54]. DirectFuzz similarly uses target-module coverage to prioritize corpus entries and a distance score to assign mutation energy [7]. This can avoid computing an expensive score for every test. The score’s optimizer is a separate choice. GenFuzz and PSO-Fuzz apply genetic and particle-swarm search to structural coverage without changing the feedback signal [9, 40].

Feedback derived from a security specification is also guidance. HyperFuzzing’s event bigrams are intended to lead the search toward adversarially affected behavior, while its HyperPLTL oracle determines whether the paired executions violate a property [50]. An intermediate taint state can likewise guide a fuzzer to expose an asset-to-sink leakage path.

5.2 Target Selection

Adopting directed guidance for fuzzing requires defining the specific target of the campaign. An engineer must state the verification goal, a user or algorithm must select a target signal or region, and the input generator must produce an input that activates it. We treat the first two as guidance and the last as input generation. An automatic solver can construct an input for a target, but it cannot determine whether that target represents the user’s verification intent.

For security verification, the target normally originates in a

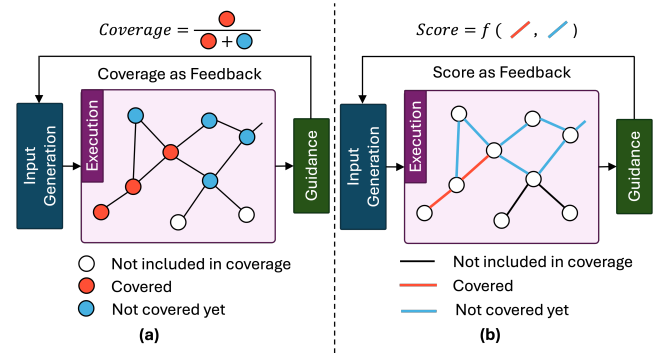


Figure 2: Types of guidance (a) Coverage rewards new coverpoints or register states. (b) A score function $f()$ measures progress toward a target, e.g., asset-to-output leakage.

threat model or property. VGF automates part of this translation by deriving monitored signals from a user-supplied security property [12]. ProFuzz selects nets manually, randomly, or by scoring fan-in, fan-out, and Shannon entropy in a netlist hypergraph [61]. Automatic targets remain structural unless connected to a verification plan or threat model.

5.3 Alignment Across the Guidance Chain

A guidance signal is the end of a chain of choices. The verification objective is translated into a target behavior, the behavior into observable values, and those observations into coverage or a score. A search policy or an optimizer then uses the resulting signal to retain candidates, select parents, or assign effort. Each mapping can discard information. Functional coverpoints begin from verification-plan behavior, and security-derived scores begin from a property, asset, or vulnerability condition [12, 27, 42]. Structural feedback begins from readily observable implementation activity and assumes

that new activity correlates with the objective [38]. Thus, coverage over a functional coverpoint may align better than a distance score to a structurally selected module, while a score derived from a security condition may align better than structural coverage. No optimizer can recover distinctions discarded earlier in the chain.

Feedback is a lossy proxy and is not evidence that a design is bug-free. Guidance should be designed backward from the verification objective through target behavior, observation, feedback, and search policy.

6 Input Generation

Input generation determines which tests the fuzzer can produce given the feedback from guidance. A bug whose triggering input is never generated will stay undetected. Input generation includes the candidate representation, seed corpus, mutation or construction operations, solver assistance, allocation of generation effort, and the harness that applies a candidate to the DUT. Table 3 groups these mechanisms into mutation, construction without a parent, and solving backward from a target.

6.1 Input Grammar and Harness

Grammar is the structure that defines how the fuzzer’s inputs are interpreted by the DUT. The same byte string generated by mutation can be sliced into pin values, decoded into a transaction sequence, interpreted as a program, or consumed as commands for an adversarial tamperer. The harness is the interface that applies the grammar to generate inputs and is therefore part of the input generator. Industry-standard CRV has counterparts to both: the OpenTitan example in Algorithm 2 preserves the TL-UL transaction grammar and protocol, while its UVM driver serves a harness-like role by applying generated transactions to the DUT.

The generator must balance validity with flexibility. Too little structure wastes executions on invalid or benign inputs, while too much can exclude the bug-prone or adversarial interactions the campaign is intended to test. A transaction generator fuzzing a crossbar without concurrent requests cannot expose contention failures. A CPU-specific input generator without interrupts cannot expose a bug that requires one.

At the IP level, a harness can typically map candidate bytes to pin values over clock cycles or to selected transaction fields. CPUs instead require programs together with execution state such as initial memory, privilege modes, and interrupts. At SoC and NoC scale, the generator must coordinate processors, firmware, interconnects, peripherals, and security state over longer executions. Such harnesses translate candidates into programs, firmware actions, packets, or UVM transactions while preserving ordering, handshakes, concurrency,

and responses [34, 42, 67]. Firmware alone may not express malicious NoC transactions or memory faults. The grammar must include explicit threat-model entry points [50].

Software fuzzing encountered and overcame similar tension between validity and flexibility. Production fuzzers now use target-specific corpora, dictionaries, parser-aware harnesses, and custom mutators. Csmith and syzkaller similarly encode domain knowledge for C programs and system-call sequences [23, 44, 73]. Hardware can reuse an ISA across CPUs, but IP and SoC harnesses remain protocol- or design-specific.

RFUZZ maps bytes directly to primary-input values over successive cycles [38]. This requires little design knowledge but does not preserve bus transactions; Fuzzing Hardware Like Software instead decodes bytes into TileLink transactions, and NoCFuzzer uses a user-written grammar to construct UVM sequences across NoC agents [42, 67]. NoCFuzzer also shows why harness design is critical. A mapping in which each mutated byte affected multiple NoC queues performed worse than random testing [42].

At the SoC level, the grammar also reflects how the adversary is modeled. Transaction-aware harnesses avoid protocol-invalid tests, but may not express attacks that deliberately break protocol, such as those explored by eXpect and Xray [76, 77]. HyperFuzzing uses tamper functions for untrusted-firmware instructions, NoC writes, and memory bit flips [50]. SoCFuzzer and TaintFuzzer instead run target-specific C executables on the SoC and mutate their inputs [27, 28]. The harness, grammar, seeds, and encoded domain knowledge should therefore be versioned campaign artifacts.

6.2 Candidate Generation Mechanisms

Surveyed fuzzers mutate parents, construct inputs without parents, solve backward from targets, or combine these mechanisms. Mutation engines may transfer across targets, but a complete generator may not because input grammars differ.

6.2.1 Mutation and Construction

Parent-derived mutation applies deterministic bit flips or arithmetic changes and nondeterministic “havoc” operations such as insertion, deletion, and overwriting. Splice or recombination operations draw from multiple parents [17, 38].

Inputs can also be constructed without being derived from a parent. SymbFuzz randomizes UVM input vectors and InterConFuzz randomizes packet fields while preserving sequence structure [47, 48]. After coverage stagnates, both add solver-derived constraints to redirect generation. Constructed or solved inputs can then become parents for mutation.

6.2.2 Construction using Language Models

Several recent CPU fuzzers use generative pre-trained transformer (GPT)-like language models to construct RISC-V

Table 3: Input generation across 52 hardware fuzzers. The first column is how a fuzzer treats the target. For example, a fuzzer may treat a CPU as an IP. The second column defines how an input is generated: **(M)**utation of a parent, **(C)**onstruction without a parent, and **(S)**olving for target. Counts are non-exclusive and fuzzers with composite mechanisms appear in multiple rows.

Target	Gen.	Details of the input-generation mechanism
IP	M(9)	bytes $\rightarrow$ pins $\times$ cycles [7, 12, 16, 18, 38, 39, 59]; ATPG-seed bit-vector mutation [61]; packet (per-cycle pin values)-level mutation [54];
(10)	S(3)	Inputs by SAT solving for target-node states with Hamming-distance constraint for diversity [60]; merged ATPG activation patterns for target nets [61]; recorded state of a partially toggled mux initializes Z3-backed completion [39]
SoC / NoC	M(7)	SoC-harness is firmware in FPGA emulation [15, 27, 28]; same, with manual constraints from formal traces [15]; bytes configure firmware/NoC/memory flip tamperers [50]; bytes $\rightarrow$ UVM sequences [42]; minimized IP bus transactions $\rightarrow$ SoC program seeds $\rightarrow$ mutation [34]; mutation on opcode/operand fields of a precompiled benchmark program [32]
(9)	C(2)	CRV-style UVM randomization of input vectors [48] or packet fields [47], with constraints tightened when coverage stalls
	S(3)	a minimal suffix solved from the fuzzing trajectory at a plateau [32]; checkpoint-derived constraints that restrict later UVM randomization [47, 48]
CPU	M(23)	ISA-instruction or program-IR edits [4, 11, 13, 22, 24, 31, 33, 40, 41]; instructions, interrupts, and addresses set by program execution [8, 19, 30, 55, 65, 70, 72]; testcase-template/transient-seed mutation [58, 74]; editing tests they first construct or solve [9, 10, 29, 71, 75];
(31)	C(12)	programs emitted by a language model [57, 68, 69]; sampled from fixed or learned instruction distributions [9, 29, 66, 75]; constructed under an ISS or execution model that supplies operands, prerequisites, page tables, or entangled control and data flow [20, 25, 37, 64, 71]
	S(1)	a model checker solves an uncovered point into an executable test [10]
Misc(3)	M(3)	bytes $\rightarrow$ TileLink transactions driving an IP [16, 67]; netlist inputs labelled control & data by user and mutated separately [62]

programs and fine-tune their generation using execution feedback without mutating a parent. ChatFuzz pretrains on Linux-kernel-derived RISC-V code and uses disassembler validity and RTL coverage as rewards [57]. GoldenFuzz builds programs from instruction blocks, refining validity on an ISA model before using DUT coverage [69]. Because such fine-tuning can overfit toward rewarded patterns, GenHuzz resets its model when coverage stalls and GoldenFuzz keeps older blocks in a recency-weighted memory [68, 69]. These models are learned program grammars, not mechanisms for deriving tests from a threat model.

6.2.3 Solving from Target to Input

Once a target net or region has been selected as described in Section 5.2, a fuzzer can search for an activating input through mutation or solving. Mutation is cheap but indirect, while construction via target-to-input solving can be resource-intensive but more direct. ProFuzz selects target nets, uses automatic test pattern generator (ATPG) to generate and merge activation patterns, and mutates the resulting seeds [61].

Software fuzzers also supplement fast mutation with specialized mechanisms for hard-to-reach paths. Developed at Microsoft Research, SAGE achieved industry-scale symbolic execution by controlling costs through techniques such as constraint slicing and caching [5, 21]. AFL++ includes Redqueen for simpler input-to-state correspondences [1, 17]. Because RTL simulation is slow, selective solver assistance may justify its overhead, but evaluations should report solving and execution costs separately.

FormalFuzzer divides the work differently between the tool and the engineer. Formal verification produces assertion traces, but an engineer interprets them using the ISA and design specification and chooses which fields may be mutated [15]. Thus, its formal stage restricts mutation rather

than directly producing a fuzz test.

6.3 Search Effort: Seeds and Scheduling

A useful seed may already establish protocol state, firmware configuration, privilege, or a transaction sequence that mutation is unlikely to reconstruct. Empirical evaluations show that fuzzer performance is highly sensitive to initial seeds [3, 28, 42, 67]. TaintFuzzer builds a smart-seed pool by combining inferred influence on hardware outputs with proximity to vulnerability-specific behavior [27]. FUZZItizer collects a set of inputs that maximize coverage at the peripheral level, translates their bus transactions into SoC programs, and uses those programs as seeds for full-SoC fuzzing [34]. NoCFuzzer found mutation more effective with shorter seeds, provided they remained long enough to express the behavior under test [42].

Beyond the seeds, mutation is shaped by three policies: parent selection chooses a corpus input, energy determines the number of attempts, and operator scheduling defines the applied transformations. DirectFuzz prioritizes inputs that cover mux-select signals in the target module and assigns more energy to inputs whose coverage is structurally closer to that module. After coverage stalls, it restores the default energy of a low-energy input to escape a local minimum [7]. SoCFuzzer and TaintFuzzer instead use a security-oriented cost to change mutation strategies; TaintFuzzer exhausts deterministic operators before switching to havoc [27, 28].

The grammar and harness limit expressibility. Seeds, construction, solving, and scheduling affect practical reachability. Evaluations should ablate these contributions and present their verification artifacts, cost, and human effort.

7 Evaluation Practices

Hardware-fuzzer evaluations commonly report bugs, time to detection, and achieved coverage. However, the evidence is difficult to compare across fuzzers [3], and rarely establishes whether fuzzing improves upon modern CRV. Our framework provides a systematic basis for determining when such comparisons are meaningful. Evaluations should first align the verification objective, target abstraction, oracle, and budget, and then isolate the effects of guidance and input generation. Table 4 summarizes current evaluation practice.

7.1 Comparability between Fuzzers

Direct comparison requires the same verification objective and target abstraction. A CPU campaign for transient-execution leakage could be compared with campaigns for the same leakage model, but not with IP-level functional coverage closure. The DUT name alone is also insufficient because its configuration and revision determine its behavior and bug set. Many studies name Rocket, BOOM, or OpenTitan without identifying the evaluated revision. Encarsia likewise identifies design and version diversity as a barrier to comparing naturally occurring bug counts [3].

Aligned verification objectives can still use different oracles. SpecDoctor combines a rollback monitor with differential timing tests, DejaVuzz uses differential information-flow tracking and taint-liveness annotations, and MileSan compares architectural and microarchitectural information flows and requires leakage to affect instruction timing [29, 37, 71]. These checks recognize different failure sets despite their shared high-level goal. Evaluations of guidance or input generation should therefore use the same oracle whenever possible. Otherwise, the result compares complete campaigns and cannot attribute an advantage to the search mechanism alone. The oracle specification, observations, and blind spots must then be reported as experimental variables.

Once these conditions align, component ablations should change one choice at a time. Guidance includes observations, feedback, retention, and scheduling. Input generation includes the harness, grammar, corpus, mutation or construction, and solver assistance. Encarsia demonstrates their interaction. Two CPU fuzzers found the same injected bugs with structural feedback enabled and disabled, while different seed programs changed which bugs were found [3].

Coverage does not ensure a bug-free design, but maximizing a widely accepted structural or functional metric could be a meaningful contribution when coverage closure is the stated objective. Such work should compare fuzzers with the same objective and abstraction, starting from a common corpus and evaluating their generated tests with the same coverage tool and configuration. Replaying fuzzer-generated inputs into the DUT under the independent metric separates evaluation coverage from each fuzzer’s internal feedback.

Initial corpora and pseudorandom seeds require explicit control. Compared fuzzers should start from the same corpus when their grammars permit it. Studies should report distributions and uncertainty over multiple independent seeds because fuzzing performance is seed-sensitive [3, 42, 67]. A reproducible artifact should preserve these campaign-level control variables and the execution scripts. If artifacts cannot be released, papers should compensate with comprehensive reporting of coverage metrics, effectiveness results, execution budgets, and compute and resource costs. If the fuzzer itself cannot be released, the complete, ordered set of inputs generated over each evaluated campaign should also be released so others can replay the baseline, measure coverage independently, and reproduce bug findings. Sec. 8 discusses the deployment requirements.

7.2 Comparison with CRV

Few hardware fuzzers compare against modern CRV. When doing so, uniform random generation and disabling guidance are useful ablations. However, neither is an adequate baseline for modern CRV, which combines constrained stimulus generation with a verification plan, functional coverage model, and oracle. Fuzzers that seek broad functional bugs or coverage closure are proposing to automate part of this CRV workflow and should compare against it directly [8, 30, 38, 42, 59].

Comparisons should give each approach the same time and compute budget, then measure the coverage achieved and bugs found rather than the number of tests executed. Fuzzing can incur greater per-test overhead than CRV due to guidance and input generation overhead. But fuzzing could be better than CRV if fewer tests close important coverage targets or expose failures. Results should state the targets and conditions under which either approach wins rather than give a universal ranking. NoCFuzzer provides a rare example by comparing fuzzing and CRV in the same UVM environment using shared coverage targets [42].

We propose a comparison based on the target design’s existing regression when one is available. Run the nightly CRV regression and record its duration, compute resources, simulator configuration, oracle outcomes, and engineer-defined functional coverage. Give the fuzzer the same time and compute budget, record the inputs it generates, and replay them through the CRV environment with the functional covergroups and oracle illustrated in Algorithm 2. This replay measures test quality with the same engineer-defined functional coverage model and oracle instead of the fuzzer’s internal coverage. The comparison should also report memory use and the manual effort for CRV constraints and for fuzzer-specific harnesses, seeds, properties, and instrumentation.

Known-bug experiments provide a complementary result. The fuzzer and CRV regression should run against the same faulty revision or injected bugs and report the number found and detection-time distributions. Reporting whether each

Table 4: Evaluation methods of 52 hardware fuzzers grouped by target design, reported metrics, and comparison with CRV.

Practice	Type	Details
Popular DUTs	SoC	Ariane/CVA6 [15, 27, 28, 32, 48]; OpenPiton [34, 42, 47]; OpenTitan [47, 48]; Caliptra [34]; PicoSoC, UeRVSoC, VeeRwolf [32]; Custom [50].
	CPU	Rocket or BOOM in all but RISCover [66]; CVA6 follows [4, 9–11, 22, 24, 25, 33, 37, 64, 68–70, 75]; morlxx/or1200 [10, 30, 33]; additional core families [8, 11, 25, 29, 37, 41, 65, 71, 72, 74, 75]; commercial RTL [69]; closed-source silicon [66].
	IP	SiFive TileLink peripherals, FFT [7, 38]; OpenTitan AES, HMAC, KMAC, RV-Timer [18, 67]; spinal.lib [59]; OpenCores and Trust-Hub IPs [12, 60–62]; small CPU cores [7, 38, 39, 60–62]; BOOM submodules [54]; generated FSM locks [67]; netlist analysis/fuzzing [60–62].
Report Metrics	SoC	Artifact: Public infrastructure [32, 50]; repository not reproducing the evaluated stack [48]; none found [15, 27, 28, 34, 42, 47]. Coverage: adversary-action bigrams [50]; branch and toggle [32]; VCS code coverage [34]; control-register state [47, 48]; line, branch, UVM covergroups [42]; none, cost-function convergence [15, 27, 28]. Progress: executions [15, 27, 28], vectors [48], packets [47], programs [34], or wall-clock [32, 42]. Comparison with other fuzzers: Self-ablation [15, 27, 28, 32, 42, 47, 50]; same-lineage rerun [15]; artifact rerun [48]; reimplementation under one coverage tool [34]; numbers from its paper [47, 48]; feature table [15, 27, 28]; no compatible prior fuzzer [42].
	CPU	Artifact: Public code [8–11, 13, 22, 24, 25, 29–31, 33, 37, 55, 64–66, 70–72]; partial/placeholder [4, 40, 41, 68]; promised, no URL [75]; on request [69]; none found [19, 20, 57, 58, 74]. Coverage: control-register state [25, 30, 40, 41, 64, 70, 75]; ISS CSR transitions [8]; commercial-simulator metrics [33], narrowed to a subset by successors [9–11, 22, 24, 57, 68, 69]; taint, leakage, timing, or contention signals [4, 13, 19, 58, 71, 74]; none [29, 37, 66]. Progress: wall-clock or test count. Comparison with other fuzzers: Artifact-derived [8, 25, 41]; baseline tests replayed in-house [70, 71]; reimplementation [9, 22, 30, 64, 65]; numbers taken from prior work [13, 22, 24, 37, 64, 72]; baseline embedded as component [10, 11]; provenance unstated [40, 57, 58, 68, 69, 74, 75]; feature table [4, 29, 55].
	IP	Artifact: Public code [16, 38, 59, 67]; promised, no URL [7]; none found [12, 18, 39, 54, 60–62]; commercial ATPG/synthesis/simulation licenses also needed [60–62]. Coverage: mux-select toggle [7, 38], stricter under the same name [39]; Verilator line or basic block [18, 59, 67]; netlist library-cell toggle [62]; target activation [60, 61]; bespoke value, distance, or flow-path metrics [12, 54]. Ten report no bugs; known benchmark faults [12]; new synthesis bugs and CVEs [62]. Comparison with other fuzzers: Fork of the baseline codebase [7]; reimplementation on another toolchain [39, 60]; external fuzzer on shared designs [12]; numbers from its paper [18, 61]; self-ablation only [38, 54]; commercial equivalence checker [62]; feature table [18, 62]; no evaluation [16].
Compared to CRV	SoC	Explicit CRV in the same UVM environment [42]; UVM random testing [48]; uniform random regression [34]; random without symbolic execution [47]; CRV as motivation only [32]; none [15, 27, 28, 50].
	CPU	riscv-dv, on one core only [70]; riscv-torture [30, 65, 70]; random regression [10, 33]; generators used only as a seed corpus [41, 72]; guidance-disabled ablation, not CRV [8, 11, 19, 20, 40, 66, 74, 75].
	IP	CRV in the same testbench [59]; CRV on generated FSM locks only [67]; regression speedup cited, not run [18]; uniform random [38]; random only as seeds [39, 54]; none [7, 12, 16, 60–62].

bug was reached, triggered, and detected helps distinguish failures of input generation from blind spots in the oracle. Similarly, directed adversarial fuzzers should be compared against baselines targeting the same security objective. A generic coverage regression cannot establish whether a fuzzer is better at triggering a transient-execution leak or violating an information-flow property unless the CRV campaign is configured to seek that same behavior.

7.3 Proposed Evaluation Practices

Every evaluation should identify the DUT commit and configuration, verification objective, target abstraction, oracle, corpus, time and resource budget, tools, and measurement conditions. It should also preserve raw results and report distributions from repeated trials. Coverage, time to failure, resource use, and user effort measure different aspects of a campaign and should be reported separately. Claims of generality require varied representative objectives and target abstractions.

Software fuzzing faced the same fragmented evaluation landscape. Klees et al. found problems in every one of 32 surveyed evaluations and showed that trials, seeds, timeouts, targets, statistics, and metrics can change conclusions [36]. FuzzBench turned these lessons into a service with versioned real targets, common corpora, equal resources, repeated trials, independent coverage, raw data, and statistical reports [45]. Magma added bug ground truth by reintroducing real bugs

into real programs and distinguishing reach, trigger, and detection [26].

Adapting the software benchmark model to hardware requires benchmark families organized by target abstraction and verification objective rather than a single leaderboard. A standardized benchmark should package versioned DUT configuration with its harness, corpus, oracle, coverage models, faulty and fixed revisions, injected bugs, and replay scripts. It should define resource limits and measurement procedures while allowing each fuzzer to retain its own internal guidance. Separate tracks could cover a suite of security IPs, an industry-scale NoC, a complex CPU, and a full SoC. OpenTitan, Caliptra, OpenPiton, and BOOM are promising starting points because they span these abstractions and are already used by multiple surveyed fuzzers [30, 34, 42, 48]. OpenTitan also provides extensive open verification assets [53].

The bug set should combine documented real bugs with reproducible injected bugs. Hack@DAC designs already provide competition-injected security bugs in PULPissimo, OpenPiton, and OpenTitan, while EnCorpus provides 90 formally checked CPU bugs [3, 48, 56]. Existing SystemVerilog assertion suites can serve as shared oracles for some of these benchmark bugs [56]. Providing both faulty and fixed DUT revisions allows researchers to confirm that a reported failure is caused by the benchmarked bug. Using our framework, benchmarks should label each bug by its verification objective, target abstraction, and oracle, allowing readers to distinguish out-of-scope bugs from in-scope bugs not reached or detected.

Automatic property generation could eventually provide a common set of properties and violations, but it remains an open problem [35]. For now, the practical alternative is a versioned benchmark with curated assertions, reference models, covergroups, and known bugs. Building them will require substantial community effort and no initial suite will satisfy every objective. A shared benchmark service would nevertheless be a major contribution and help establish the evidence and infrastructure needed for the widespread adoption of hardware fuzzing.

Fair evaluation aligns the objective, target abstraction, DUT version, oracle, and budget before comparing search mechanisms. Coverage-maximizing fuzzers should be measured against established CRV, while directed campaigns need baselines with the same security objective. The community should build shared, versioned benchmarks to make these comparisons reproducible.

8 Deployment and Reproducibility

The reproducibility challenges discussed in Section 7 stem partly from deployment requirements that extend beyond the fuzzer itself. Users may need to supply or reconstruct target-specific artifacts, an oracle, toolchains, and access to the design or execution platform, affecting both usability and reproducibility. We examine these requirements as the artifacts needed to construct inputs and specify verification objectives, and the infrastructure access to run campaigns.

8.1 Required Artifacts

The initial deployment hurdle involves providing the design artifacts and input-generation artifacts required by the fuzzer. At the structural level, nearly all frameworks require access to a representation of the target hardware, but the required representation varies across methodologies. Some operate on high-level hardware descriptions such as Chisel, Flexible Intermediate Representation for RTL (FIRRTL), and SpinalHDL [29, 30, 59], while others require conventional Verilog or SystemVerilog RTL [10, 15, 48]. More specialized approaches require additional artifacts, such as synthesized netlists and standard-cell libraries [62], ATPG-specific configuration [61], or complete UVM environments for NoC verification [42, 47].

The harness and seed corpus artifacts required to generate valid inputs likewise vary with the target abstraction. Processor fuzzers rely on program-aware representations [25, 30, 33, 70], whereas SoC and interconnect fuzzers [15, 27, 28, 42, 47] and RTL-IP fuzzers [18, 60, 67] depend on target-specific input models and harnesses. As Section 6 details, the performance of fuzzers and the failures they can find within a limited budget are highly sensitive to these artifacts, making them a critical part of deployment.

CRV-style verification can build on established verification flows and reusable infrastructure, whereas directed adversarial testing typically requires more manual effort to translate campaign-specific threat models and security specifications into properties that various tools can use. Formal- and property-driven approaches require users to encode behaviors or properties that guide exploration [10, 15, 48, 50]. Microarchitectural security fuzzers may require threat models, speculation gadgets, taint annotations, or contention templates [20, 29, 37, 58, 71, 74]. In these settings, encoding the security assumptions and specifications that define the search space accounts for most of the user effort.

Recent machine-learning and optimization-based approaches extend this requirement to substantial data preparation and configuration. Large language model (LLM)-based frameworks may require large instruction datasets, while reinforcement-learning and optimization-based approaches rely on auxiliary state representations, reward formulations, seed-generation configurations, optimization parameters, or historical test suites [9, 11, 22, 24, 57, 68, 69]. Consequently, the artifacts required to deploy a fuzzer can extend well beyond the DUT and a seed corpus, with user effort increasingly determined by the domain-specific information required by the methodology.

Table 5: Visibility models adopted across the 52 fuzzers.

Visibility	Works
Black-box (2)	[41, 66]
Grey-box (30)	[7, 9, 10, 15, 16, 27, 28, 33, 38–40, 60–62, 70] [8, 12, 18, 19, 22, 25, 30, 31, 54, 55, 57, 59, 64, 67, 72] [11, 24, 29, 32, 34, 42, 47, 48, 50, 69, 75]
White-box (20)	[4, 13, 20, 37, 58, 65, 68, 71, 74]

8.2 Infrastructure and Design Visibility

A primary challenge in deploying hardware fuzzers is the toolchain infrastructure and specialized hardware required to execute them. A substantial portion of the analyzed CPU, SoC, and NoC fuzzers depend on proprietary simulation or formal verification tools, most notably Synopsys VCS and Cadence JasperGold/Xcelium [4, 10, 11, 22, 24, 31, 33, 34, 42, 57, 58, 61, 62, 68–70]. Open-source simulators such as Verilator are widely used for RTL-IP modules and lightweight CPU targets [7, 8, 18, 25, 30, 32, 38, 50, 54, 59, 64], but they often require additional co-simulation infrastructure, including custom C++ harnesses and external libraries such as Z3, PyVerilog, or cocotb [15, 16, 19, 30, 32, 39, 47, 48, 58]. Most academic fuzzers also require users to adapt their designs, testbenches, and verification assets to a custom fuzzing environment rather than integrating directly into an existing industry-standard workflow such as a UVM-based environment [43].

These requirements can extend beyond the simulation environment itself. Frameworks that rely on physical or hardware-accelerated execution may require specialized hardware platforms, while machine learning (ML)- and LLM-guided approaches can introduce substantial compute and software dependencies in addition to conventional UVM infrastructure [24, 27, 28, 57, 66, 68, 69, 75]. Thus, deployment may require not only software toolchains but also dedicated hardware and compute environments that are not part of the fuzzer.

These infrastructure requirements are closely coupled with the level of internal design access required by the fuzzer. As shown in Table 5, most surveyed frameworks use grey-box or white-box visibility. Grey-box approaches use lightweight instrumentation to obtain runtime feedback, while white-box approaches require access to internal design artifacts such as RTL, netlists, or data flow and may require users to integrate assertions, sanitizers, or information-flow tracking logic. Black-box approaches avoid internal instrumentation but provide less internal observability and may rely on physical execution environments. Thus, the visibility model determines both what information the fuzzer can exploit and what design access and instrumentation the user must provide.

From a reproducibility perspective, these requirements mean that reproducing a published campaign may require substantially more than the fuzzer implementation itself. Researchers may also need the exact DUT and configuration, input artifacts, tool versions or licenses, supporting verification infrastructure, and specialized hardware. When these components are unavailable or unreleased, independent reproduction becomes difficult even when the design is public.

Hardware fuzzing should move towards well-defined, stable interfaces between reusable fuzzer engines and target-specific verification assets. Researchers should release these assets to improve reproducibility, support fair comparisons, and accelerate industry adoption.

9 Summary and Conclusion

In this paper, we introduce a framework developed through an analysis of 52 hardware fuzzers. The framework treats verification as a bounded search shaped by the verification objective, campaign budget, target abstraction, and the components of the fuzzer. We then examine how software fuzzing grew from a novel testing technique into a mature research field with widespread industry adoption. This history provides lessons that can help advance hardware fuzzing.

Our central takeaway is *you find what you seek*. A fuzzing campaign finds failures only where its oracle, input generator, and guidance overlap and align with its verification objective, within its limited budget. The oracle defines the set of failures that can be detected, guidance prioritizes tests for further

exploration, and input generation produces new tests. All three mechanisms individually and collectively define what the campaign seeks, and improving one component cannot remove the limits imposed by the others. Generic structural feedback reduces manual effort, but it is meaningful only when it preserves behavior relevant to the objective. Coverage closure is informative when the covered items correspond to a verification plan, while directed progress requires a target and score derived from a property or threat model. Quality guidance therefore depends on preserving relevant distinctions from the verification objective through target behavior, observation, feedback, and search policy. Hardware fuzzers should therefore strive for strong alignment among their oracle, guidance, and input generation so that every component directs the campaign toward the intended failures. Researchers should also perform subcomponent-level ablations to understand how each choice affects the fuzzer’s behavior.

The same principle applies to target-specific verification assets, which define what a campaign can generate, reach, and observe. Fuzzers, like other search and optimization engines, are likely to be better suited to particular applications and bug classes. The community should establish a comprehensive family of benchmarks across the IP, CPU, NoC, and SoC designs to identify these relative strengths. Each benchmark should include versioned DUTs and the complete verification and fuzzing collateral needed to reproduce a campaign, including seeds, covergroups, oracles, injected and real bugs, and replay scripts. It should also define target metrics, resource budgets, and testing procedures for fair comparisons. Coverage-maximizing fuzzers should be compared with established CRV in the same environment, with the manual effort for both approaches reported. Directed adversarial campaigns should show that their guidance and input generation reach a specified violation more effectively than alternatives with a comparable objective, threat model and budget.

Finally, industry adoption requires reusable tools that support continuous fuzzing, failure triage, and reliable replay across platforms. Software fuzzing became sustainable through reusable engines, stable interfaces, documentation, standardization, open-source development practices, continuous triage, and shared evaluation infrastructure [17, 23, 45]. Hardware verification already maintains target-specific UVM sequences, constrained generators, assertions, scoreboards, reference models, functional covergroups, and threat models. Researchers who seek adoption should make their tools easy to integrate with these workflows and release the assets needed to reproduce and extend their work. As a community, we should address the unique challenges of hardware verification by drawing lessons from the path established by software fuzzing. For hardware fuzzing to become mainstream, its components must align with the verification objective, integrate with existing verification assets through stable interfaces, and be evaluated through reproducible campaigns that report effectiveness, cost, and required user effort.

A Open Science

This paper is a Systematization of Knowledge based on an analysis of publicly available literature. It does not report new experiments or empirical measurements requiring source code, datasets, binaries, or other experimental artifacts. The works analyzed in this paper are identified in the bibliography.

References

- [1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.
- [2] Parijat Biswas, Badri Gopalan, Enzhi Ni, Danish Jawed, Ying Chen, and Jiang Chen. Connecting random variables to coverage targets using an ensemble of static analysis, dynamic analysis and machine learning and guided constraint solving of the random variables during simulation of an integrated circuit. U.S. Patent 11,983,474 B1, May 2024. Filed Dec. 23, 2021; issued May 14, 2024. URL: <https://patents.justia.com/patent/11983474>.
- [3] Matej Bölskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Encarsia: Evaluating CPU fuzzers via automatic bug injection. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2577–2593, Seattle, WA, August 2025. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity25/presentation/bolskei>.
- [4] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. WhisperFuzz: White-Box fuzzing for detecting and locating timing vulnerabilities in processors. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5377–5394, Philadelphia, PA, August 2024. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/borkar>.
- [5] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131, May 2013. doi:10.1109/ICSE.2013.6606558.
- [6] Cadence Design Systems, Inc. Cadence delivers machine learning-optimized Xcelium logic simulation with up to 5x faster regressions, August 2020. Accessed: Aug. 12, 2026. URL: https://www.cadence.com/en_US/home/company/newsroom/press-releases/pr/2020/cadence-delivers-machine-learning-optimized-xcelium-logic-simula.html.
- [7] 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 *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 529–534, Dec 2021. doi:10.1109/DAC18074.2021.9586289.
- [8] 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 *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12, May 2023. doi:10.1109/HOST55118.2023.10133714.
- [9] Chen Chen, Vasudev Gohil, Rahul Kande, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Psfuzz: Fuzzing processors with particle swarm optimization. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, Oct 2023. doi:10.1109/ICCAD57390.2023.10323913.
- [10] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HyPFuzz: Formal-Assisted processor fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1361–1378, Anaheim, CA, August 2023. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/chen-chen>.
- [11] Chen Chen, Zaiyan Xu, Mohamadreza Rostami, David Liu, Dileep Kalathil, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. ReFuzz: Reusing tests for processor fuzzing with contextual bandits. In *Proceedings of the 2026 Network and Distributed System Security Symposium*. Internet Society, 2026. URL: <https://www.ndss-symposium.org/ndss-paper/refuzz-reusing-tests-for-processor-fuzzing-with-contextual-bandits/>, doi:10.14722/ndss.2026.240118.
- [12] Ruochen Dai, Michael Lee, Patrick Hoey, Weimin Fu, Tuba Yavuz, Xiaolong Guo, Shuo Wang, Dean Sullivan, and Orlando Arias. Vgf: Value-guided fuzzing—fuzzing hardware as hardware. *arXiv preprint arXiv:2312.06580*, 2023.
- [13] Alvis de Faveri Tron, Raphael Iseemann, Hany Ragab, Cristiano Giuffrida, Klaus von Gleissenthall, and Herbert Bos. Phantom trails: Practical Pre-Silicon discovery of transient data leaks. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2539–2556, Seattle, WA, August 2025. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity25/presentation/de-faveri-tron>.

- [14] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. Isadora: Automated information flow property generation for hardware designs. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, ASHES '21*, page 5–15, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3474376.3487286.
- [15] Nusrat Farzana Dipu, Muhammad Monir Hossain, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. Formalfuzzer: Formal verification assisted fuzz testing for soc vulnerability detection. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference, ASPDAC '24*, page 355–361. IEEE Press, 2024. doi:10.1109/ASP-DAC58780.2024.10473911.
- [16] Brandon Fajardo, Kevin Laeuffer, Jonathan Bachrach, and Koushik Sen. Rtlfuzzlab: Building a modular open-source hardware fuzzing framework. In *Workshop on Open-Source EDA Technology (WOSET)*, 2021.
- [17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies, WOOT'20*, USA, 2020. USENIX Association.
- [18] Deepak Narayan Gadde, Aman Kumar, Djones Lettnin, and Sebastian Simon. Fuzzwiz - fuzzing framework for efficient hardware coverage. In *2024 International Symposium on Electronics and Telecommunications (ISETC)*, pages 1–5, Nov 2024. doi:10.1109/ISETC63109.2024.10797245.
- [19] Gideon Geier, Pariya Hajipour, and Jan Reineke. Coverage-guided pre-silicon fuzzing of open-source processors based on leakage contracts, 2025. URL: <https://arxiv.org/abs/2511.08443>, arXiv: 2511.08443.
- [20] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 874–887, June 2021. doi:10.1109/ISCA52012.2021.00073.
- [21] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, 2008. URL: <https://api.semanticscholar.org/CorpusID:1296783>.
- [22] Vasudev Gohil, Rahul Kande, Chen Chen, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Mabfuzz: Multi-armed bandit algorithms for fuzzing processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, March 2024. doi:10.23919/DATE58400.2024.10546726.
- [23] Google. OSS-Fuzz: Continuous fuzzing for open source software. <https://github.com/google/oss-fuzz>, 2016. Accessed: Aug. 12, 2026.
- [24] Raphael Götz, Christoph Sendner, Nico Ruck, Mohammadreza Rostami, Alexandra Dmitrienko, and Ahmad-Reza Sadeghi. RLFuzz: Accelerating hardware fuzzing with deep reinforcement learning. In *2025 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 358–369. IEEE, 2025. doi:10.1109/HOST64725.2025.11050051.
- [25] Zihui Guo, Miaomiao Yuan, Yanqi Yang, Liwei Chen, Gang Shi, and Dan Meng. Divefuzz: Enhancing cpu fuzzing via diverse instruction construction. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1964–1978, 2025.
- [26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020. doi:10.1145/3428334.
- [27] 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 *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, Oct 2023. doi:10.1109/ICCAD57390.2023.10323726.
- [28] 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 *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, April 2023. doi:10.23919/DATE56975.2023.10137024.
- [29] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*, page 1473–1487, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560578.
- [30] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303, May 2021. doi:10.1109/SP40001.2021.00103.

- [31] Intel Labs. PreSiFuzz: Pre-silicon hardware fuzzing toolkit. <https://github.com/IntelLabs/PreSiFuzz>, 2022. GitHub repository, accessed July 27, 2026.
- [32] Aruna Jayasena, Sai Suprabhanu Nallapaneni, and Prabhath Mishra. Fuss: Coverage-directed hardware fuzzing with selective symbolic execution. *ACM Trans. Embed. Comput. Syst.*, 24(5s), September 2025. doi:10.1145/3760529.
- [33] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, Boston, MA, August 2022. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/kande>.
- [34] Rahul Kande, Mohamadreza Rostami, Chen Chen, Haresh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Fuzzitizer: Hardware sanitizer-assisted fuzzing for automated soc vulnerability detection. In *Proceedings of the ACM Asia Conference on Computer and Communications Security, ASIA CCS '26*, page 1139–1155, New York, NY, USA, 2026. Association for Computing Machinery. doi:10.1145/3779208.3806078.
- [35] Ryan Kastner, Francesco Restuccia, Andres Meza, Sayak Ray, Jason Fung, and Cynthia Sturton. Automating hardware security property generation: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22*, page 1384–1387, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3489517.3530637.
- [36] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243804.
- [37] Tobias Kovats, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Milesan: Detecting exploitable microarchitectural leakage via differential hardware-software taint tracking. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS '25*, page 2579–2593, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3719027.3765066.
- [38] Kevin Laeuffer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: coverage-directed fuzz testing of rtl on fpgas. In *Proceedings of the International Conference on Computer-Aided Design, IC-CAD '18*, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3240765.3240842.
- [39] Tun Li, Hongji Zou, Dan Luo, and Wanxia Qu. Symbolic simulation enhanced coverage-directed fuzz testing of rtl design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2021. doi:10.1109/ISCAS51556.2021.9401267.
- [40] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Bruce Khailany, Shih-Hsin Wang, and Tsung-Wei Huang. Genfuzz: Gpu-accelerated hardware fuzzing using genetic algorithm with multiple inputs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, July 2023. doi:10.1109/DAC56929.2023.10247942.
- [41] Hao Lyu, Jingzheng Wu, Xiang Ling, Yicheng Zhong, Zhiyuan Li, and Tianyue Luo. Simfuzz: Similarity-guided block-level mutation for risc-v processor fuzzing. *arXiv preprint arXiv:2601.11838*, 2026.
- [42] Ruiyang Ma, Jiayi Huang, Shijian Zhang, Yuan Xie, and Guojie Luo. Nocfuzzer: Automating noc verification in uvm. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 44(1):371–384, January 2025. doi:10.1109/TCAD.2024.3430195.
- [43] Ruiyang Ma, Tianhao Wei, Jiaxi Zhang, Chun Yang, Jiangfang Yi, and Guojie Luo. Bridging the gap between hardware fuzzing and industrial verification. In *Proceedings of the Great Lakes Symposium on VLSI 2025, GLSVLSI '25*. Association for Computing Machinery, 2025. doi:10.1145/3716368.3735190.
- [44] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, Nov 2021. doi:10.1109/TSE.2019.2946563.
- [45] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3473932.
- [46] Andres Meza, Francesco Restuccia, Jason Oberg, Dominic Rizzo, and Ryan Kastner. Security verification of the opentitan hardware root of trust. *IEEE*

- Security & Privacy*, 21(3):27–36, May 2023. doi:10.1109/MSEC.2023.3251954.
- [47] Samit Shahnawaz Miftah, Hyunmin Kim, and Kanad Basu. Interconfuzz: A fuzzing-based comprehensive noc verification framework. In *Proceedings of the 62nd Annual ACM/IEEE Design Automation Conference*, DAC ’25. IEEE Press, 2025. doi:10.1109/DAC63849.2025.11133216.
- [48] Samit Shahnawaz Miftah, Amisha Srivastava, Hyunmin Kim, Shiyi Wei, and Kanad Basu. Symbfuzz: Symbolic execution guided hardware fuzzing. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO ’25, page 1477–1490, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3725843.3756131.
- [49] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. doi:10.1145/96267.96279.
- [50] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. Hyperfuzzing for soc security validation. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD ’20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3400302.3415709.
- [51] OpenTitan Project. Access control range check (AC Range Check). OpenTitan documentation, n.d. Accessed: Aug. 22, 2026. URL: https://opentitan.org/book/hw/top_darjeeling/ip_autogen/ac_range_check/index.html.
- [52] OpenTitan Project. AC_RANGE_CHECK design verification. OpenTitan documentation, n.d. Accessed: Aug. 22, 2026. URL: https://opentitan.org/book/hw/top_darjeeling/ip_autogen/ac_range_check/dv/index.html.
- [53] OpenTitan Project. Design verification methodology within OpenTitan. OpenTitan documentation, n.d. Accessed: Aug. 12, 2026. URL: <https://github.com/lowRISC/opentitan/blob/master/doc/contributing/dv/methodology/README.md>.
- [54] Hany Ragab, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Bugsbunny: Hopping to rtl targets with a directed hardware-design fuzzer. In *Fourth Workshop on the Security of Software/Hardware Interfaces*, pages 1–7. SILM, 2022.
- [55] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. Sigfuzz: A framework for discovering microarchitectural timing side channels. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, April 2023. doi:10.23919/DATE56975.2023.10136966.
- [56] Jayden Rogers, Niyaz Shakeel, Xiao Tan, Samantha Espinosa, Divya Mankani, Cade Chabra, Kaki Ryan, and Cynthia Sturton. Hardware security benchmarks for open-source systemverilog designs. In *Proceedings of the Secure Development Conference (SecDev)*. IEEE, 2025.
- [57] Mohamadreza Rostami, Marco Chilese, Shaza Zeitouni, Rahul Kande, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. Beyond random inputs: A novel ml-based hardware fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, March 2024. doi:10.23919/DATE58400.2024.10546625.
- [58] Mohamadreza Rostami, Shaza Zeitouni, Rahul Kande, Chen Chen, Pouya Mahmoody, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. Lost and found in speculation: Hybrid speculative vulnerability detection. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC ’24, page 1–6. ACM, June 2024. URL: <http://dx.doi.org/10.1145/3649329.3658469>, doi:10.1145/3649329.3658469.
- [59] Katharina Ruep and Daniel Große. Spinalfuzz: Coverage-guided fuzzing for spinalhdl designs. In *2022 IEEE European Test Symposium (ETS)*, pages 1–4, May 2022. doi:10.1109/ETS54262.2022.9810421.
- [60] Raghul Saravanan and Sai Manoj P D. Logic solver guided directed fuzzing for hardware designs, 2025. URL: <https://arxiv.org/abs/2509.26509>, arXiv:2509.26509.
- [61] Raghul Saravanan, Sudipta Paria, Aritra Dasgupta, Swarup Bhunia, and Sai Manoj P D. Profuzz: Directed graybox fuzzing via module selection and atpg-guided seed generation. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, Oct 2025. doi:10.1109/ICCAD66269.2025.11240782.
- [62] Raghul Saravanan, Sudipta Paria, Aritra Dasgupta, Venkat Nitin Patnala, Swarup Bhunia, and Sai Manoj P D. Synfuzz: Leveraging fuzzing of netlist to detect synthesis bugs, 2025. URL: <https://arxiv.org/abs/2504.18812>, arXiv:2504.18812.
- [63] Siemens Digital Industries Software. Questa One Sim CX: Coverage acceleration, n.d. Accessed: Aug. 12, 2026. URL: <https://www.siemens.com/en-us/products/ic/questa-one/simulation/coverage-acceleration/>.

- [64] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade:CPU fuzzing via intricate program generation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5341–5358, 2024.
- [65] Yuichi Sugiyama, Reoma Matsuo, and Ryota Shioya. Surgefuzz: Surge-aware directed fuzzing for cpu designs. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, Oct 2023. doi:10.1109/ICCAD57390.2023.10323819.
- [66] Fabian Thomas, Eric García Arribas, Lorenz Heterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. Riscover: Automatic discovery of user-exploitable architectural security vulnerabilities in closed-source risc-v cpus. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS ’25*, page 3326–3340, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3719027.3765141.
- [67] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254, Boston, MA, August 2022. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/trippel>.
- [68] Lichao Wu, Mohamadreza Rostami, Huimin Li, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. GenHuzz: An efficient generative hardware fuzzer. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 1787–1805, Seattle, WA, August 2025. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity25/presentation/wu-lichao>.
- [69] Lichao Wu, Mohamadreza Rostami, Huimin Li, Nikhilesh Singh, and Ahmad-Reza Sadeghi. Goldenfuzz: Generative golden reference hardware fuzzing. *arXiv preprint arXiv:2512.21524*, 2025.
- [70] 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 *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1307–1324, Anaheim, CA, August 2023. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jinyan>.
- [71] Jinyan Xu, Yangye Zhou, Xingzhi Zhang, Yinshuai Li, Qinhan Tan, Yinqian Zhang, Yajin Zhou, Rui Chang, and Wenbo Shen. Dejavuzz: Disclosing transient execution bugs with dynamic swappable memory and differential information flow tracking assisted processor fuzzing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS ’25*, page 64–80, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3676642.3736115.
- [72] Yinan Xu, Sa Wang, Dan Tang, Ninghui Sun, and Yungang Bao. Pathfuzz: Broadening fuzzing horizons with footprint memory for cpus. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC ’24*, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649329.3655911.
- [73] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. doi:10.1145/1993316.1993532.
- [74] Kanqi Zhang, Peinan Li, Miao Li, Xin Tian, Zelong Du, Quanchen Liu, Yongqiang Lyu, Yu Jiang, Dan Meng, and Rui Hou. Sonar: A hardware fuzzing framework to uncover contention side channels in processors. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture, MICRO ’25*, page 125–139, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3725843.3756136.
- [75] Yang Zhong, Haoran Wu, Xueqi Li, Sa Wang, David Boland, Yungang Bao, and Kan Shi. TurboFuzz: FPGA-accelerated hardware fuzzing for processor agile verification. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–16. IEEE, 2026. doi:10.1109/HPCA68181.2026.11408489.
- [76] Melisande Zonta, Nora Hinderling, and Shweta Shinde. Xray: Detecting and exploiting vulnerabilities in arm axi interconnects. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, March 2025. doi:10.23919/DATE64628.2025.10992968.
- [77] Melisande Zonta-Roudes, Andres Meza, Nora Hinderling, Lucas Deutschmann, Francesco Restuccia, Ryan Kastner, and Shweta Shinde. expect: On the security implications of violations in axi implementations. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, ICCAD ’24*, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3676536.3676844.