Beyond Fuzzer Islands: CPU Fuzzing via Smart Coordination

Yuchen Hu[1,2], Jialin Sun[1,2], Yushu Du[2], Renshuang Jiang[3], Ning Wang[4], Weiwei Shan[1,2], Xinwei Fang[5], Xi Wang[1,2], Nan Guan[4], Zhe Jiang[1,2†]
[1]Southeast University, China [2]National Center of Technology Innovation for EDA, China
[3]National University of Defense Technology, China [4]City University of Hong Kong, Hong Kong [5]University of York, UK
[†]Corresponding author: zhejiang.uk@gmail.com

Abstract

Despite recent progress, every CPU fuzzer explores only a narrow slice of the vast micro-architectural state space due to its fixed mutation and feedback biases. While different fuzzers thus excel in disjoint regions, naïve combination fails because of conflicting strategies, seed pollution, and early saturation. LiFU introduces micro-architecture-aware orchestration that dynamically profiles heterogeneous fuzzers, detects complementary strengths, decreases harmful interactions, and steers each fuzzer in real time using coverage and bug feedback, augmented by semantic seed triage. Evaluated on the BOOM core, LiFU achieves 93.4% line, 43.6% FSM, and 95.7% condition coverage (90.1%, 75.0%, 78.3% on Rocket) with around 40% fewer tests than the best standalone fuzzer, consistently closing long-standing verification gaps in modern CPU.

1 Introduction

Modern hardware systems continue to scale in complexity, integrating heterogeneous intellectual properties (IPs) and advanced processors, ranging from in-order designs (e.g., Rocket [1]) to out-of-order superscalar cores (e.g., BOOM [2]). This scaling dramatically expands the architectural state space and micro-architectural interactions that require validation, encompassing pipeline speculation, memory coherency and privilege transitions. With post-silicon bug fixes often incurring billion-dollar costs and pre-silicon verification already accounting for over 70% of the total design effort [3–9], enhancing verification efficiency and depth is paramount for ensuring functional correctness and security of the CPU cores.

In recent years, hardware fuzzing has emerged as a scalable alternative to formal verification [10, 11] and constrained random testing [12, 13] for RTL designs [14–21]. Inspired by software fuzzing, hardware fuzzing techniques generate instruction streams and refine them via coverage feedback from DUT simulation. Early black-box tools (RFUZZ [22], DirectFuzz [23]) treated streams as raw bit-sequences, scaling well but converging quickly due to poor architectural visibility. Later hardware-aware fuzzers introduced register coverage (DifuzzRTL [24]), explicit dependency chains (Cascade [25]), formal counterexample seeding (HyPFuzz [26]), and assembly synthesis guided by language model (LM) (GenHuzz [27]).

Despite continuous progress, no existing fuzzer simultaneously delivers high instruction diversity (e.g., rapidly hit basic arithmetic operations) and deep multi-cycle dependency chains (e.g., trigger pipeline corner cases). On Rocket, even state-of-the-art (SOTA) tools reach only 80% coverage, leaving substantial portions of the pipeline untested [27]. This phenomenon occurs because current fuzzers suffer from weaknesses that leave distinct portions of the micro-architectural state space unexplored. Closer inspection shows that these tools often converge early because certain micro-architectural behaviours are statistically or structurally hard to trigger [28]. For example, exposing a data-forwarding hazard typically

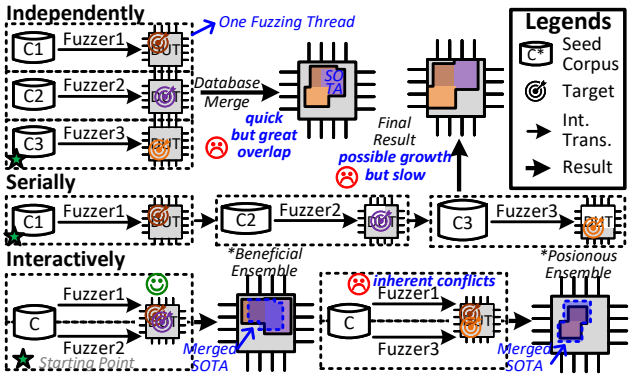

Figure 1. Independent, serial, and interactive parallel execution of multiple fuzzers. While interactive execution enables real-time collaboration, certain fuzzer pairs exhibit severe negative interference.

requires a carefully ordered sequence such as ADD ⟶ STORE ⟶ LOAD, where the store and load access the same address within a few cycles. [1] Longer hazards, such as forwarding through multiple pipeline stages or dependent branches, reduce the likelihood exponentially, making them virtually unreachable under random mutation. Learning-based generators [18, 27, 29–32] mitigate early randomness by producing semantically coherent programs, but rare multi-cycle interactions remain unreachable.

Biasing mutations [20, 27, 33] toward dependency-heavy patterns improve depth yet sacrifice diversity, systematically missing uncorrelated bug classes. In contrast, more stochastic fuzzers [22, 24] maintain higher diversity and occasionally uncover unexpected behaviours, but they struggle to penetrate the structured semantic dependencies. This diversity, however, has led to a natural intuition: combine multiple fuzzers to leverage their strengths [34]. However, naïve composition rarely works [35–37]. As shown in Figure 1, running fuzzers independently and merging corpora post-hoc results in the loss of cross-guided exploration advantages; running them sequentially is slow and ordering-sensitive. Interactive parallel execution enables runtime seed sharing and can yield more balanced performance. However, we observe empirically that some fuzzer combinations accelerate coverage, while others are poisonous, degrading performance, even below a single fuzzer, due to incompatible heuristics or mismatched abstraction levels. These behaviours indicate that the challenge is no longer merely "design a better fuzzer," but how to manage multiple fuzzers.

Contributions. We introduce LiFU, a verification framework orchestrating fuzzing engines and formal methods via feedback-driven coordination. LiFU detects coverage growth conditions and selects abstraction levels using lightweight heuristics and semantic-aware LLM-powered analysis. On Rocket, LiFU achieves 90.1% line, 75.0% FSM, 78.3% condition coverage (versus individual Cascade [25] (the

DAC 2026, Long Beach, USA
2026. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/10.1145/3770743.3804351

[1]Assuming uniform instruction sampling over 1146 opcodes and 32 registers, the probability of generating that exacts three-instruction dependency chain by chance is roughly $(1/1146)^3 \times (1/32)^2 \approx 6 \times 10^{-12}$.

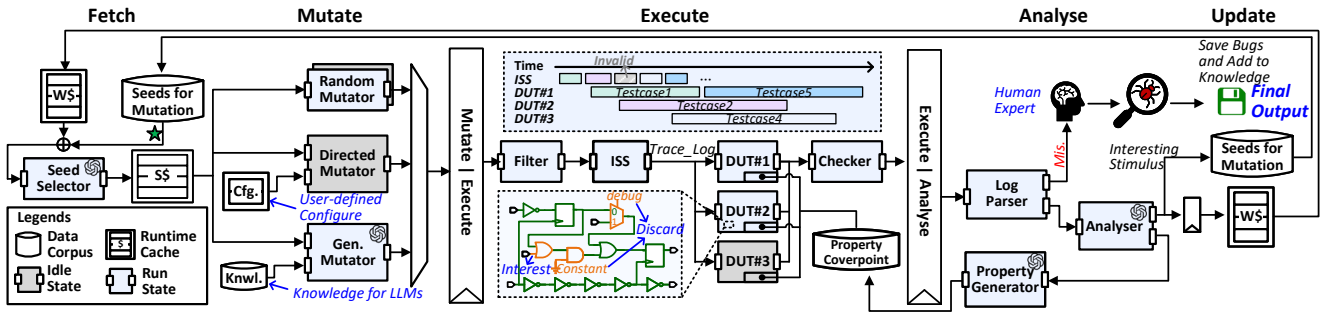

Figure 2. The LiFU Framework Overview: The five-stage pipeline begins with the seed fetch. Phase I (Fetch) selects seeds from a hybrid corpus (initially, prior instruction/program sets). Phase II (Mutate) applies a suite of hybrid mutators to produce candidate testcases. Phase III (Execute) runs the ISS ahead as a fast predictive oracle, generating a golden trace/log; DUTs follow asynchronously, streaming partial traces to the Checker for live differential comparison (execution timeline inset). Phase IV (Analyse) parses logs, evaluates uncovered points, and identifies interesting stimuli and mismatches with human-in-the-loop triage. Finally, Phase V (Update) saves confirmed bugs, updates the knowledge base, and feeds high-value seeds back for the next iteration.

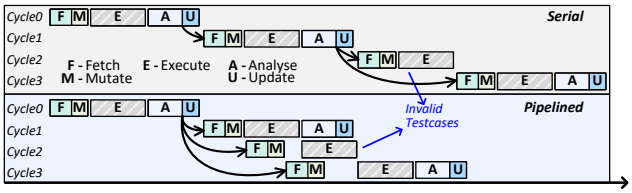

Figure 3. The Fuzzing Pipeline.

SOTA fuzzer): 89.2%, 60.0%, 77.5%). On BOOM, it attains 93.4% line, 43.6% FSM, 95.7% condition (versus individual Cascade: 83.2%, 11.6%, 90.8%, with 4–5× faster convergence. When compared to the merged coverage database obtained by combining the best results from current the SOTA fuzzers Cascade [25] and ProcessorFuzz [16], LiFU achieves relative gains of 10.9%, 221.5%, 4.9% in line, FSM, condition coverage on BOOM. It cuts time-to-90% coverage by more than 50%, boosts final coverage by around 15% and finds six bugs.

2 Background and Motivation

CPU fuzzers exhibit complementary but asymmetric strengths. Random engines (e.g., RISCV-Torture [38], RFUZZ [22]) provide broad testcase diversity, whereas directed fuzzers (e.g., HyPFuzz [26], WhisperFuzz [20]) are designed to drive long dependency chains and explore deeper in micro-architectural behaviours. Although such complementary strengths suggest that ensembles should outperform individual fuzzers, the performance of existing ensemble fuzzing is insufficient due to lack of effectiveness and efficiency.

Effectiveness refers to the ability that an ensemble consistently generate high-value test cases without interference between its fuzzes. Achieving this is non-trivial: when multiple fuzzzers share seed, their interplay can inadvertently hinder progress.

HyPFuzz couples a formal verifier with TheHuzz [14], sharing their seeds. On Rocket, its adaptive proof-time budgeting policy yields 41× faster architectural-state coverage than TheHuzz alone. However, switching to fixed time slices (common practice for multi-component system) lets the formal engine to dominate trivial proofs, hindering progress; the speed-up decreases by 75% and 107 unnecessary context switches waste 18% of the total budgets [26].

Similar observations are shown when combining a targeted fuzzer with random mutators. WhisperFuzz targets data-dependent timing channels. For example, `DIVUW` shows a 14–56-cycle gap between divisors 0/1, which activates WhisperFuzz's leakage analyser and yields highly focused seeds. When these seeds are passed directly to a random fuzzer that mutates raw 32-bit streams, two issues arise: many seeds induce division-by-zero traps or unstable branches, producing many unexecutable mutants, and random byte

Table 1. Idealised throughput on Rocket core (unit time is 20 minutes)

Config.	Testcases / Unit Time	Coverage Gain
Random baseline	20	5.78%
Serial monolithic	3	6.71%
LiFU pipelined	8	10.28%

mutation rarely preserves the register dependencies WhisperFuzz relies on, collapsing its gradient signal. On CVA6 [39], this uncoordinated ensemble drops timing-path coverage from 39.6% to 29–33%, misses 4/12 timing bugs, and increases runtime by 1.5–2.2×.

> ***Requirement for effectiveness.*** Effective ensembles demand continuous assessment of which seeds benefit which engine, plus dynamic resource allocation and selective seed transformation.

Efficiency is defined as the capacity of the fuzzing system to execute the maximum number of test cases and reach coverage goals as quickly as possible, by fully utilising available computational resources. The challenge is that any added coordination (as needed for effectiveness) will introduce overhead.

Classic fuzzers use a serial loop, so any coordination barrier would block the entire pipeline and directly compound the already-severe load imbalance of slow RTL simulation: fast cores sit idle waiting for the slowest simulator and for coordination decisions. A central scheduler naïvely onto the serial fuzzing loop would make coordination overhead directly worsen the imbalance.

We reconstruct the pipeline into an asynchronous five-stage one (Fetch → Mutate → Execute → Analyse → Update; Figures 2 & 3). All coordination (seed ownership, heuristic priority, etc.) is handled in lightweight, lock-free stages that execute in parallel with the slow Execute stage, completely hidden in its slack. High-value seeds and partial traces flow instantly without stalling any fast stage. Result on Rocket (Table 1) shows, the pipelined design processes 2.7× more testcases per unit time and achieves roughly 77% greater coverage gain than a traditional serial loop.

> ***Requirement for efficiency.*** Coordination must never sit on the critical path of the no-idle-stage pipeline, requiring analysis and scheduling to guide the next few cycles.

LLM-aided fuzzing: a step forward. The two requirements above show that effective and efficient ensembles demand semantic awareness to avoid poisonous interactions and predictive, non-blocking scheduling that existing heuristics fail to provide. LLMs offer a promising path forward. Because they can interpret RTL behaviour, infer dependency patterns, and anticipate how different fuzzing engines will react to a given seed, they have the potential to enable

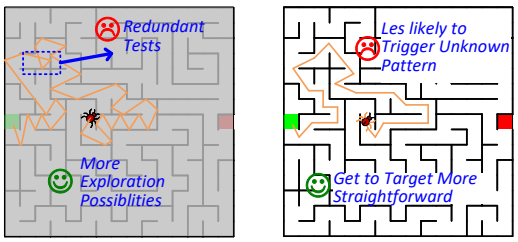

Figure 4. Exploration dynamics in processor fuzzing. Grey-box fuzzing (left) receives only architectural feedback (registers, memory, ISA-level coverage). White-box fuzzing (right) has direct visibility into the RTL source and internal pipeline/functional-unit states.

compatibility-aware seed exchange and eliminate the poisonous interactions that plague existing ensembles. Meanwhile, their ability to reason about micro-architectural structures and forecast a seed's possible coverage payoff opens the door to predictive, pipeline-friendly scheduling that keeps every stage saturated without introducing stalls. Together, these abilities allow LLMs to act as the coordinating layer that current ensemble fuzzers lack.

3 LiFU: The Framework Pipeline

LiFU introduces a staged workflow that generates, filters, executes, and analyses testcases. The framework combines simulation accuracy with selective LLM assistance. LLMs contribute semantic cues where helpful, while structured mutators, fast filtering, and differential checking ensure correctness and stability. The following subsections describe each stage of the workflow. The Fetch stage selects high-value seeds using novelty and historical performance (Section 3.1). Section 3.2 explains how LiFU applies hybrid-level mutation strategies. Section 3.3 outlines the Execute stage, including fast ISS pre-runs, filtering for non-progressing tests, and RTL simulation with mismatch and coverage logging. The Analyse stage, where mismatches are classified and unreachable coverpoints are pruned, is introduced in Section 3.4. Finally, the data from the Analyse stage will be inserted into the data corpus (Section 3.5). Together, these stages form a feedback-driven loop that expands coverage efficiently and exposes bugs missed by traditional fuzzers.

3.1 Fetch: Better Seeds to Start

The Fetch stage governs how LiFU selects and schedules input programs for mutation and execution. Its goal is to transform a static queue into an adaptive feedback-driven process that continuously reallocates effort toward the most promising test directions.

Seed Corpus. The pipeline begins with a curated seed corpus derived from standard RISC-V test suites [18], e.g., `riscv-dv` [40] and `riscv-tests` [41]. These corpora provide structurally valid and semantically diverse instruction sequences that cover common ISA behaviours, serving as a reliable bootstrap for the early fuzzing stages. This avoids the cold-start inefficiency observed in random initialisation, where initial inputs often fail trivial decode or privilege checks and waste simulation cycles.

Weight Cache (W$). Each executed seed updates a runtime *Weight Cache* (W$) , which records historical metadata including coverage contribution, mutation lineage, and observed novelty. Rather than serving as a static priority queue, W$ acts as a dynamic scoring oracle that evolves alongside the campaign (see Section 3.5). Seeds that repeatedly unlock new coverage are reinforced, while those exhibiting diminishing returns are decayed in priority. This persistent weighting mechanism mitigates the common "early winner"

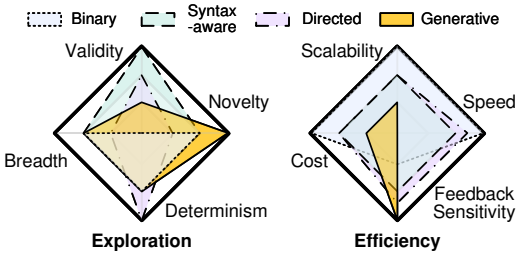

Figure 5. Qualitative trade-offs of mutation strategies, grouped into exploration and efficiency axes: novelty (new micro-architectural behaviours), breadth (state-space volume), validity (runnable tests), determinism (predictability and reproducibility); feedback sensitivity (response to precise coverage gaps), speed/cost/scalability (throughput and resource demands). LLM-based generative mutation, by reasoning over the actual RTL, achieves superior novelty and feedback sensitivity at the cost of some determinism.

effect seen in coverage-guided fuzzers, where a few high-yield seeds dominate mutation cycles and stall global exploration.

Seed Selector. Guided by W$, the Seed Selector constructs an ordered queue that balances exploitation and exploration. Each seed receives a composite score reflecting both its cumulative coverage potential and its structural novelty relative to the current corpus. The Selector consumes the ranked list, then performs *adaptive mutator dispatching*: high-value seeds with complex instruction dependencies are routed to LLM-guided mutators that preserve semantic consistency, while low-performing or stagnant seeds are handed to binary-level mutators for disruptive exploration.

3.2 Mutate: Hybrid Test Case Generation

Mutation lies at the core of every fuzzer, yet existing hardware fuzzers almost exclusively operate at the architectural surface: random bit flips, instruction splicing, or grammar-guided changes. These quickly saturate visible architectural states but leave micro-architectural corner cases untouched, because they cannot see the data and timing dependencies hidden inside the RTL (Figure 4, left).

Semantic-Aware Generative Mutation. LiFU breaks this barrier with a micro-architecture-aware generative mutator powered by LLMs that have ingested the processor's full design hierarchy, coverage gaps and gap-related RTL codes. Instead of blind edits, LLMs interpret uncovered functional blocks (e.g., an unexercised carry chain or shifter bypass) and synthesises short, semantically precise instruction sequences that activate the missing logic (typically 3–30 instructions; see Listing 1). As Figure 5 illustrates, this white-box strategy trades some speed and determinism for dramatically higher novelty and feedback sensitivity compared with traditional binary, syntax-aware, or directed grey-box mutations.

```
1 li   t0, 0xF0F0F0F0   # Alternating nibbles
2 bset t0, t0, zero, 31 # Set sign bit
3 bext t1, t0, zero, 4  # Extract low nibble
4 bins t0, t1, t0, 8, 12# Insert into upper half
```

Listing 1. LLM-generated testcase targeting Rocket ALU paths

The generated fragments are never run standalone. We insert them into valid base programs drawn from the regular corpus, preserving liveness while injecting targeted behaviour. Conventional low-level and syntactic mutators continue to run in parallel, supplying raw diversity and preventing entrapment in local optima. The resulting hierarchy (random and syntactic mutations for breadth, directed mutations for validity, LLM guidance for depth) keeps coverage growing long after pure grey-box techniques have plateaued, without sacrificing their raw throughput advantage.

Algorithm 1: Filter for Non-Progressing Testcases

Input: Testcase t, ISS simulator, timeout $T_{\max}$, min progress δ, max repair attempts R
Output: (Valid, t') where Valid $\in \{\top, \bot\}$ and t' is the (possibly repaired) testcase

$state_0 \leftarrow$ InitState() $pc_{\text{hist}} \leftarrow \emptyset$, $cycle \leftarrow 0$ $stuck\_pc \leftarrow \bot$, $stuck\_state \leftarrow \bot$
while $cycle < T_{max}$ ***and*** ¬Timeout **do**
  $state \leftarrow \text{Step}_{ISS}(t, state)$; // single-cycle step
  $pc \leftarrow$ GetPC($state$)
  if $pc \in pc_{hist}$ ***and*** ArchDelta($state_0, state$) $< \delta$ **then**
    $stuck\_pc \leftarrow pc$, $stuck\_state \leftarrow state$;
    break ; // loop without architectural progress
  end
  $pc_{\text{hist}} \leftarrow pc_{\text{hist}} \cup \{pc\}$;
  $cycle \leftarrow cycle + 1$;
end
if $stuck\_pc = \bot$ **then**
  return $(\top, t)$; // normal progressing testcase
end
$t' \leftarrow t$ **for** $attempt = 1$ ***to*** R **do**
  $t'' \leftarrow t'$; // Repair at $stuck\_pc$
  if $attempt \leq R/2$ **then**
    // Insert NOPs / privilege-neutral instructions
    $t'' \leftarrow$ InsertNopsAfter($t', stuck\_pc$, Rand(1, 8))
  else
    // Inject reg.-forwarding breakers or CSR writes
    $reg \leftarrow$ RandLiveOutReg($stuck\_state$) ;
    $instr \leftarrow$ GenForwardingBreaker(reg);
    $t'' \leftarrow$ ReplaceOrInsertAfter($t', stuck\_pc, instr$)
  end
  if QuickSim($t'', stuck\_state, 64$) *shows* $\geq \delta$ *progress* **then**
    return $(\top, t'')$; // repair succeeded
  end
  $t' \leftarrow t''$; // keep best attempt so far
end
return $(\bot, t)$; // failed to repair within budget

3.3 Execute: Asymmetric Simulation with ISS Pre-Run

It is necessary to get feedback for the fuzzing process in the Execute stage. RTL simulation provides the highest micro-architectural fidelity but is prohibitively slow, often several orders of magnitude slower than instruction-set simulation. To balance precision and throughput, LiFU adopts an *asymmetric simulation strategy* that decouples reference generation from detailed RTL execution. Drawing from techniques in prior work [42], a fast Instruction Set Simulator (ISS) (e.g., Spike) runs ahead as a golden oracle to produce architectural traces, while multiple RTL DUT instances execute asynchronously using these traces as reference checkpoints.

Early Filtering for Efficiency. Although each testcase undergoes standard syntactic compilation, correctness at the ISA level does not guarantee meaningful progress at runtime. Random or mutation-based generators often produce *non-progressing* tests—programs that loop indefinitely, execute very few static instructions, or repeatedly stall due to dependency or privilege traps. Such deadlock-like behaviour wastes valuable RTL cycles and distorts coverage feedback, since most instructions remain unexecuted. To mitigate this, the Execute stage incorporates a lightweight *pre-filter* (Algorithm 1) that evaluates each candidate through bounded static and dynamic checks, including loop-depth analysis, control-flow graph traversal, and minimal progress estimation via a short ISS pre-run. Instead of complete rejection, the filter can also trigger *repair actions*, such as inserting bounded-loop exits or safe termination sequences, when a high-value seed (as indicated by its Weight Cache score) is blocked only by trivial control-flow issues.

Checker and Reachability Refinement. Upon completion of RTL simulation, the Checker module performs instruction-granular differential check between DUT outputs and the ISS golden trace. Comparisons span register states and memory transactions, with mismatches (e.g., register value mismatch) captured in a structured **mismatch log**. This log, enriched with context such as triggering instructions and affected modules, is forwarded to the Analyse stage for classification and guided repair. For cases with no direct mismatches, the Checker merges coverage data across runs and performs micro-architectural **reachability refinement** (Algorithm 2) to exclude constant or empirically unreachable RTL coverpoints (e.g., tied-off debug ports or inactive configuration lines). [2]

Algorithm 2: Reachability Analysis for Uncovered Point

Input: RTL coverpoint c, Trace set $\mathcal{T}$, signal driver map D
Output: ReachClass(c) $\in$ {unreachable, unlikely, reachable}

$sig \leftarrow$ GetSignal(c) ; // signal driving coverpoint
if $sig \in D$ ***and*** $D(sig) = constant$ **then**
  return *unreachable* ; // hard-wired 0/1
end
$vals \leftarrow \{\text{Read}(sig, tr) \mid tr \in \mathcal{T}\}$
if $|vals| = 1$ ***and*** $\mathcal{T}$ *spans* $\geq N$ *diverse seeds* **then**
  return *unlikely* ; // empirically constant
end
return *reachable*

Algorithm 3: LLM-Guided Property Generation

Input: Uncoverpoint u, fused coverage C, knowledge base K, prompt template P
Output: Set of synthesised properties $\mathcal{P}$

$context \leftarrow$ ExtractSnippet(C, u) ; // uncoverpoint log
$prompt \leftarrow P.format(context, K)$; // grounded prompt construction
$\hat{\mathcal{P}} \leftarrow$ LLMQuery($prompt$) ; // initial generation
$\mathcal{P} \leftarrow \emptyset$ **foreach** $\hat{p} \in \hat{\mathcal{P}}$ **do**
  if ValidateSyntax($\hat{p}$) ***and*** SimCheck($\hat{p}, C$) $\neq \bot$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \hat{p}$; // add if syntactically valid and consistent
  end
end
if $|\mathcal{P}| = 0$ **then**
  $\mathcal{P} \leftarrow$ RefinePrompt($prompt$) ; // refinement
end
return $\mathcal{P}$

3.4 Analyse: Fusion with LLM-Guided Interpretation

Log Parsing and Coverage Fusion. The Analyse stage begins with a log parser ingesting Checker artefacts and simulator coverage databases (e.g., URG reports): ISS/DUT traces, mismatch logs, and metrics (uncovered points and architectural events). For each testcase t, it computes $\Delta\text{cov}(t)$ (new-bit fraction), extracts cycle-accurate mismatches with severity, and ranks uncoverpoints by criticality, proximity, and difficulty. Metrics are normalised across DUTs for consistent scoring, quantifying completeness and exposing under-tested regions (e.g., rare instructions or pipeline stalls) to enable targeted reseeding and LLM-guided synthesis.

LLM-Guided Property Generation and Interpretation. This stage fuses coverage data with LLM-driven analysis to generate temporal properties for hard-to-reach coverpoints flagged as uncovered by simulator, e.g., redundant FSM transitions or condition sequences. The LLM examines fused logs and prioritised uncoverpoints via targeted prompts (e.g., "Synthesise an SVA property requiring FSM state S5 to S2 after a 3-cycle stall and branch

[2] This prevents such artefacts from misleading the whole fuzzing process into futile search directions. Points labelled as "unlikely reachable" are withheld from feedback, while those remaining uncertain may later escalate to formal reachability checks. This layered processing ensures that coverage metrics reflect genuine design behaviour rather than simulation noise or unreachable artefacts.

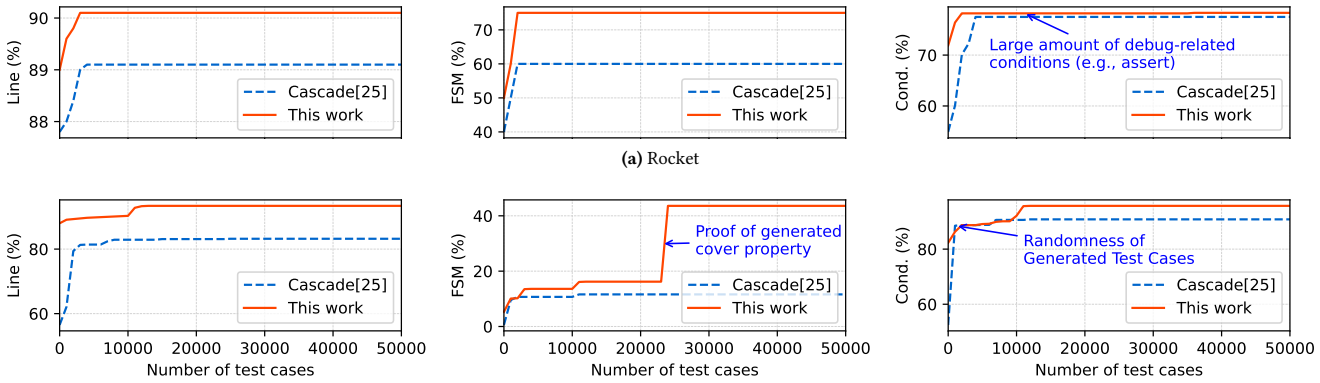

Figure 6. Coverage benchmark on the CPUs. LiFU continuously outperforms Cascade [25] in all coverage metrics, with BOOM achieving large FSM coverage gains from the formal proof of LLM-generated properties. The largest gap is in BOOM FSM coverage, where LiFU has LLMs turn uncoverpoints into formal properties to prove rare controller transitions. On Rocket, condition coverage plateaus early for both tools because many remaining conditions are unreachable debug-related logic. On BOOM, Cascade occasionally edges ahead in condition coverage simply because of the randomness of generated testcases.

Table 2. Weight Cache (W$) Scoring Factors

Factor	Definition	Computation
$\Delta\text{cov}(t)$	New coverage contribution	$\frac{\lvert\text{cov}(t)\setminus C_{\text{prev}}\rvert}{\lvert C_{\text{total}}\rvert}$
$\text{bugsc}(t)$	Bug detection impact	$\sum \text{severity}(m)$; $\forall m \in \text{mismatches}(t)$
$\text{nov}(t)$	Structural uniqueness	$1 - \text{sim}(t, \mathcal{S}_{\text{seen}})$ (e.g., normalised edit distance)
$\text{eff}(t)$	Insight per cycle	$\frac{\Delta\text{cov}(t)+\text{bugsc}(t)}{\text{cycles}(t)}$

mispredict"), producing assertions that explain coverage gaps and guide stimulus mutation. Grounded in empirical traces and a constrained knowledge base, the LLM translates raw metrics into natural-language insights (e.g., "This bin is blocked by missing hazard resolution") while reducing hallucination. Valid properties and high-impact stimuli are prioritised for reseeding. Algorithm 3 formalises this iterative, validation-gated workflow.

Mismatch Detection and Triage. Human experts may intervene to analyse the errors, identify the causes, and check for false positives. Analysed results will be annotated , providing ground-truth labels that refine automated thresholds.

3.5 Update: Knowledge Integration

Following analysis, the Update stage archives confirmed bugs and high-priority stimuli in the knowledge base. New interesting seeds are added to the seed corpus, along with weights obtained from the priority formula that emphasises coverage deltas and novelty, thereby closing the feedback loop with minimal overhead.

W$ Update. For each testcase t, W$ computes a priority weight $w(t)$ using a multi-factor scoring function:

$$w(t) = \alpha \cdot \Delta\text{cov}(t) + \beta \cdot \text{bugsc}(t) + \gamma \cdot \text{nov}(t) + \delta \cdot \text{eff}(t) \quad (1)$$

where $\alpha+\beta+\gamma+\delta = 1$. Table 2 details each factor and its calculation. Coefficients are user-configurable or adapted via online feedback (e.g., multi-armed bandit). High-$w(t)$ testcases are promoted to the runtime cache and prioritised for reseeding.

4 Evaluation

This section presents our experimental setup, evaluation metrics, and discussions to the results.

Setup. We evaluated LiFU on two widely-used open-source RISC-V processor cores: the in-order RocketChip [1] and the out-of-order BOOM [2]. All RTL simulations were performed using Synopsys VCS (O-2018.09-SP2) [43], while formal verification was conducted with VC Formal (S-2021.09-SP2) [44]. Stimulus generation in LiFU leveraged direct calls to the OpenAI API, using GPT-4-turbo as the default model to synthesise assembly sequences. We also adopted Cascade [25] and ProcessorFuzz [16] for seed mutation.

Table 3. Final Coverage on BOOM. "Merged" means merging the database of Cascade [25] and ProcessorFuzz [16] after independent fuzzing; Percentages report the relative improvement of LiFU over the merged solution.

Method	Line	FSM	Condition
Cascade [25]	83.2	11.6	90.8
ProcessorFuzz [16]	80.3	13.55	86.79
Merged	84.2	13.56	91.2
This work	93.4 (10.9%↑)	43.6 (221.5%↑)	95.7 (4.9%↑)

All experiments were conducted on an AMD EPYC 7763 2.45GHz CPU. We compare LiFU against five SOTA hardware fuzzers: DifuzzRTL [24], ProcessorFuzz, ChatFuzz [45], Cascade, and GenHuzz [27]. Coverage (line, FSM, condition) and bug detection data were collected via VCS coverage reports and differential checking against Spike. Considering the limited simulation resources, the time budget for one testcase is 1.5 hours. The thread shall be cancelled exceeding the budget, with executed information logged.

4.1 Results and Discussions

Comparison with Individual Cascade. We compare LiFU against Cascade, currently the strongest published open-source hardware fuzzer. Across both Rocket and BOOM, our approach delivers higher final coverage and significantly faster convergence in every metric (Figure 6). On Rocket, LiFU achieves 90.1% line, 75.0% FSM, and 78.3% condition coverage, beating Cascade's 89.2%, 60.0%, and 77.5%. We further reach 89.0% line coverage using fewer than 1,000 tests, whereas Cascade needs around 5,000, and our FSM/condition curves rise far more steeply from the start. The gap grows larger on BOOM, where we obtain 93.4% line (vs. 83.2%), 43.6% FSM (vs. 11.6%), and 95.7% condition coverage (vs. 90.8%). The FSM gap is especially revealing: with the combination of formal tools, LiFU discovers 3.76× more micro-architectural controller states. Although Cascade temporarily leads in BOOM condition coverage during some intervals (roughly 1,000–8,000 tests), these brief crossings stem from random variation and Cascade's targeting in deep dependencies. Overall, our method consistently ends with greater results across

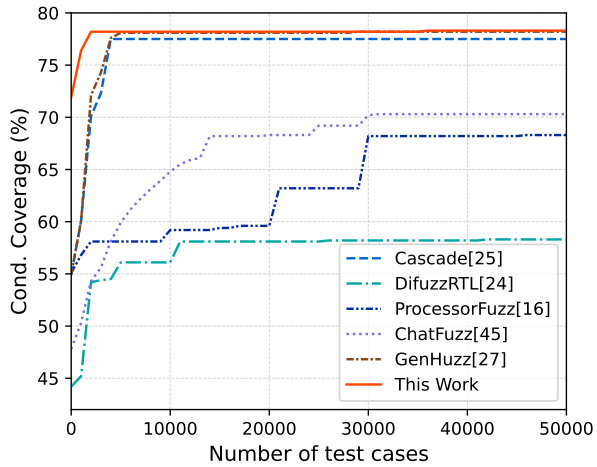

Figure 7. Condition coverage benchmark on Rocket.

all metrics (line, FSM, condition) on both designs, confirming clear and stable improvement over the SOTA individual fuzzer.

LiFU vs. Naïve Merging. Different fuzzers show complementary strengths. A simple merge of Cascade and ProcessorFuzz coverage databases on BOOM yield 84.2% line, 13.56% FSM, and 91.2% condition coverage (Table 3). These values exceed the individual fuzzer (+1.0% line, +0.4% condition). Our method LiFU finally achieves 93.4% line, 43.6% FSM, and 95.7% condition coverage under the same time budget. The large gap (relative increases of 10.9% for lines, 221.5% for FSM states, 4.9% for conditions) proves that LiFU succeeds not by loosely combining separate tools, but by tightly integrating sequence generation, micro-architectural state tracking, and coverage feedback. Naïve database merging fails to drive the deep, systematic exploration that our design consistently attains.

Benchmark against Published SOTAs. We benchmark LiFU against these published results, with Cascade included for reference[3]. As shown in Figure 7, our approach significantly outperforms all prior methods. While DiffuzzRTL exhibits rapid saturation after modest initial gains, indicating constrained exploration capacity, our method sustains strong upward progress even as the test-case count scales to 50,000 and beyond. Most strikingly, we achieve coverage comparable to or exceeding the results of all prior tools using only a small fraction, roughly 1/4 to 1/3, of their test cases. This superior efficiency in exploring diverse hardware states translates directly into accelerated bug and vulnerability detection, establishing our approach as the most effective tool for Rocket.

Coverage Gap Analysis. Although we achieve the highest coverage on both Rocket and BOOM, a small residual gap remains. Manual analysis of VCS reports shows that nearly all uncovered elements are untestable or irrelevant logic belonging to three main categories: (1) debug and verification constructs (asserts, assumes, $fatal) that only trigger on illegal states never generated by the core itself and should therefore be excluded from functional coverage; (2) severe VCS over-reporting that inflates trivial logic (e.g., BOOM's BranchMaskGenerationLogic expands simple 2–4-bit mux/priority-encoder circuitry into 420 FSM states, and less than 10% states are triggered during massive simulation processes without formal proof); and (3) unreachable redundant code introduced by design reuse—most notably Rocket ALU paths for rotate and miscellaneous instructions that are deliberately replaced in the BOOM, plus partially instantiated parametrised modules. Some of these items are waived during fuzzing execution yet still counted in final metrics, and a few remaining holes are simply unreasonable artefacts of the coverage tool rather than genuine verification gaps.

[3]Since most recent RTL fuzzers such as GenHuzz and TheHuzz remain closed-source, direct evaluation is not feasible. We therefore utilise the data reported in the [27].

Table 4. Reported bugs with LiFU in BOOM and Rocket. B* denotes bugs in BOOM, and R* denotes bugs in Rocket.

ID	Core	Bug Description	New
B1	BOOM	Misdecoding when processing single-precision operations like fcvt.l(u).s	✓
B2	BOOM	Improper floating-point state after mstatus.FS state transition	×
B3	BOOM	Incorrect Restoration of MPP field in mstatus during trap return	✓
B4	BOOM	Uninitialised CSR registers return non-deterministic values after reset	✓
R1	Rocket	Misdecoding when processing single-precision operations like fcvt.l(u).s	✓
R2	Rocket	Uninitialised CSR registers return non-deterministic values after reset	✓

4.2 Bugs Reported

Table 4 summarises the bugs identified in the BOOM and Rocket during our verification. Both cores (B1 in the BOOM, R1 in the Rocket) misdecode the single-precision floating-point-to-integer conversion instructions such as `fcvt.l(u).s`, resulting in incorrect integer outputs. A previously known defect in the BOOM (B2) causes improper handling of the floating-point register file when the `mstatus.FS` field transitions (e.g., from Off to Initial), potentially leaving stale or corrupted data in the FP registers. Another bug in the BOOM (B3) involves incorrect restoration of the `mstatus.MPP` field during trap return; the processor may update or restore this field erroneously, causing `mret` to return to an unintended privilege mode (e.g., from machine mode directly to user mode), which triggers subsequent illegal instruction faults on operations like `sret`. Also, both cores suffer from a post-reset behaviour (B4 in the BOOM, R2 in the Rocket) where certain CSR registers, including performance counters such as `scounteren` and `sscratch`, remain no-zero-initialised and return non-deterministic values, resulting in mismatches between CPU and Spike.

5 Conclusion

We present LiFU, a CPU testing framework that coordinates fuzzers and formal tools via runtime analysis and seed selection. Evaluated on BOOM, LiFU outperforms Cascade with 93.4% line, 43.6% FSM, and 95.7% condition coverage (vs. 83.2%, 11.6%, 90.8%), achieving 4–5× faster convergence and 89% line coverage in less than 1,000 tests (vs. around 5,000). It sustains gains beyond 50,000 tests, matching SOTA using 25–33% of the budget; overall, it cuts time-to-90% coverage by more than 50% and uncovers six bugs. No individual fuzzer can achieve comprehensive performance; superior coordination is essential to advance CPU verification. LiFU establishes adaptive, semantically informed orchestration as a scalable paradigm.

6 Acknowledgement

We greatly appreciate the reviewers for the helpful feedback. This work is supported by the National Natural Science Foundation of China (No. 62472086, 92464204), the Science and Technology Major Special Program of Jiangsu (No. BG2024010), and the Fundamental Research Funds for the Central Universities (No. 2242025K20013).

References

[1] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), 6–2.

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

[3] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 898–911.

[4] Kan Shi, Shuoxiang Xu, Yuhan Diao, David Boland, and Yungang Bao. 2023. ENCORE: Efficient architecture verification framework with FPGA acceleration. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 209–219.

[5] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. 2024. MEIC: Re-thinking RTL Debug Automation using LLMs. *arXiv preprint arXiv:2405.06840* (2024).

[6] Ziqing Zhang, Weijie Weng, Yaning Li, Lijia Cai, Haoyu Wang, David Boland, Yungang Bao, and Kan Shi. 2024. Hassert: Hardware Assertion-Based Verification Framework with FPGA Acceleration. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 142–154.

[7] Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, et al. 2024. Uvllm: An automated universal rtl verification framework using llms. *arXiv preprint arXiv:2411.16238* (2024).

[8] Jie Zhou, Youshu Ji, Ning Wang, Yuchen Hu, Xinyao Jiao, Bingkun Yao, Xinwei Fang, Shuai Zhao, Nan Guan, and Zhe Jiang. 2025. Insights from rights and wrongs: A large language model for solving assertion failures in rtl design. *arXiv preprint arXiv:2503.04057* (2025).

[9] Junhao Ye, Yuchen Hu, Ke Xu, Dingrong Pan, Qichun Chen, Jie Zhou, Shuai Zhao, Xinwei Fang, Xi Wang, Nan Guan, et al. 2025. From Concept to Practice: an Automated LLM-aided UVM Machine for RTL Verification. *arXiv preprint arXiv:2504.19959* (2025).

[10] Czea Sie Chuah, Christian Appold, and Tim Leinmueller. 2023. Formal verification of security properties on risc-v processors. In *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*. 159–168.

[11] Lennart Weingarten, Kamalika Datta, Abhoy Kole, and Rolf Drechsler. 2024. Complete and efficient verification for a RISC-V processor using formal verification. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[12] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. 2021. Constrained random verification for RISC-V: overview, evaluation and discussion. In *MBMV 2021; 24th Workshop*. VDE, 1–8.

[13] Muhammad Kashif Minhas, Haroon Waris, Yasir Farooq, Nasir Mohyuddin, and Sajid Baloch. 2023. Coverage-Driven and Constrained-Randomized Sub-System Level Verification Methodology for RISC-V Based SoCs. In *2023 20th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. IEEE, 80–85.

[14] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 3219–3236.

[15] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. 2023. SIGFuzz: A framework for discovering microarchitectural timing side channels. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[16] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2023. Processorfuzz: Processor fuzzing with control and status registers guidance. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 1–12.

[17] Deepak Narayan Gadde, Aman Kumar, Djones Lettnin, and Sebastian Simon. 2024. FuzzWiz-Fuzzing Framework for Efficient Hardware Coverage. In *2024 International Symposium on Electronics and Telecommunications (ISETC)*. IEEE, 1–5.

[18] Yinan Xu, Sa Wang, Dan Tang, Ninghui Sun, and Yungang Bao. 2024. PathFuzz: Broadening Fuzzing Horizons with Footprint Memory for CPUs. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[19] Fabian Thomas, Lorenz Hetterich, Ruiyi Zhang, Daniel Weber, Lukas Gerlach, and Michael Schwarz. 2024. RISCVuzz: Discovering architectural CPU vulnerabilities via differential hardware fuzzing. *https://ghostwriteattack. com/* (2024).

[20] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5377–5394.

[21] Rihui Sun, Jin Wu, Hanyin Liu, Zikang Tao, Gang Qu, Dongsheng Wang, Yongqiang Lyu, and Jian Dong. 2025. BPUFuzzer: Effective Fuzz Testing for Branching Transient Execution Vulnerabilities of RISC-V CPU. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[22] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[23] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 529–534.

[24] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1286–1303.

[25] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade:{CPU} fuzzing via intricate program generation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5341–5358.

[26] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. HyPFuzz:Formal-Assisted processor fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1361–1378.

[27] Lichao Wu, Mohamadreza Rostami, Huimin Li, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. 2025. GenHuzz: An Efficient Generative Hardware Fuzzer. In *34th USENIX Security Symposium (USENIX Security 25)*. 1787–1805.

[28] Flavien Solt, Patrick Jattke, and Kaveh Razavi. 2022. Rememberr: Leveraging microprocessor errata for design testing and validation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1126–1143.

[29] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural guided constraint logic programming for program synthesis. *Advances in Neural Information Processing Systems* 31 (2018).

[30] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.

[31] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1410–1421.

[32] Ethan Brooks, Janarthanan Rajendran, Richard L Lewis, and Satinder Singh. 2021. Reinforcement learning of implicit and explicit control flow instructions. In *International Conference on Machine Learning*. PMLR, 1082–1091.

[33] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[34] Matej Bölcskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2025. Encarsia: Evaluating cpu fuzzers via automatic bug injection. In *34th USENIX Security*.

[35] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[36] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX security symposium (USENIX security 19)*. 1949–1966.

[37] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1967–1983.

[38] RISC-V International and contributors. 2023. RISC-V Torture. https://github.com/riscv/riscv-torture.

[39] Florian Zaruba and Luca Benini. 2019. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640.

[40] Google. 2019. RISC-V DV. https://github.com/google/riscv-dv.

[41] RISC-V Software contributors. 2025. riscv-tests. https://github.com/riscv-software-src/riscv-tests. [Online; accessed].

[42] Jialin Sun, Yuchen Hu, Dean You, Yushu Du, Hui Wang, Xinwei Fang, Weiwei Shan, Nan Guan, and Zhe Jiang. 2025. ISAAC: Intelligent, Scalable, Agile, and Accelerated CPU Verification via LLM-aided FPGA Parallelism. *arXiv preprint arXiv:2510.10225* (2025).

[43] Synopsys, Inc. 2023. *VCS: Synopsys Verification Compiler System.* https://www.synopsys.com/verification/simulation/vcs.html

[44] Synopsys, Inc. 2023. *VC Formal: Synopsys Verification Compiler Formal Verification Solution.* https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html

[45] Mohamadreza Rostami, Marco Chilese, Shaza Zeitouni, Rahul Kande, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. 2024. Beyond random inputs: A novel ML-based hardware fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.