Focus Session: What the Fuzz! Pushing Beyond Randomness in Hardware Security with Generative AI

Nikhilesh Singh[†], Mohamadreza Rostami[†], Lichao Wu[‡], Chen Chen[§], Stephen Muttathil[§], Jeyavijayan Rajendran[§] and Ahmad-Reza Sadeghi[†]
[†]*Technische Universität Darmstadt,* [‡]*University of Bristol,* [§]*Texas A&M University*
{nikhilesh.singh, mohamadreza.rostami, ahmad.sadeghi}@trust.tu-darmstadt.de
{lichao.wu}@bristol.ac.uk, {chenc, stephen7929, jv.rajendran}@tamu.edu

***Abstract*—Security is a foundational requirement for modern hardware, as design vulnerabilities can undermine entire computing systems from firmware to applications. However, traditional validation techniques often fall short as processor designs grow more complex, with rich privilege mechanisms and heterogeneous components. Hardware fuzzing offers a scalable way to probe implementations by generating large volumes of tests and using execution feedback to expose unintended behavior. However, conventional fuzzing pipelines often struggle to reach deep, security-critical corner cases efficiently due to limited semantic awareness and the high cost of hardware execution.**

In this work, we investigate the effectiveness of augmenting hardware fuzzing pipelines with generative AI to address these limitations. Specifically, we explore the feasibility of AI models to (a) generate syntactically valid, semantically meaningful instruction sequences rather than relying on random mutations, (b) incorporate execution feedback to guide exploration toward deep hardware states, and (c) prioritize promising tests to reduce simulation time. Finally, we highlight open challenges and outline potential future research directions to strengthen hardware security through more intelligent, scalable validation techniques.

***Index Terms*—Hardware Security, Generative AI, Hardware Fuzzing**

I. Introduction

Secure computing rests on the trustworthiness of the hardware it runs on, which is why hardware assurance is increasingly treated as both a security imperative and a strategic priority. For instance, the U.S. CHIPS and Science Act [6] and the European Chips Act [7] explicitly elevate semiconductor capability and supply-chain trust as national and regional objectives. In parallel, organizations such as the National Institute of Standards and Technology (NIST) are expanding focused work on semiconductor security threats and mitigations [19]. This policy momentum reflects the technical trend of processors and systems-on-chip (SoCs) rapidly growing in scale and architectural diversity, with many specialized blocks and tightly optimized microarchitectural features. The resulting growth in design complexity strains traditional hardware verification, increasing the risk that subtle vulnerabilities and security-critical corner cases go undetected.

Since changes after fabrication are expensive or infeasible, pre-silicon verification is the primary opportunity to identify defects before tape-out. In practice, verification teams combine formal analysis, simulation, and emulation to build confidence in correctness and security properties. However, formal approaches can require substantial manual specification effort and may not scale smoothly to highly optimized designs, while simulation-based testing depends heavily on the quality of generated test cases and observability. These limitations are particularly acute for security, where problematic behaviors can be rare, sequence-dependent, and tied to complex interactions across privilege modes and microarchitectural state.

Hardware fuzzing complements these workflows by automating test generation and using execution feedback to guide exploration. Inspired by coverage-guided software fuzzing, hardware fuzzers generate many candidate tests, execute them on a model of the design under test (DUT), and preferentially retain those that expand the set of explored behaviors for further mutation or recombination. Recent work has adapted this paradigm to pre-silicon settings for both register transfer level (RTL)-centric and Instruction Set Architecture (ISA)-level processor targets, leveraging coverage signals and differential checking against architectural references to surface discrepancies [3], [14], [16], [17], [34]. Despite this progress, conventional hardware fuzzers face persistent bottlenecks when applied to modern processors. Low-level mutations often produce invalid or low-value programs, and coarse feedback can cause exploration to plateau once easily accessible behaviors are reached. Moreover, high-fidelity RTL simulation is slow, and deeper instrumentation reduces throughput further, limiting the number of candidates that can be evaluated and the depth of state space that can be explored within a fixed budget [25], [28].

In this paper, we examine how generative AI can address these bottlenecks by shifting fuzzing from mostly syntactic mutation toward more semantics-aware, feedback-driven search. First, learned generators can internalize ISA constraints and program structure to produce a larger fraction of executable tests. Second, generators can be refined using execution-derived signals, such as coverage and oracle outcomes, to bias generation toward behaviors correlated with deeper exploration. Third, AI-guided pipelines can restructure evaluation, for example, by using fast architectural screening and staged oracles to focus ex-

[‡]The author was associated with TU Darmstadt during the work.

978-3-9826741-1-7/DATE26/© 2026 EDAA

pensive DUT execution on the most promising candidates [37]–[39]. We focus on pre-silicon, processor-centric fuzzing where test cases are ISA-level programs and correctness is assessed using assertions and differential checking against architectural references. Within this scope, we synthesize a connected progression of recent AI-assisted fuzzers and highlight the design choices that improve validity, feedback utilization, and cost-aware exploration. Specifically, we address the following research questions.

- **RQ1:** Can generative models synthesize ISA-valid, semantically meaningful instruction programs that reduce invalid and low-value tests compared to mutation-based generation?
- **RQ2:** Can these generators learn effectively from execution feedback such as coverage metrics or oracle outcomes, to drive exploration into deeper, security-relevant behaviors?
- **RQ3:** Can staged evaluation and AI-guided prioritization reduce RTL simulation burden while preserving the ability to uncover security-critical corner cases?

In addition, we distill the key open challenges that currently block robust deployment of AI-assisted hardware fuzzing and outline research directions suggested by the progression of techniques reviewed in this paper.

The remainder of the paper is organized as follows. Section II provides background on hardware fuzzing and generative AI. A review of conventional pre-silicon fuzzing frameworks and processor-focused fuzzers is presented in Section III. In Section IV, we describe the progress of AI-assisted solutions aligned with our research questions. Section V summarizes open challenges and future research directions. Section VI concludes the paper.

II. Background

A. Hardware Fuzzing

Fuzzing is an automated testing workflow that searches for bugs and vulnerabilities by repeatedly generating inputs, executing a target, and using the observed outcomes to guide subsequent input generation. In coverage-guided fuzzing, systems such as AFL [9], libFuzzer [15], and Honggfuzz [33] maintain a corpus of *interesting* seeds and apply fast heuristic mutations, retaining inputs that expand explored behaviors. The effectiveness of this loop depends on producing inputs that are sufficiently well-formed to exercise meaningful behaviors and on extracting feedback that helps prioritize exploration rather than repeatedly revisiting easy-to-reach states.

Hardware fuzzing applies the same principle in pre-silicon verification of a design under test (DUT) by executing generated *input* on an executable model of the design, typically at the register transfer level (RTL) or via FPGA-based emulation. For processors, the inputs are ISA programs; for accelerators and system-on-chip (SoC) components, they are often protocol transactions or interface-level sequences [25], [28]. As hardware rarely fails with a clean crash signal, solutions typically rely on domain oracles such as assertion or property violations, illegal architectural states, unexpected signal activity, or divergences from a golden reference model (GRM), and they preserve traces needed to reproduce and diagnose flagged behaviors [14], [16], [27], [28]. Guidance signals extend beyond software-style coverage to include hardware-centric metrics such as toggle activity, finite-state machine (FSM) coverage, and events tied to control and status registers (CSRs), which can be used to decide which tests remain in the seed pool [25], [27]. A defining constraint for hardware fuzzing is execution cost, as high-fidelity RTL simulation is slow, and richer observability can further reduce throughput, so effective hardware fuzzers must use their simulation budget carefully [17], [34]. This makes test case validity, feedback quality, and efficient oracle integration central to practical success, and motivates pipelines that screen or prioritize candidates before committing expensive DUT runs [17], [25], [34].

B. Generative AI

Generative AI refers to models that synthesize new data, such as text or code, by learning the patterns and constraints present in training examples. Unlike discriminative models that map inputs to labels, generative models learn to produce sequences token by token, conditioned on an input context. Recent progress is largely driven by transformer-based foundation models, whose attention mechanisms support long-context generation and help maintain consistency across many interacting elements, which is essential for structured artifacts like code and assembly-like text [12], [35]. In practice, these models are adapted to specialized domains via supervised fine-tuning on task-specific corpora, and can be further steered with alignment methods that incorporate reward or preference signals when requirements are difficult to fully specify [12], [20], [21].

These capabilities are a natural fit for fuzzing, where the quality of generated test cases directly determines how efficiently a campaign explores the state space. Hardware test cases in particular are tightly constrained as instruction programs and interface transactions must respect encoding rules, privilege and CSR access policies, alignment, and protocol semantics, otherwise expensive RTL executions are wasted on trivially invalid tests [25], [28]. Generative models can internalize these constraints to produce higher-validity, more semantically meaningful tests than byte-level mutation, and they can be refined using execution-derived feedback so that generation increasingly targets behaviors correlated with exploration progress or oracle triggers. Moreover, learned scoring or preference-based refinement can rank or filter candidates to reduce the number of DUT runs required, reserving high-fidelity simulation for inputs that are most likely to expand coverage or expose discrepancies [37]–[39].

III. Related Work

Hardware fuzzing borrows the core idea of coverage-guided fuzzing from software [9], [15], [33], namely that feedback from executed tests can steer a corpus-driven search toward behaviors that would be unlikely under unguided random testing. In the hardware setting, this general recipe must be adapted to slower execution substrates and to failure signals

that are typically oracle-based using assertions, invariants, or reference mismatches rather than crash-based [25], [28]. As a result, the practical performance of a hardware fuzzer is often dominated by how efficiently it generates valid test cases, how informative its feedback is under limited observability, and how effectively it amortizes the cost of simulation.

Existing pre-silicon frameworks [5], [10], [16], [26], [27], [30], [37]–[39] span multiple execution strategies and visibility assumptions. RFuzz [17] demonstrates FPGA-accelerated fuzzing for RTL by adding instrumentation that enables AFL-style feedback loops, trading throughput gains against instrumentation overhead and toolchain coupling. A complementary line of work compiles RTL into a high-performance cycle-accurate software model [34], enabling reuse of software-style instrumentation and simplifying experimentation. Verilator [36] is frequently used as the underlying compiler and simulator in such pipelines. Surveys and systematizations [25], [28] further categorize fuzzers by observability, ranging from black-box setups to grey-box designs that leverage selected internal signals for guidance, and white-box approaches that assume deeper RTL access at higher cost.

Processor-centric fuzzers focus on ISA-level program generation and typically rely on differential checking against architectural references to expose inconsistencies. TheHuzz [16] executes instruction sequences on RTL while collecting multiple coverage signals, but it can be limited when the generation lacks semantic structure. DifuzzRTL [14] and ProcessorFuzz [3] incorporate processor-aware feedback, including control and status behavior, to guide exploration more effectively, while MorFuzz [41] improves exploration by combining semantics-aware seeding with runtime-guided morphing. More recent systems further extend reach by integrating formal components, security-oriented objective functions, or long-program construction, each introducing additional complexity and scalability tradeoffs [4], [13], [30]. Collectively, this body of work has advanced pre-silicon fuzzing, but it also highlights persistent bottlenecks in validity, long-horizon test structure, and RTL-limited throughput, which motivate learning-guided generation and prioritization.

IV. AI-driven Hardware Fuzzing

In this section, we trace the progress of AI-assisted processor fuzzers that form a clear progression in how learning is integrated into the fuzzing loop. An early assembly-level system, Hardware Fuzzing Loop (HFL) [38] demonstrates that a trainable generator can reduce invalid tests and improve exploration when updated from execution feedback. The subsequent work, GenHuzz [37], builds on this by strengthening long-horizon program synthesis and shaping feedback to better reach deep, sequence-dependent behaviors. Finally, GoldenFuzz [39] restructures refinement to reduce dependence on slow RTL execution by using a fast architectural oracle for early learning, while preserving a DUT-grounded stage for discrepancy discovery. Together, these systems provide a chained perspective on our research questions by progressing from improved test validity (RQ1), to feedback-driven deep exploration (RQ2), and finally to cost-aware staging that reduces RTL burden without sacrificing bug-finding capability (RQ3).

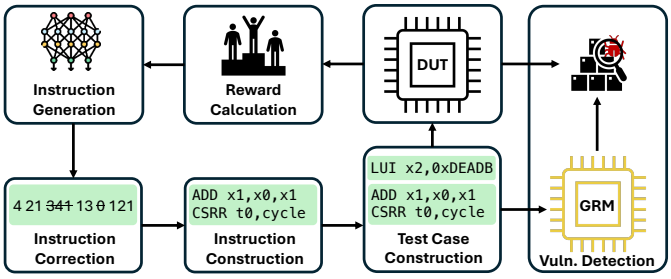

Fig. 1: HFL [38] workflow overview. A multi-head LSTM generator synthesizes assembly-level programs and is refined via coverage-driven reinforcement learning with instruction masking and reset mechanisms, with ISA-level correction before DUT execution and GRM-based differential checking to surface bug candidates.

A. RQ1: Closing the Generation-Feedback Loop

A recurring failure mode of mutation-heavy processor fuzzing is that deep corner cases rarely emerge from small, local edits. Useful tests typically require coordinated instruction sequences that satisfy ISA constraints while establishing specific control and data conditions. When the generation is dominated by random mutation, many candidates are invalid, redundant, or terminate too early to meaningfully exercise the design. HFL [38] addresses this gap by making generation itself learnable and by operating over assembly-level structure rather than unconstrained bytes.

As shown in Fig. 1, HFL uses a multi-head LSTM [31] instruction generator that decomposes instruction synthesis into coordinated fields such as opcode, registers, immediates, and special fields like CSR identifiers or branch targets. This structure enables fine-grained control over instruction formation and better captures intra-instruction dependencies. HFL then couples generation with a reinforcement learning [32] loop driven by hardware execution feedback, primarily RTL simulation coverage, to bias generation toward sequences that explore deeper internal states. Naive coverage-reward optimization can induce reward hacking in reinforcement learning [32] and unstable updates, especially when improving one instruction type destabilizes others. To mitigate this, HFL introduces an instruction mask that selectively enables updates only for the heads relevant to the chosen instruction class, reducing parameter interference. It also introduces a reset module that monitors coverage growth and partially reinitializes the generator when learning stagnates, while preserving learned intra-instruction semantics to escape local optima [38].

Taken together, HFL provides evidence that structured, learned generation combined with feedback-driven refinement can reduce wasted tests and improve exploration efficiency relative to purely heuristic mutation, directly supporting RQ1. In its evaluation, HFL achieves higher coverage with fewer than 1% of the test cases compared to prior work, and its

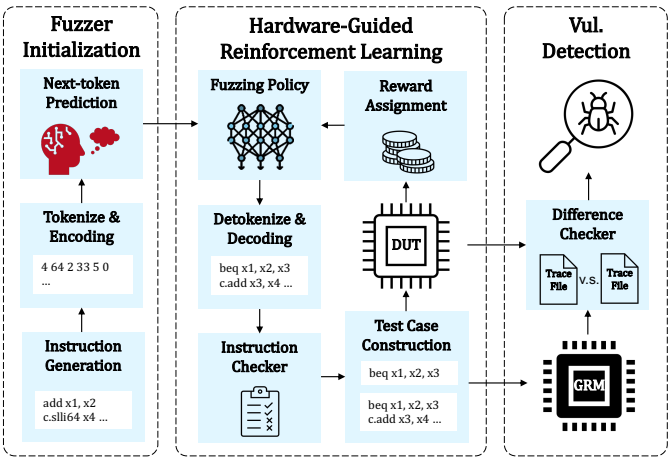

Fig. 2: GenHuzz [37] workflow overview. A transformer-based generator synthesizes long instruction programs and refines generation using hardware-guided feedback that incorporates exploration, validity, and diversity, while DUT–GRM trace mismatches flagged by a difference checker guide bug triage.

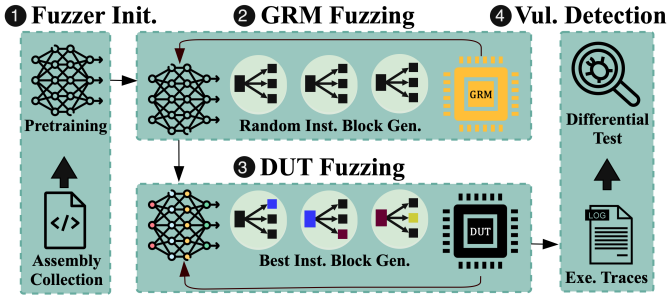

Fig. 3: GoldenFuzz [39] workflow overview. The generator is refined in a fast GRM stage to learn executable instruction blocks, then evaluated in a DUT stage to pursue coverage growth and microarchitecture-dependent discrepancies. Preference-based updates promote diversity, and DUT-GRM trace mismatches flag bug candidates.

coverage prediction network achieves 94-97% accuracy across multiple RISC-V cores including RocketChip [8], BOOM [43] and CVA6 [42]. HFL also identifies previously unknown bugs and vulnerabilities across tested designs [38]. At the same time, the approach highlights remaining issues, since scalar-reward reinforcement learning can still drift toward repetitive patterns and limited context modeling makes it difficult to sustain coherent intent over long programs.

B. RQ2: Capturing Program Semantics in the Fuzzing Loop

The limitations above are especially concerning for security, where interesting behaviors often depend on longer structures, such as initialization, privilege transitions, and delayed interactions between configuration and later memory accesses. If a generator only captures short-range patterns, it struggles to reliably build these dependencies, limiting its ability to reach deep architectural states. The follow-up research to HFL in the AI-driven hardware fuzzing domain, GenHuzz [37] addresses this gap by using a long-context generator and explicitly treating instruction generation as program synthesis over extended histories.

As illustrated in Fig. 2, GenHuzz employs a transformer-based [35] generator trained on assembly corpora and then refined in a hardware-guided loop. For bug and vulnerability discovery, a difference checker compares DUT execution traces against a golden reference model (GRM), and flags mismatches for triage (Fig. 2). After generating batches of test cases, the DUT executes each one and produces coverage metrics that form the core of the reward signal. Beyond coverage, GenHuzz incorporates validity and diversity components into the reward to encourage syntactically correct and semantically sound programs while mitigating collapse into a small set of easy-to-score patterns. Rather than optimizing only for more coverage, its feedback is shaped to jointly encourage executability and sustained exploration, incorporating explicit validity and diversity signals to avoid mode collapse. When coverage growth saturate, GenHuzz triggers a reset module that partially reinitializes the model parameters while preserving stable intra-instruction knowledge, allowing the policy to escape local optima without forgetting basic syntax and structure [37].

This design strengthens the answer to RQ2 by showing that long-context generation, coupled with feedback shaping, can more reliably produce coherent programs that reach deeper, security-relevant behaviors [37]. Empirically, GenHuzz outperforms strong baselines across three widely studied RISC-V cores [8], [42], [43]. It identifies 10 previously unknown vulnerabilities, five with CVSS severity scores exceeding 7.3 [18]. Several of the discovered issues require carefully coordinated multi-instruction sequences to manifest [37], which prior work would not be able to achieve. However, the throughput bottleneck persists because DUT execution and coverage collection remain in the refinement loop, motivating pipelines that decouple early learning and screening from expensive RTL [37].

C. RQ3: Breaking the RTL Bottleneck

GoldenFuzz [39] starts from the observation that early-stage learning does not always require microarchitectural feedback. Before a generator can effectively search for subtle discrepancies, it must first learn to produce instruction sequences that are syntactically legal, semantically coherent, and extendable. Paying RTL cost for each candidate during this phase is inefficient, since many inputs are filtered for basic executability. GoldenFuzz addresses this by separating refinement into stages with different oracles, using a fast architectural reference where it is sufficient and reserving RTL for where it is necessary.

As shown in Fig. 3, GoldenFuzz begins with pretraining and then enters a GRM-driven stage that rapidly generates and evaluates instruction blocks for ISA-level executability. This high-throughput stage supports aggressive screening and refinement without RTL overhead, producing stronger building blocks before transitioning to DUT fuzzing. In the DUT stage, the objective shifts toward coverage growth and the discovery of microarchitecture-dependent discrepancies, with bug candidates identified via differential testing between DUT

and GRM traces. GoldenFuzz further improves stability and corpus health by replacing scalar-reward reinforcement learning with preference-style updates. It scores candidates to capture both their individual contribution, such as new coverage, and their redundancy relative to the current population, encouraging diversity as campaigns mature. The generator is then updated by comparing *better* and *worse* examples, aligning generation with robust improvements while directly learning patterns to avoid, consistent with direct preference optimization [21], [39].

Overall, GoldenFuzz provides a concrete affirmative direction for RQ3, since staged evaluation can substantially reduce RTL dependence while preserving a DUT-grounded path to vulnerability discovery [39]. In reported evaluations, GoldenFuzz achieves the highest hardware coverage compared to existing approaches while producing significantly shorter test cases (less than 30 instructions). It detects all previously known vulnerabilities in benchmark cores, and uncovers five new vulnerabilities on open-source cores (four with CVSS 3.0 scores above 7) [18], along with two previously unknown vulnerabilities during internal verification on a commercial core [1]. While the specific findings were not disclosed, this result indicates that the approach can surface issues even in proprietary, commercially deployed designs, suggesting practical industrial relevance. Despite its scalability, important challenges remain since a GRM cannot capture timing and microarchitecture-specific effects during the fast stage, the approach still depends on informative feedback design and robust triage in the DUT stage, and generalizing beyond accessible training artifacts and architectures remains nontrivial. These open issues motivate the broader challenges summarized in the next section.

V. Open Challenges and Future Work

AI-assisted fuzzing has significantly improved the generation, scheduling, and optimization of processor test programs. However, existing approaches largely reuse conventional hardware coverage and vulnerability detection oracles, without substantially advancing them. Improvements in this area can have a profound impact on hardware security research. In this section, we highlight these open challenges and discuss potential directions for future work.

Coverage Metric. Coverage remains essential in hardware fuzzing as it guides test case generation toward deep, unexplored internal states of the DUT. However, widely adopted hardware-centric metrics, such as toggle, branch/condition, and FSM-state coverage, are insufficient. While simple and efficiently measurable, these metrics treat observations as *independent* events. In other words, each bit in the coverage map corresponds to a local predicate (e.g., a signal toggle) that was captured regardless of the execution context. This independence assumption overlooks the *path-dependent* nature of hardware execution, a fundamental property in which the semantic meaning of an event is defined by its temporal order and guards derived from previously latched states. A coverage metric suitable for modern hardware fuzzing must capture not just local events, but *structured progress* across protocol, control, and data-flow phases. Coverage should represent *dependent transitions* [2], [27] by capturing temporally ordered event sequences together with the state predicates that make those events meaningful.

Listing 1: Example hardware module with dependent coverage conditions.

```
module bus_ifc (...);
   ...
   always_ff @(posedge clk or posedge rst) begin
      if (rst) begin
         req_seen <= 1'b0;
         transfer_done <= 1'b0;
      end
      // Stage 1: request must be seen first
      if (req) begin
         req_seen <= 1'b1;
      end
      // Stage 2: acknowledge only matters after
         ↪ request
      if (ack && req_seen) begin
         transfer_done <= 1'b1;
      end
   end
   // Output valid only when both stages occurred
   assign ready = transfer_done;
   ...
endmodule
```

To make this limitation concrete, consider a design illustrated in Listing 1, in which the assertion of one signal is only meaningful if another signal has previously occurred. For instance, in a bus interface, the event represented by `ack` is only semantically relevant after a request signal (`req`) has been observed and recorded internally. An independent coverage map credits `ack` whenever it asserts, even if asserted prematurely, treating every activity as valid progress. A dependency-aware metric, in contrast, assigns coverage to the causal relationship embodied in the design (e.g., `req` occurs, then `ack` occurs while `req_seen` is true). By encoding such control and data-flow dependencies through guarded state-transition graphs or bounded-length temporal patterns, AI-assisted fuzzers can be provided with a feedback signal that faithfully reflects hardware semantics, thereby improving stimulus generation by integrating microarchitectural semantics with ISA semantics.

Thus, introducing a coverage metric for hardware that captures path-dependent behavior represents an important and timely research direction. Currently, AI-assisted hardware fuzzers [37]–[39] primarily leverage AI to generate high-quality test cases, but these models can be extended to provide richer, semantically informed hardware feedback. For example, graph neural networks (GNNs) or other structured reasoning models [40] can encode the relationships among microarchitectural states, instruction sequences, and signal dependencies, effectively learning the causal structure of hardware behavior. By integrating these models into the fuzzing loop, the fuzzer can prioritize inputs that explore previously unseen paths, detect subtle violations, and identify vulnerabilities that would be invisible to conventional coverage metrics.

Vulnerability Detection. Vulnerability detection is a critical component of any fuzzing framework. Without an effective vulnerability detection mechanism, a fuzzer may generate test cases that trigger a bug or security flaw in the target. However, if the framework cannot detect these violations through crashes, assertion failures, or trace analysis, no vulnerability is reported,

rendering the fuzzing effort ineffective.

This challenge is particularly critical for hardware fuzzers. Unlike software, hardware does not *crash* in the traditional sense. Even when presented with invalid inputs or bugs, hardware continues execution. In software fuzzing, operating systems abstract execution details and provide feedback to the fuzzer in the form of crashes, such as memory violations, signaling potential vulnerabilities [9], [15], [33]. In contrast, hardware fuzzers must continuously monitor execution to detect security rule violations, as there is no built-in notion of a crash. Hardware fuzzers, therefore, require explicit definitions for different types of faults and vulnerabilities to provide meaningful feedback during testing. As discussed in Section III, existing hardware fuzzers employ a variety of vulnerability detection methods [5], [10], [16], [26], [27], [30], [37]–[39]. Many rely on differential vulnerability detection using GRMs. GRMs represent the expected behavior of hardware at the instruction level, providing a standard against which actual execution can be compared. While this approach is practical for detecting certain classes of bugs and vulnerabilities, GRMs have fundamental limitations that can lead to missed vulnerabilities or excessive false positives. Two of the most important limitations are the following.

1) Microarchitectural behavior is not represented. GRMs generally capture only the architectural state of the processor (i.e., the state visible at the instruction set architecture level). However, they do not model microarchitectural components such as reorder buffers, branch predictors, or speculative execution units. This limitation can cause hardware fuzzers to miss vulnerabilities that manifest only in microarchitectural modules [29]. For example, consider Spike [22], a widely used GRM for RISC-V that forms the basis for several recent hardware fuzzers. Since Spike does not model microarchitectural features, fuzzers relying solely on it may fail to detect vulnerabilities in modules such as branch prediction, out-of-order execution, or speculative memory accesses.

2) GRMs represent a single implementation scenario for each ISA instruction. While GRMs capture the defined behavior of the instruction set, real hardware implementations may include flexibility or optional behaviors that are not modeled. When GRMs are used for vulnerability detection, this mismatch can generate false positives or fail to account for edge-case vulnerabilities. For example, in the RISC-V ISA [23], [24], certain instructions allow multiple legal implementations for features such as memory ordering or atomicity. A GRM models only one canonical implementation, potentially missing hardware-specific vulnerabilities in alternative implementations or reporting benign deviations as errors.

These limitations highlight a key gap in current hardware fuzzing. While GRMs provide a foundation for architectural validation, they are insufficient for comprehensive vulnerability detection. A promising direction for future work is to leverage AI models not only to generate high-quality test cases but also to detect vulnerabilities directly. Recent advances in AI reasoning make it feasible to train domain-specific models that internalize expert knowledge of hardware security, capturing how vulnerabilities emerge from complex interactions among microarchitectural components, execution sequences, and control-flow dependencies. Initiatives such as HackTheSilicon [11], the world's largest hardware security competition, demonstrate the value of systematically codifying expert reasoning. HackTheSilicon is conducted in collaboration with leading industry and academic partners, including Intel, OpenTitan, lowRISC, and Synopsys, and exposes participants to real-world, industry-quality vulnerabilities under competitive conditions. By leveraging the insights, methodologies, and annotated vulnerabilities developed through such competitions, future AI-assisted fuzzers could go beyond conventional coverage feedback, using execution traces, assertion violations, and causal signal relationships to identify vulnerabilities in a scalable, continuously improving manner. This approach promises to close the detection gap left by GRM-based methods and substantially enhance the real-world impact of hardware fuzzing and hardware security.

VI. Conclusion

AI-assisted hardware fuzzing is emerging as a practical response to the rising complexity of modern processors. As designs grow deeper and more heterogeneous, mutation-driven fuzzers increasingly waste effort on invalid or low-value tests and often fail to reach the long-horizon behaviors where many security issues reside. Learning-guided generation and refinement shift fuzzing toward semantics-aware exploration, producing more coherent instruction sequences, adapting to execution feedback, and using expensive simulation more selectively. Future progress will require richer, dependency-aware coverage metrics and AI-driven reasoning engines that embed expert hardware security knowledge, enabling fuzzers to interpret execution traces, connect low-level signals, and identify vulnerabilities more effectively. Insights from initiatives like HackTheSilicon demonstrate how expert reasoning can be distilled and applied at scale. By integrating these approaches, hardware fuzzers can move beyond generating high-quality test cases toward intelligent, auditable, and semantically guided vulnerability discovery, closing the gap left by conventional coverage and detection methods.

Acknowledgement

Our research work was partially funded by Intel's Scalable Assurance Program, DFG-SFB 1119-236615297, the European Union under Horizon Europe Programme-Grant Agreement 101070537-CrossCon, NSF-DFG-Grant 538883423, the European Research Council under the ERC Programme-Grant 101055025-HYDRANOS, and Synopsys with EDA tools licenses support. This work does not in any way constitute an Intel endorsement of a product or supplier. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of Intel, the European Union, or the European Research Council.

REFERENCES

[1] Beyond Semiconductor. https://www.beyondsemi.com/.

[2] 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, 2024.

[3] 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. IEEE, 2023.

[4] 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, 2023.

[5] 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. *The Network and Distributed System Security (NDSS) Symposium 2026*, 2026.

[6] Congress.gov. H.R.4346 - CHIPS and Science Act 117th Congress (2021-2022). https://www.congress.gov/bill/117th-congress/house-bill/4346, 2022.

[7] Bernhard Dachs. The European Chips Act. Technical report, FIW-Kurzbericht, 2023.

[8] Krste Asanović et al. The Rocket Chip Generator. (UCB/EECS-2016-17), 2016.

[9] Google. American Fuzzy Lop. https://github.com/google/AFL, 2019.

[10] Raphael Götz, Christoph Sendner, Nico Ruck, Mohamadreza Rostami, Alexandra Dmitrienko, and Ahmad-Reza Sadeghi. RLFuzz: Accelerating Hardware Fuzzing with Deep Reinforcement Learning. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2025.

[11] HackTheSilicon. Hack The Silicon: The World's Largest Hardware Security Capture the Flag Competitions. https://hackthesilicon.com. [Online; accessed 22-Aug-2025].

[12] Muhammad Usman Hadi, Qasem Al-Tashi, Rizwan Qureshi, Abbas Shah, Amgad Muneer, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, and Seyedali Mirjalili. A Survey on Large Language Models: Applications, Challenges, Limitations, and Practical Usage. *Authorea Preprints*, 2023.

[13] 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. IEEE, 2023.

[14] 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. IEEE, 2021.

[15] The LLVM Compiler Infrastructure. libFuzzer: A Library for Coverage-guided Fuzz Testing. *https://llvm.org/docs/LibFuzzer.html (visited on 01/12/2025)*, 2021.

[16] 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, 2022.

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

[18] NIST. Common Vulnerability Scoring System. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator, 2019.

[19] National Institute of Standards and Technology (NIST). Chips For America. https://www.nist.gov/chips, 2025.

[20] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training Language Models to Follow Instructions with Human Feedback. *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.

[21] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023.

[22] RISC-V. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim.

[23] RISC-V. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-8b9dc50-2024-08-30, 2024.

[24] RISC-V. The RISC-V Instruction Set Manual Volume II: Privileged ISA. https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-8b9dc50-2024-08-30, 2024.

[25] Mohamadreza Rostami, Chen Chen, Rahul Kande, Huimin Li, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. Fuzzerfly Effect: Hardware Fuzzing for Memory Safety. *IEEE Security and Privacy*, 22(4):76–86, 2024.

[26] 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. IEEE, 2024.

[27] 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 DAC*, pages 1–6, 2024.

[28] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. The Fuzz Odyssey: A Survey on Hardware Fuzzing Frameworks for Hardware Design Verification. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, pages 192–197, 2024.

[29] Nikhilesh Singh, Vinod Ganesan, and Chester Rebeiro. *Secure Processor Architectures*, page 1–29. Springer Nature Singapore, 2022.

[30] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: CPU Fuzzing via Intricate Program Generation. In *Proc. 33rd USENIX Secur. Symp*, pages 1–18, 2024.

[31] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM Neural Networks for Language Modeling. In *Interspeech*, 2012.

[32] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[33] Robert Swiecki. Honggfuzz: A General-purpose, Easy-to-use Fuzzer with Interesting Analysis Options. *https://github.com/google/honggfuzz (visited on 01/12/2025)*, 2017.

[34] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing Hardware like Software. In *31st USENIX Security Symposium*, pages 3237–3254, 2022.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *In 31st NeurIPS*, page 6000–6010, 2017.

[36] Verilator. Welcome to Verilator. https://www.veripool.org/verilator/, 2019.

[37] 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, 2025.

[38] Lichao Wu, Mohamadreza Rostami, Huimin Li, and Ahmad-Reza Sadeghi. HFL: Hardware Fuzzing Loop with Reinforcement Learning. In *Design, Automation & Test in Europe Conference*, pages 1–7, 2025.

[39] Lichao Wu, Mohamadreza Rostami, Huimin Li, Nikhilesh Singh, and Ahmad-Reza Sadeghi. GoldenFuzz: Generative Golden Reference Hardware Fuzzing. In *The Network and Distributed System Security (NDSS) Symposium 2026 (NDSS Symposium 26)*, 2026.

[40] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, January 2021.

[41] 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, 2023.

[42] Florian Zaruba and Luca Benini. The Cost of Application-class Processing: Energy and Performance Analysis of a Linux-ready 1.7-GHz 64-bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[43] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *4th Workshop on Computer Architecture Research with RISC-V*, 2020.