State of Hardware Fuzzing: Current Methods and the Potential of Machine Learning and Large Language Models

Kevin Immanuel Gubbi*, Mohammadnavid Tarighat*, Arvind Sudarshan*, Inderpreet Kaur*
Pavan Dheeraj Kota*, Avesta Sasan*, Houman Homayoun*
*Department of Electrical and Computer Engineering, University of California Davis, Davis, CA 95616
*{kgubbi, mtarighat, asudarshan, inpkaur, pkota, asasan, hhomayoun}@ucdavis.edu;

***Index Terms*—Hardware Fuzzing, Machine Learning, Large Language Models, Hardware Security Verification.**

***Abstract*—Hardware fuzzing has emerged as a powerful technique for detecting security vulnerabilities and functional bugs in modern hardware systems. Unlike traditional verification approaches that rely on predefined testbenches and formal proofs, hardware fuzzing generates and mutates inputs dynamically to uncover unexpected behaviors. Despite its effectiveness, hardware fuzzing faces challenges such as test case explosion, coverage limitations, and debugging complexity. Recent advancements in Machine Learning (ML) and Large Language Models (LLMs) offer new opportunities to enhance hardware fuzzing by improving test case generation, optimizing coverage feedback, and automating debugging processes. This paper provides a comprehensive survey of the current state of hardware fuzzing, highlighting its methodologies, applications, and limitations. Furthermore, we explore the potential of ML and LLMs in augmenting fuzzing workflows and discuss key challenges that must be addressed for broader adoption. By synthesizing insights from existing research and industry practices, we outline future research directions that can bridge the gap between automated hardware fuzzing and intelligent, adaptive testing frameworks.**

I. Introduction

Modern hardware designs are becoming increasingly complex, making it challenging to ensure their correctness, reliability, and security. As integrated circuits (ICs) grow in scale and sophistication, traditional verification techniques face significant limitations in scalability, automation, and vulnerability detection. Hardware verification has traditionally relied on techniques such as simulation-based verification (e.g., Universal Verification Methodology, or UVM), formal verification, and emulation. While these methods have been effective in catching design flaws, they require extensive manual effort, domain expertise, and, in many cases, suffer from test coverage gaps. To address these challenges, hardware fuzzing has emerged as a promising alternative. Inspired by software fuzzing, this technique applies dynamic, automated test generation strategies to hardware designs, helping uncover security vulnerabilities and functional inconsistencies that conventional methods may miss. Hardware fuzzing does not rely on predefined testbenches or constrained verification models; instead, it generates and mutates inputs to explore different execution states of the design-under-test (DUT). This approach is particularly useful in identifying corner-case bugs, security exploits, and robustness issues in modern processors, accelerators, and system-on-chip (SoC) architectures. Despite its advantages, hardware fuzzing is still in its early stages and has not yet seen widespread adoption in industry. Some of the primary challenges include computational overhead, difficulty in debugging failures, and a lack of standardization in hardware fuzzing frameworks. Furthermore, unlike formal verification, which can provide correctness guarantees, fuzzing does not inherently ensure exhaustive verification coverage. Recent advancements in Machine Learning (ML) and Large Language Models (LLMs) present exciting opportunities to enhance hardware fuzzing. ML-based techniques can optimize test case generation, improve feedback-driven coverage exploration, and assist in vulnerability detection. Meanwhile, LLMs have the potential to automate testbench creation, script generation, and even debugging analysis, significantly reducing human effort in verification processes. Integrating AI-driven methodologies into hardware fuzzing could make it a more efficient and intelligent approach to hardware security and functional validation. This paper provides a comprehensive survey of the current state of hardware fuzzing, highlighting its methodologies, advantages, and limitations. Additionally, we explore how ML and LLMs can be leveraged to enhance hardware fuzzing and discuss the key challenges that must be addressed for broader adoption in both academic and industrial settings. Finally, we outline open research directions that could drive the development of AI-enhanced hardware fuzzing frameworks, making them a viable addition to modern hardware verification workflows.

II. Current State of Hardware Fuzzing

A. The Need for Hardware Fuzzing in Security Validation

Modern hardware systems, particularly system-on-chips (SoCs) and processor architectures, have become increasingly complex, integrating multiple intellectual property (IP) blocks from diverse vendors. This growing complexity introduces security vulnerabilities that are difficult to detect using traditional verification techniques, such as constrained random verification (CRV) and formal verification (FV). Unlike software, where patches can be applied after deployment, hardware

vulnerabilities cannot be easily fixed post-fabrication, making pre-silicon security validation critical. Formal verification techniques, such as theorem proving and model checking, offer exhaustive validation but face scalability challenges due to state explosion and the requirement of manual assertion writing. Similarly, CRV generates randomized test cases but lacks the guided intelligence needed to reach deep hardware states where security vulnerabilities might reside. To address these limitations, hardware fuzzing has emerged as an effective alternative, leveraging automated input generation and mutation strategies to systematically explore hardware state spaces and identify potential vulnerabilities.

B. Evolution of Hardware Fuzzing Techniques

1) Fuzzing Hardware Like Software: One of the earliest approaches to hardware fuzzing was introduced by Trippel et al. [1], where the idea of translating register-transfer level (RTL) designs into software models enabled the use of software fuzzing methodologies like American Fuzzy Lop (AFL). This method involved translating RTL into software-executable C++ models using Verilator, applying dynamic mutation of test inputs within a software-based fuzzing framework, and tracking HDL line coverage and finite state machine (FSM) transitions to refine test cases. This technique was two orders of magnitude faster than constrained random verification, achieving 83 percent HDL line coverage in OpenTitan cores. However, its reliance on golden reference models for correctness checking made it impractical for large-scale hardware designs. Furthermore, the equivalence between software execution traces and actual hardware behavior was sometimes questionable.

2) Coverage-Guided Hardware Fuzzing: RFUZZ [2], introduced at ICCAD 2018, was one of the earliest attempts at coverage-directed test generation (CDG) for hardware fuzzing. Unlike software-based fuzzing, RFUZZ utilized FPGA-accelerated simulation to improve execution speed. Its core innovations included mux toggle coverage, which tracked multiplexer control signal toggles to measure circuit exploration, meta-reset transformation to reset registers efficiently, and sparse memory tracking to reduce overhead in large designs. RFUZZ demonstrated significant improvements over random testing, particularly for feedback-driven circuits. However, it struggled with scalability as fitting large designs on an FPGA was a challenge. Additionally, the fuzzing speed was bottlenecked by the software analysis overhead of test input mutation and coverage tracking.

3) Differential Fuzzing and Coverage-Guided CPU Verification: DIFUZZRTL [3], introduced at IEEE Symposium on Security and Privacy in 2021, improved upon RFUZZ by introducing register-coverage guided fuzzing, which focused on control register transitions rather than multiplexer toggles. This technique was particularly effective in processor architectures, enabling more precise finite state machine state exploration. DIFUZZRTL provided a 40 times increase in execution speed compared to RFUZZ and was 6.4 times more efficient in exploring state spaces, leading to faster vulnerability detection. This approach employed a differential testing methodology, comparing RTL execution traces with a golden model to identify inconsistencies. DIFUZZRTL successfully fuzzed complex out-of-order processors such as the RISC-V BOOM core, which RFUZZ failed to scale to. However, DIFUZZRTL required an instruction set architecture (ISA) simulator, limiting its ability to detect non-ISA vulnerabilities.

4) Instruction-Level and Security-Driven Hardware Fuzzing: TheHuzz [4], introduced at the USENIX Security Symposium in 2022, focused on processor instruction fuzzing using golden-reference models to detect deviations from expected CPU behavior. TheHuzz incorporated comprehensive coverage metrics, including statement, branch, toggle, expression, condition, and FSM coverage. TheHuzz outperformed Cadence JasperGold by mitigating state explosion and reducing the manual effort required for formal verification. However, its runtime overhead was significantly higher than that of DIFUZZRTL, reaching 71 percent compared to DIFUZZRTL's 6.9 percent, making it less practical for large-scale designs requiring rapid iteration.

5) Security-Oriented System-on-Chip Fuzzing: SoCFuzzer [5], introduced at DATE 2023, shifted away from general coverage metrics and introduced cost function-driven fuzzing for security-specific validation. Instead of maximizing code coverage, SoCFuzzer employed cost functions to guide fuzzing toward security-critical areas. The framework incorporated FPGA-based real-time security monitoring and gray-box fuzzing techniques, allowing verification without requiring a golden model. SoCFuzzer successfully detected security vulnerabilities in RISC-V-based SoCs, including AES key leakage and privilege escalation attacks. However, the cost function required for guiding fuzzing had to be carefully designed, necessitating significant expert knowledge, which limited automation.

6) Machine Learning-Based and Hybrid Hardware Fuzzing: ChatFuzz [6], introduced in 2024, integrated machine learning-based input generation, leveraging large language models and reinforcement learning to generate effective test cases. It achieved 75 percent condition coverage on RocketCore in 52 minutes, significantly outperforming previous approaches. The methodology behind ChatFuzz employed a three-step training process, beginning with unsupervised learning to model machine instruction sequences, followed by reinforcement learning with an ISA disassembler to refine instruction validity, and concluding with coverage-driven reinforcement learning using RTL simulation feedback. ChatFuzz also utilized a differential testing approach, comparing ISA model outputs with RTL simulations to detect mismatches. Although ChatFuzz significantly improved fuzzing efficiency, it was heavily dependent on large-scale machine learning training datasets and required manual bug validation, necessitating further advancements in AI-based filtering techniques to improve automation.

7) Hybrid and Formal Verification-Assisted Fuzzing: HyPFuzz [7] introduced a hybrid approach by integrating formal

TABLE I: Comparison of Hardware Fuzzing Techniques

Fuzzer	Methodology	Key Contributions	Advantages	Limitations
Fuzzing Hardware Like Software (Trippel et al. [1])	RTL-to-software translation using Verilator; Uses AFL for fuzzing; Software fuzzing harness executes RTL as C++ models; Tracks HDL line coverage and FSM state transitions.	First to adapt software fuzzing to hardware; Faster execution using compiler-inserted coverage tracking; Achieves 83% HDL line coverage on OpenTitan.	Significant speedup over CRV; No need for direct RTL instrumentation.	Questionable equivalence between software models and real hardware; Requires golden reference models.
RFUZZ [2]	FPGA-accelerated simulation for Coverage Directed Test Generation (CDG); Uses Mux Toggle Coverage to track multiplexer signal toggling; Introduces MetaReset transformation and Sparse Memories.	First FPGA-accelerated RTL fuzzer; Improves over random testing; Defines a new hardware coverage metric.	Faster execution via FPGA acceleration; Reduces redundancy through deterministic resets.	Limited scalability due to FPGA constraints; Bottlenecked by software analysis.
DIFUZZRTL [3]	Register-Coverage Guided Fuzzing for CPU designs; Measures control register values per cycle; Differential Testing comparing RTL execution with a golden ISA model.	40× faster than RFUZZ in CPU simulations; 6.4× more efficient in state space exploration; Successfully fuzzed out-of-order BOOM cores.	Highly effective for CPU verification; No FPGA dependency.	Requires an ISA simulator, limiting non-ISA vulnerability detection.
TheHuzz / [4]	Instruction-level fuzzing targeting CPU RTL; Uses golden-reference models for correctness checking; Tracks statement, branch, FSM, toggle, and expression coverage.	Detects software-exploitable CPU vulnerabilities; Outperforms JasperGold formal verification.	Covers a broad range of hardware behaviors.	High runtime overhead (71% vs. DIFUZZRTL's 6.9%); Requires golden reference models.
SoCFuzzer [5]	Cost function-driven fuzzing for security verification of SoCs; FPGA-based real-time monitoring for vulnerabilities; Gray-box fuzzing without full SoC access.	Introduces security-aware cost functions; Detects AES key leakage and privilege escalation vulnerabilities.	Effective for security-driven fuzzing without golden models.	Defining cost functions requires expert knowledge; Limited automation.
ChatFuzz [6]	Machine Learning (ML)-based instruction sequence generation; Large Language Models (LLMs) trained on machine code; Reinforcement Learning (RL) with RTL simulation as a reward.	Achieves 75% condition coverage on RocketCore in 52 minutes; Uses AI-based test generation.	Automates test case generation; Outperforms prior fuzzers in speed.	Requires extensive ML training datasets; Needs human validation for bug detection.
HyPFuzz [7]	Hybrid approach combining formal verification with fuzzing; Dynamic scheduling for balancing methods; Heuristic coverage point selection.	11.68× faster than TheHuzz; 3.06× faster vulnerability detection.	Expands design space coverage significantly.	Complex integration of formal methods; Marginal total design space coverage improvement.

TABLE II: Evaluation Metrics for Hardware Fuzzing Frameworks

Metric	Description
Total Bugs Found	Measures the number of unique hardware vulnerabilities identified during fuzzing. A higher number of detected bugs indicates a more effective framework.
Time-to-First Bug (TTFB)	Quantifies the time required for the fuzzer to detect the first security vulnerability in the design, reflecting its initial efficiency.
Cumulative Bug Detection	Tracks the rate at which new vulnerabilities are discovered over time, providing insight into long-term effectiveness.
Toggle Coverage	Ensures that all logic gates and signals within the design toggle between 0 and 1 during testing, validating full circuit activation.
Functional Coverage	Assesses whether the fuzzer exercises all FSM transitions, module interactions, and expected behaviors to confirm operational correctness.
Code Coverage	Measures how much of the RTL code is executed, including statement, branch, and path coverage. Ensures deeper analysis of the design.
Line Coverage	Verifies whether each line of RTL code has been executed at least once, ensuring full design exploration.
Verdi Coverage	Determines the activation of predefined cover groups and assertions, ensuring verification of security constraints and edge behaviors.
Execution Speed	Evaluates how quickly the fuzzer generates, executes, and analyzes test inputs, determining overall fuzzing efficiency.
Resource Utilization	Tracks CPU, memory, and FPGA usage during fuzzing, ensuring that computational overhead remains manageable.
Scalability	Assesses the framework's ability to handle large and complex SoC designs while maintaining efficiency.
Verification Plan	Ensures rigorous validation through design constraints, corner case testing, and integration with formal verification techniques.

TABLE III: ML techniques applied in hardware security.

ML Method	Use Case	Strengths	Challenges
Supervised Learning	Bug Classification	High Accuracy	Requires Labeled Training Data
Reinforcement Learning	Adaptive Test Input Generation	Maximizes Coverage	High Training Cost
Graph Neural Networks (GNNs)	RTL Structure Analysis	Captures Hardware Dependencies	Computationally Expensive

verification with fuzzing to expand design space coverage. It employed a dynamic scheduling strategy to alternate between formal verification techniques and mutation-based fuzzing, achieving an eleven-fold improvement in fuzzing speed over TheHuzz and reducing vulnerability detection time by a factor of 3.06. Although HyPFuzz successfully uncovered previously undetected security vulnerabilities, its overall effectiveness was constrained by the complexity of integrating formal verification with fuzzing techniques. Additionally, the expansion in design space coverage achieved by this method was limited to approximately 1 percent, highlighting the challenges of balancing formal verification with practical fuzzing methodologies.

Hardware fuzzing has evolved from software-based approaches to coverage-guided, differential, security-driven, and machine learning-enhanced methodologies. Each method has contributed to improving hardware security validation by addressing different aspects of the verification process. While machine learning and hybrid methods have shown promising results, challenges remain in scalability, automation, and generalizability across diverse hardware architectures. Continued advancements in AI-driven fuzzing, security-aware coverage metrics, and efficient system-on-chip-scale fuzzing will shape the next generation of hardware security validation tools.

III. Metrics for Evaluating Hardware Fuzzing Frameworks

To effectively assess the performance and efficiency of hardware fuzzing frameworks, we define a set of quantitative metrics that measure bug identification, coverage, execution efficiency, and verification accuracy. These metrics provide an objective basis for comparing different fuzzing methodologies and ensuring comprehensive hardware security validation. Bug identification plays a critical role in evaluating a fuzzing framework. The total number of unique bugs detected is a primary measure of effectiveness. Additionally, time-to-first-bug (TTFB) is an important metric, quantifying how quickly a fuzzer detects its first vulnerability in the design. The cumulative bug detection rate provides further insight into how efficiently new vulnerabilities are discovered over time. A fuzzing framework that maintains a high and consistent bug detection rate is preferable, as it ensures continuous discovery of security flaws.

Coverage metrics assess how thoroughly the fuzzer explores different aspects of the hardware design. Toggle coverage measures whether all logic gates and signals transition between 0 and 1, ensuring that every element of the circuit is activated during testing. Functional coverage evaluates whether all expected finite-state machine (FSM) transitions and module interactions are exercised, confirming that the fuzzer effectively tests the intended operational states of the hardware. Code coverage provides a more detailed breakdown of RTL execution, incorporating statement coverage, which measures the percentage of executed statements, branch coverage, which determines whether both true and false conditions of branching statements have been tested, and path coverage, which ensures different execution paths within the design are explored.

Additionally, line coverage verifies whether each individual line of RTL code has been executed at least once. Beyond these, Verdi coverage provides a specialized measure of the fuzzer's effectiveness in activating cover groups and assertions embedded in the design. This ensures that the framework exercises predefined security and correctness properties, allowing for validation of edge behaviors and potential corner cases. Performance metrics determine the computational efficiency and scalability of the fuzzing framework. Execution speed measures how quickly the fuzzer generates, executes, and evaluates test inputs, which is crucial for large-scale designs. Resource utilization tracks the consumption of CPU, memory, and FPGA resources, ensuring that the framework does not impose excessive computational overhead. Scalability measures how well the fuzzer adapts to increasingly complex system-on-chip (SoC) designs, determining whether it can maintain efficiency across large and deeply integrated architectures. Verification planning is essential for ensuring that the hardware fuzzing process is rigorous and systematic. A well-defined verification plan should include design constraint validation, which ensures that RTL constraints are met under all test conditions. Comprehensive corner case testing evaluates whether the fuzzer effectively triggers rare execution scenarios that could lead to functional failures. Formal verification integration complements fuzzing by providing exhaustive correctness checks, enabling a higher degree of design security assurance.

IV. LLMs and Fine-Tuning

Pre-trained models like Llama3, Mistral or GPT-4 are trained on large corpora of text for them to learn general linguistics and knowledge. However they lack the knowledge for specialized tasks like Hardware Fuzzing, where domain-specific knowledge, such as knowledge of hardware description languages and known circuit vulnerabilities are essential. Fine tuning is a method for domain specific adaptations. Fine tuning a pretrained model from scratch is a resource intensive task. For example GPT-4 is a model with 175 Billion parameters and fine tuning a model this big is an intensive task. The time required for training such a big model is one problem but the amount of space required is also a problem.

A. LoRA [8]

LoRA is a method that achieves the work of fine-tuning with minimal computational resources while proving just as effective as full fine tuning for most cases.

The main idea behind Low Rank Adaptation(LoRA) is that we fine tune a model by only adapting a small number of parameters while keeping the rest of them frozen/unchanged. According to Hu et. Al. (LoRA paper reference) the idea behind LoRA is based on a simple observation that model weight matrices in LLMs can be decomposed into lower rank matrices.

The following equation represents the main idea behind LoRA:

$$W = W_0 + \Delta W = W_0 + BA$$

$W_0$is the original weight matrix of the model,
ΔW is the learned adaptations for the specific task,
W is the final weight matrix,
B is a matrix of dimension $d \times r$, and
A is a matrix of dimension $r \times k$

For instance consider that you have a weight matrix with dimensions 800×500, this gives you a total of $400,000$ parameters to train which would take approximately. Let's say that it has an intrinsic rank of 6, that means we can decompose it into two matrices: $A - 800 \times 6$ and $B - 500 \times 6$. So now instead of needing to fine-tune $400,000$ parameters you only need to fine-tune $7,600$ parameters.

Advantages:

- LoRA significantly reduces the number of parameters to be adapted during the process of fine tuning. Instead of adapting all of the model's parameters we only need to adapt a few making the process memory efficient. This means we can fine tune large language models on consumer grade hardware with limited memory
- The base model's original parameters remain frozen, preserving the original knowledge of the model
- Since LoRA only creates adapters for the parameters keeping the original parameters of the model unchanged we can swap LoRA adapters for different tasks without needing to retrain the model

Disadvantages:

- While LoRA may perform well in most cases, it may not capture very complex adaptations as effectively as full fine-tuning for tasks requiring significant adaptations over the base model. For example, consider a model developed to detect side channel attacks in microprocessor architectures. LoRA's linear adaptation might struggle to capture the signatures of side-channel attacks, potentially missing critical, subtle security indicators in microprocessor designs.
- LoRA best works for models with transformer like architecture, however it may not be the best approach for all architectures. For example in physics or climate models which are characterized by nonlinear equations differential equations to using LoRA might introduce numerical instability leading to a discrepancy in the predicted and actual output of the model.
- There is a risk of the model overfitting especially if the dataset is limited. Consider a model adapted to identify side channel attacks for microcontrollers but the dataset used to train the LoRA adapter only had the side-channel traces of a specific model of microcontroller, it might learn the noise patterns unique to that specific device failing to identify legitimate attacks as it fails to generalize the knowledge from the dataset.

V. The Role of Machine Learning in Hardware Fuzzing

Machine Learning (ML) has significantly impacted various domains, and hardware fuzzing is no exception. Traditional

TABLE IV: Potential applications of LLMs in hardware fuzzing.

Application	Benefit	Limitation
Testbench Generation	Automates Test Creation	Needs Extensive Training Data
Debugging Automation	Faster Root Cause Analysis	Interpretability of AI Decisions
Constraint Optimization	Enhances Coverage-Driven Fuzzing	Requires Fine-Tuning for Hardware Constraints

fuzzing techniques rely on heuristics and randomization to generate test cases. However, ML-based fuzzing introduces a data-driven approach that optimizes test input generation, coverage feedback, and vulnerability detection, resulting in a more efficient and intelligent fuzzing process. Several hardware fuzzers, such as ChatFuzz [6] and HyPFuzz [7], have already started incorporating ML-based methodologies to improve input generation, coverage maximization, and bug discovery.

A. ML Techniques for Hardware Fuzzing

Several ML methodologies have been explored in hardware fuzzing, each offering unique advantages in optimizing test input selection, improving execution efficiency, and enhancing the discovery of vulnerabilities. Supervised learning is frequently used to classify detected hardware vulnerabilities and anomalies. By training models on previously identified hardware bugs, supervised classifiers can predict and categorize new vulnerabilities, reducing the manual effort required for bug triaging. ChatFuzz, for example, uses an ML-based classification mechanism to distinguish between functional mismatches and security-critical deviations in CPU execution.

Reinforcement learning (RL) has shown promise in adaptive fuzzing strategies, particularly in generating optimal test inputs. RL-based fuzzers iteratively refine their inputs based on feedback, leading to an exploration-exploitation balance. ChatFuzz implements coverage-driven reinforcement learning, where an ISA disassembler and RTL simulator act as the reward agent, ensuring maximal state-space exploration within processor architectures.

Graph Neural Networks (GNNs) provide a structured way to model RTL circuit connectivity and dependencies. Unlike conventional ML models, GNNs represent netlists as a graph, allowing for more context-aware fuzzing decisions. GNNs have been explored in RTL structure analysis, identifying hard-to-reach states in finite state machines (FSMs), which are critical for security validation.

VI. The Potential of Large Language Models in Hardware Fuzzing

Large Language Models (LLMs), such as GPT-based models, have demonstrated remarkable capabilities in automating code analysis, test generation, and debugging. In hardware fuzzing, LLMs can augment traditional methods by automating testbench creation, improving debugging assistance, and optimizing fuzzing constraints.

A. LLM Applications in Hardware Fuzzing

Testbench generation is a time-consuming process in hardware verification. LLMs have the potential to automate test script creation by analyzing RTL descriptions and synthesizing corresponding testbenches. This application has already been explored in ChatFuzz, where LLM-generated instruction sequences enhance test diversity, leading to higher coverage in processor architectures. Debugging assistance is another area where LLMs can provide value. Hardware fuzzing often generates a large volume of failing test cases, making manual root cause analysis impractical. LLMs can interpret error messages, extract relevant debugging traces, and suggest potential failure sources, accelerating the debugging workflow. Constraint optimization plays a crucial role in symbolic and concolic execution, where test inputs must satisfy specific constraints to reach targeted design states. LLMs can learn from prior fuzzing campaigns and refine constraint models, reducing false positives and enhancing constraint satisfaction-based fuzzing efficiency.

VII. Challenges and Open Research Directions

Despite its advantages, AI-driven hardware fuzzing faces multiple challenges. One significant limitation is the high computational cost of training complex ML and LLM models. Reinforcement learning-based fuzzers, for example, require thousands of RTL simulations to learn effective fuzzing strategies, making them prohibitively expensive for large-scale SoC designs. Another major challenge is lack of explainability. Unlike traditional verification techniques, AI-based fuzzers do not always provide a clear rationale for why certain test inputs trigger vulnerabilities. Debugging failures and interpreting ML-generated test cases remain open research problems. Limited training data is another bottleneck. Hardware fuzzing lacks large, labeled datasets that could be used to train supervised learning models. Unlike software vulnerability detection, where public datasets of security exploits exist, hardware fuzzing datasets remain proprietary and fragmented.

A. Future Research Directions

Future research should focus on hybrid verification approaches, where AI-based fuzzing is combined with formal verification techniques to enhance vulnerability detection accuracy. Hybrid approaches, such as HyPFuzz, have already demonstrated improvements in state-space coverage by integrating formal verification heuristics with fuzzing strategies. Developing efficient AI models tailored for hardware fuzzing is another promising direction. Current LLMs and RL-based models are resource-intensive, requiring significant hardware infrastructure. Exploring lightweight ML models that optimize test input selection without excessive computation overhead is necessary for practical adoption. The use of LLM-powered debugging assistants is an emerging field, where AI models

provide real-time feedback during fuzzing campaigns. By integrating LLM-based reasoning with verification logs, future tools could automatically suggest counterexamples, propose patches, and refine fuzzing strategies.

VIII. Conclusion

This paper surveyed the evolution of hardware fuzzing techniques, highlighting how machine learning and large language models are transforming the landscape of security validation in hardware design. AI-driven fuzzing techniques, including reinforcement learning-based adaptive test generation, graph-based RTL modeling, and supervised bug classification, offer significant advantages over traditional methods by improving coverage, automating test generation, and enhancing bug detection efficiency. However, challenges remain, including the high computational overhead of ML training, the difficulty of explaining AI-generated test cases, and the lack of structured hardware fuzzing datasets. Future research should explore hybrid verification strategies, efficient AI models, and LLM-assisted debugging tools to bridge the gap between automation and interoperability. As hardware security threats continue to evolve, AI-based fuzzing frameworks will play an essential role in identifying vulnerabilities early in the design cycle, ensuring robust and secure hardware architectures for future computing systems.

References

[1] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3237–3254, 2022.

[2] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[3] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1286–1303, IEEE, 2021.

[4] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3219–3236, 2022.

[5] M. M. Hossain, A. Vafaei, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023.

[6] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond random inputs: A novel ml-based hardware fuzzing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2024.

[7] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "{HyPFuzz}:{Formal-Assisted} processor fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1361–1378, 2023.

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.