The Fuzz Odyssey: A Survey on Hardware Fuzzing Frameworks for Hardware Design Verification

Raghul Saravanan
George Mason University
FairFax, Virginia, USA
rsaravan@gmu.edu

Sai Manoj Pudukotai Dinakarrao
George Mason University
FairFax, Virginia, USA
spudukot@gmu.edu

ABSTRACT

Hardware Security is at stake driven by the growing complexity and integration of processors, SoCs, and diverse third-party intellectual property (IP) hardware, all geared toward delivering advanced solutions. To preserve the system integrity and mitigate the post-production re-engineering costs, the Design Verification (DV) community employs dynamic and formal verification strategies. However, with the ever-increasing complexity of modern processors, these techniques fall in short of scalability and increased verification time. Recently, hardware fuzzing inspired by software testing has been navigating uncharted territories in hardware bug detection capabilities. Multiple hardware fuzzing techniques have been recently introduced that either utilize the hardware design in its inherent form for fuzzing or convert the hardware into software models and perform fuzzing to detect bugs. However, the existing techniques claim to be a silver bullet in their way, we provide some critical insights on these techniques by reviewing the fundamental principles of hardware fuzzing frameworks, the methodologies involved, and the diverse hardware designs in which they can be employed. Furthermore, we discuss the challenges and limitations of the fuzzing framework. We also present feasible future research directions based on our observations and insights.

CCS CONCEPTS

• **Hardware → Functional verification**; *Logic circuits*; **Application specific processors**.

ACM Reference Format:
Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. 2024. The Fuzz Odyssey: A Survey on Hardware Fuzzing Frameworks for Hardware Design Verification. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649476.3658697

1 INTRODUCTION

Addressing the complexities of the global semiconductor supply chain requires collaboration between various IC designers and vendors [7]. This collaboration spans the entire IC design cycle, from design to integration, to meet industry demands[2]. For example, in the design and evaluation of the Apple® A17 SoC chip, more than 11 third-party entities have been involved in delivering sophisticated solutions for the device [23]. As IC designs become more intricate, incorporating various IPs, the risk of bugs and vulnerabilities increases [1]. In 2021, the number of identified common vulnerability enumerations (CVEs) reached approximately 18,439, marking a substantial 184% surge from 2015 [22]. It is estimated that up to 70% of the time and effort of the IC development cycle is spent on the verification activities [11, 20], which highlights the prominence of verification [8].

The two popular verification methods are the dynamic [13] and formal verification [10] methods. However, both dynamic and formal verification techniques have failed to match the pace of ever-increasingly complex IC and SoC and are less efficient in detecting bugs [5, 16, 24] due to the lack of scalability for complex SoC designs. Hence, there is a need for efficient and automated verification methodologies that can detect bugs compatible with the current IC design and verification flow. Hardware fuzzing has been introduced in recent years [15, 17, 18, 24] to surpass the limitations of the latter. Fuzzing is a widely used software testing methodology for bug detection in software applications. Fuzzing, in simple terms, is bombarding the software with test cases and analyzing for any invalid targets, such as memory crashes.

RFuzz [17] is one of the first hardware fuzzing frameworks proposed in the literature. The hardware-fuzzing frameworks aim at fuzzing CPU designs for vulnerability detection. Later, some of the works proposed translating the hardware as software for the adaption of software fuzzers [24], whereas some works proposed fuzzing hardware as hardware [5, 15, 16] which will be discussed in Section 3. In this work, we present a comprehensive overview of the research into hardware fuzzing frameworks. The main contributions of this work are :

- We lay out the fundamental principle of the Design Verification (DV) techniques and its limitations.
- A comprehensive overview of the research into hardware fuzzing frameworks inspired by software fuzzing.
- We classify the several fuzzing frameworks based on their fuzzing methodologies.
- The overlooked challenges and fundamental issues with deploying existing tool flows for hardware fuzzing and the future directions for this field of research is discussed.

2 BACKGROUND

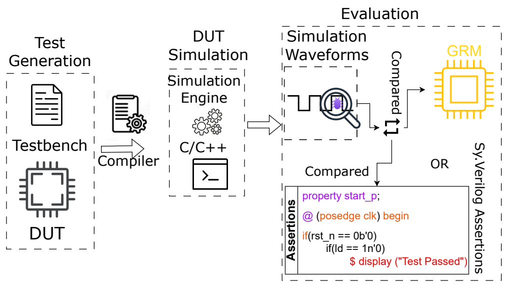
In this section, we offer a concise summary of the conventional verification techniques utilized in contemporary IC design processes. Furthermore, we highlight the limitations of these methods and explore how hardware fuzzing emerges as a solution.

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3658697

2.1 Formal Verification

Formal verification is one of the conventional methods in hardware verification techniques that ensure the correctness and reliability of complex integrated circuits and electronic systems [12, 13]. It encompasses a diverse range of mathematical theorems and logical techniques to prove the correctness of hardware designs against specified properties. Formal verification is supported by commercial EDA tools such as Cadence Jasper Gold and Synopsys VCS Formal which requires System Verilog Assertions (SVA) properties to cover the design space for verification. Formal verification has gained significant traction due to its ability to mitigate deep flaws in hardware design. However, the existing formal verification techniques are crippling due to their inability to cope with increasingly large complex modern processor designs and the need for expert knowledge [10, 16].

2.2 Dynamic Verification



```
property start_p;
@ (posedge clk) begin
if(rst_n == 0b'0)
    if(ld == 1n'0)
        $ display ("Test Passed")
```

Figure 1: Dynamic Verification

Dynamic hardware verification techniques (a.k.a runtime bug detection) **simulate or emulate the design-under-test (DUT) with a few test cases to verify the functionality of the design.** The three cardinal steps in DV are ***1) Test Generation 2) DUT Simulation 3) Evaluation*** as shown in Figure 1. DV engineers craft a sequence/series of a few input test vectors to simulate the DUT. Once the test vectors are generated, the HDL of the design can be simulated through various commercially available and open-source EDA tools for extracting the output responses. Test evaluation is performed by monitoring the outputs of the hardware for a given input stimuli against the Golden Reference Model (GRM) or System Verilog Assertions (SVA). The model is examined for any violation against the defined assertion properties or the behavior of the GRMs. However, existing DV frameworks have scalability limitations as the input space is exponentially large for CPU designs and deployment bottlenecks [16].

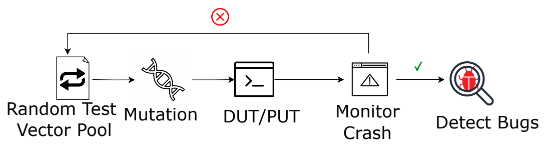


Figure 2: Overview of Software Fuzzing

3 HARDWARE FUZZ ODYSSEY

To surpass the limitations in the existing traditional DV frameworks, hardware fuzzing [16, 17, 24] has gained traction due to its popularity in the software testing community. The concept of fuzzing primarily involves: ***1) random test case generation and mutations; 2) monitoring the DUT/program-under-test (PUT); and 3) analyzing for bugs or errors*** as shown in Figure 2.

The core concept behind traditional fuzzing begins with the generation of **random acceptable test case generations** (i.e., input stimuli). To efficiently cover the DUT's state space, most fuzzers **apply mutation algorithms to generate new test cases** in contrast to dynamic verification. The input stimuli are then fed to the DUT/PUT for monitoring the running status of the program and is subjected to record any crash during the period of fuzzing. The monitored output is analyzed for bugs. The DUT/PUT is bombarded with test input until a crash is recorded. Software fuzzers analyze crashes for bug detection, whereas in hardware fuzzing, the expected outcome from DUT is verified against assertions or expected output from the GRM. Fuzzing incurs low deployment costs and a reduced verification period for testing complex designs. Depending on the available information regarding the DUT, fuzzing techniques can be broadly classified into three types: i) Blackbox fuzzing , ii) Greybox fuzzing, and iii) Whitebox fuzzing.

In Greybox Fuzzing (GF) such as American Fuzzy Lop (AFL) [24], the fuzzer has limited knowledge about the DUT while extracting the peripheral information about the data flow, control flow, data format, protocols, and high-level architecture. These fuzzers discard the primary source code after the necessary instrumentation for the program is added. The instrumented binary is subjected to monitoring based on the coverage reports.

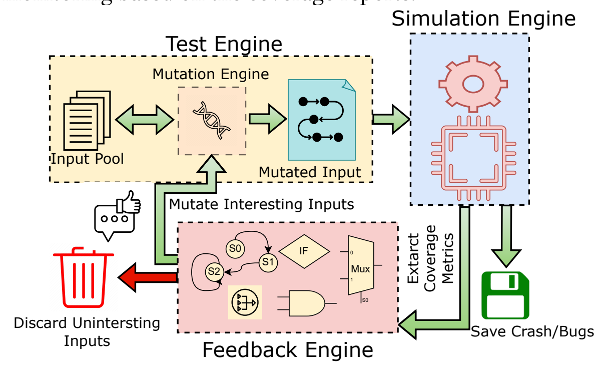


Figure 3: Coverage Greybox Fuzzing

Coverage-based Greybox Fuzzing (CGF): In CGF, the fuzzing is aimed at achieving maximum code coverage through feedback engines (i.e., coverage metrics). In hardware, the coverage metrics that can be used are Finite State Machine (FSM), line, conditional, and MUX toggle. As shown in Figure 3, a set of input seeds is stored and passed to the mutation engine that performs mutation operations to generate multiple input seeds. During the runtime, the coverage reports are extracted based on the input provided to the DUT and given as feedback to the mutation engine. Based on the coverage feedback, the mutation engine further mutates the interesting input seed to generate a new set of seeds. The uninteresting input seeds are discarded from the input pool. The design is simulated with these seed inputs, and any potential crashes are saved for analyzing the vulnerabilities.

Directed Greybox Fuzzing (DGF): Hardware and Software designs often undergo revisions for updating a component for better performance (i.e., incremental designs). To fuzz such incremental designs, DGF is used for fuzzing at a particular region rather than the whole DUT/PUT, resulting in reduced verification time.

These state-of-the-art fuzzing-based DV frameworks are competent to fuzz SoCs, CPUs, and standalone IP blocks. Based on fuzzing methodologies, the existing fuzzing frameworks can be classified into **1) Direction Adoption of Software Fuzzer for Hardware 2) Fuzzing Hardware as Software 3) Fuzzing Hardware as Hardware.**

3.1 Direct Adoption of Software Fuzzer for Hardware

As a solution to the existing shortcomings such as deployment bottleneck (i.e., the inefficiency of coverage tracing) for complex designs, RFuzz [17] is the first FPGA emulation-based hardware fuzzing technique introduced, as shown in Figure 4. The RFuzz translates the target HDL to Flexible Intermediate Representation of RTL (FIRRTL) such that instrumentation is leveraged through compiler passes, enabling the test harness generator to design a wrapper for the RTL design. The top-level hardware module input pins, which take in values at each test cycle, are connected to the testing tool. The input pins are concatenated to form a bit vector and are mapped to a series of bytes representing the input values. To ensure the tests are deterministic and repeatable, the RTL should be reset to a known test before each test execution. *MetaReset* (for resetting the registers to zero for enabling DUT reset) and *Spare memories* (to reset the memory locations that have been written in previous test execution) are deployed to enable quick RTL reset without modifying the DUT characteristics. Furthermore, the instrumentation is appended to enable *MUX control coverage* in hardware, ensuring FPGA-synthesizable coverage extraction. RFuzz utilizes the AFL-based mutation functions. The AFL fuzzer efficiently communicates with the FPGA through high-speed Direct Memory access (DMA), as shown in Figure 4.

3.2 Fuzzing Hardware as a Software

In contrast to RFuzz, Trippel *et al.* [24] proposed fuzzing hardware-like software rather than porting software fuzzers directly on the hardware designs. The cardinal aspect of fuzz hardware like software is driven by the executable software version of the hardware provided by the hardware simulation tools. The translated software version of the hardware is compiled, instrumented and fuzzed using AFL for bug detection.

In [24], the authors make use of Verilator to translate the hardware to an equivalent software model as shown in Figure 4. Verilator is a popular open-source tool that leverages the equivalent software model in C++ for the given System Verilog of the RTL design. To trace the hardware coverage in the software domain, it exploits the seamless binary instrumentation provided by the AFL fuzzer to trail code coverage. With edge coverage from AFL, both code and functional coverage of the hardware in a software domain can be monitored. The fuzzing cores are monitored for crashes and are evaluated against SVAs. This work monitors for crashes such as improper memory mapping, errors in registers, buffer overflow, and floating-point computation for vulnerability detection, as the hardware is modeled as software.

HyperFuzzer: In contrast to the fuzzing techniques discussed above, works such as [9] focus on fuzzing SoCs. It employs Hyperproperties [9], which are higher-level properties that describe security policies by comparing the behavior of instances of a system. The concept of hyperproperties is commonly employed in analyzing security protocols and detecting bugs in a system. These hyperproperties can be used for laying out the SoC security specifications. HyperFuzzer, as shown in Figure 4 [19], encompasses defining SoC security properties that adhere to confidentiality, integrity, and noninterference expressed in HyperPLTL (Hyper Past-time Linear Temporal Logic). Hyperfuzzer complies with CGF with High-Level coverage metrics. The famous Verilator is used for SoC RTL simulation, which is further instrumented to collect coverage metrics. AFL fuzzes the instrumented code to find potential bugs and is evaluated using a property checker by comparing it against the security property specifications (i.e., hyperproperties) in HyperPLTL. Alternatively, Soc Fuzzer [14] simulates the SoC in an FPGA framework, which fuzzes the SoC based on the cost function.

3.3 Fuzzing Hardware as Hardware

Recent advancements in hardware fuzzing have embraced a domain-specific approach to eliminate the need for translating hardware to software, along with the associated preliminary tasks and equivalence checks. TheHuzz [16], DifuzzRTL [15], HyPFuzz [6] and ProcessorFuzz [5] adeptly integrate hardware fuzzing into conventional industry-standard hardware design and verification workflows.

Ⓐ ***TheHuzz:*** The three pivotal components of TheHuzz include the Seed Generator, Stimulus Generator, and Bug Detector, as illustrated in Figure 4. Operating at the instruction set architecture (ISA) abstraction level, the Seed Generator furnishes the input seed in the form of instruction sequences. The Stimulus Generator then mutates the instruction at the binary level, ensuring that all bits of the instruction undergo mutation to test the processor with illegal instructions. This includes mutating opcode bits and data bits to unveil unexplored datapaths.

The RTL design of the processor is simulated with the binary format of the instruction using Synopsys VCS, a tool entrenched in the semiconductor industry for several decades. Synopsys VCS traces code coverage through various metrics, including branch, condition, toggle, FSM, and functional coverage. Based on these coverage metrics (the feedback engine), optimal weights are assigned to each instruction-mutation pair, and uninteresting pairs are discarded. Bugs are identified by simulating the ISA emulator (GRM) and comparing the behavior of the RTL design against the GRM behavior. Seamless compatibility with traditional EDA tools and requiring minimal additional effort are some of the unique aspects of this work.

Ⓑ ***ProcessorFuzz and DifuzzRTL:***. Other prominent works associated with hardware-specific fuzzing to detect CPU bugs are the ProcessorFuzz as shown in Figure 4 [5], and DifuzzRTL [15]. At an elevated stratum of hardware abstraction, a Central Processing Unit (CPU) manifests as an intricately designed FSM. It is indispensable to monitor these transitions of CPU states for potential bug detection for a given assembly program. ISA simulators are used to simulate the functional behaviors of the CPU, serving as a reference model for the existing fuzzing frameworks. Rather than using coverage metrics from the RTL simulation, ProcessorFuzz exploits Control Status Registers (CSRs) from the ISA simulator as coverage metrics. The CSR values and the transitions in them represent the changes in the architectural state flow of the CPU.

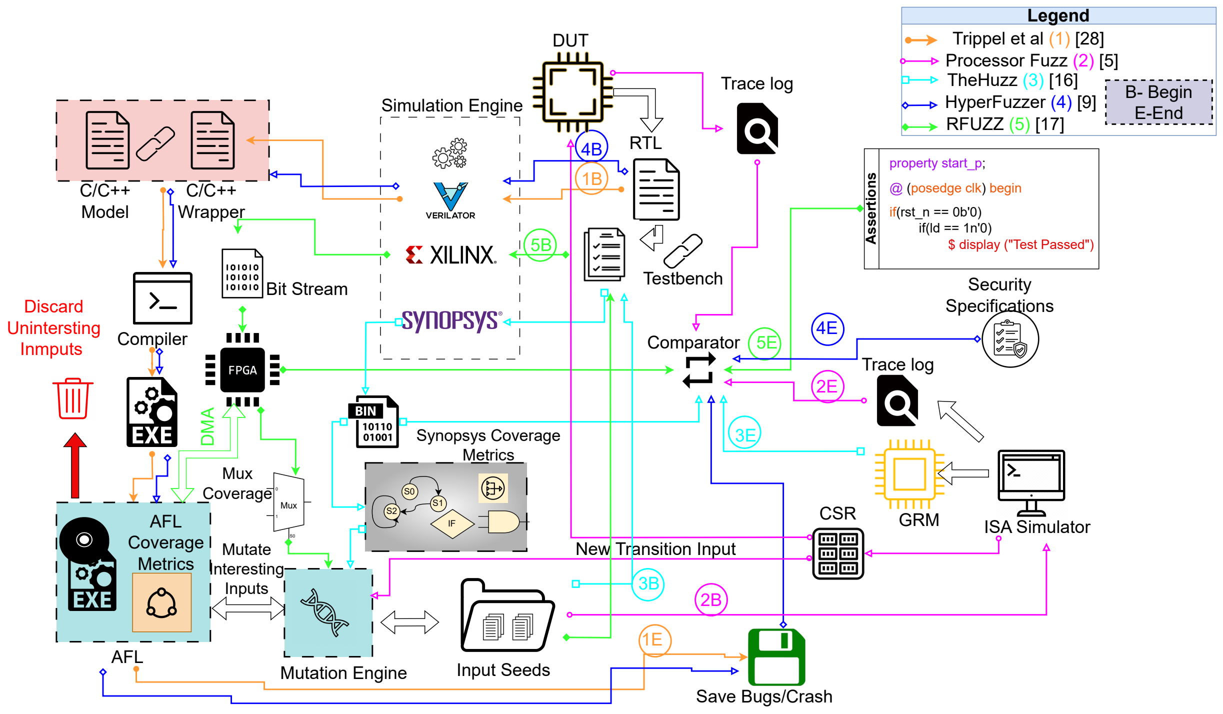


Figure 4: Hardware Fuzzing Frameworks

The input pool for the ProcessorFuzz encompasses a pool of instruction at the assembly level adhering to the target ISA upon which mutation operations are performed. Rather than simulating the RTL with the mutated input, ISA simulator simulates the targeted CPU with the mutated input sequences. During the simulation, the simulator initiates trace logs from which CSR transitions, program counter, and disassembled instructions can be obtained. The trace logs of the RTL and ISA emulators are compared to detect CPU vulnerabilities. In contrast, DifuzzRTL captures the FSM state transition of the RTL design through static analysis of the small group of registers, which plays a vital role in controlling the states of the CPU. The feedback is given through the register coverage in RTL and is compared against the ISA for evaluation of vulnerability detection.

Ⓒ***HyPFuzz:*** The increasing complexity of the CPU designs mandates the hardware fuzzing frameworks to penetrate deeper into the designs to reach all possible coverage points. Though the existing hardware fuzzing frameworks are proficient in bug detection, they fail to explore the entire design space of the CPU design. For instance, TheHuzz fuzzing framework was able to cover only 63% of the total design space states in the processors, paving the way for vulnerabilities in the uncovered regions. To alleviate the above, a hybrid fuzzing framework *HyPFuzz* was proposed encompassing of formal verification and fuzzing verification techniques in tandem. In this hybrid approach, the fuzzer is led by the formal verification tools reaching hard-to-reach design spaces, upon which the fuzzer exploits the hard-to-reach design spaces to detect bugs and vulnerabilities. This can be leveraged by amalgamating commercially available EDA tools such as Cadence JasperGold, and Synopsys VCS Formal with the fuzzing frameworks such as *TheHuzz*.

The HyPFuzz selects the uncovered region or points of the RTL design through the *point selector* upon which the formal verification tools can verify. To facilitate the formal tools with the System Verilog Assertions (SVA) properties, a *property generator* generates cover properties for the selected points of the point selector. However, the inherent nature of the formal and the fuzzer tools is incompatible as the former relies upon the assertion properties of the design and the latter relies upon the coverage points. To address this challenge, a *test case converter* translates the boolean assignment signals emitted by the formal tools during the property proving stage into a seed of inputs. The fuzzer can utilize these seeds of inputs to stimulate the DUT facilitating deep granularity. For a seamless transition from formal tools to the fuzzer, a dynamic timing scheduler is deployed and the fuzzing process ends when the scheduler terminates it. This hybrid method of fuzzing is 11x faster than TheHuzz framework and has unraveled three new hardware vulnerabilities.

Ⓓ*SoCFuzzer:* Alternatively, SoC Fuzzer [14] directs the fuzzing based on the security properties (generic cost function) that detects vulnerabilities in the DUT. The SoC Fuzz leverages the fuzzing through AFL guided by cost functions to proliferate mutations towards unexplored regions of the design. The SoC under test is simulated in an FPGA platform and instrumentation on the DUT is appended based on the security policies. The FPGA-based emulation facilitates the monitoring of real-time internal HW signals and the instrumentation appends real-time observation points with respect to the vulnerability point of interest. The fuzzer engine executes the program with fuzzed inputs with evaluation metrics as shown in Table 1 for faster convergence and will be compared against a database of vulnerabilities.

Table 1: Existing Fuzzing Framework

Framework	Fuzzer Type	Input	Simulator	Coverage Metric	Target Design	Evaluation	Bugs Reported
RFUZZ [17]	HW Fuzzer (1)	Series of bits	Any	Mux Toggle	Peripherals, RISC-V	Assertion	0
Li et al [18]	HW Fuzzer (1)	Series of bits	PyRTL	Mux Toggle	RISC-V Opencore 1200	Assertion	0
DifuzzRTL [15]	HW Fuzzer (1)	Assembly	Any	Register Coverage	RISC-V CPU	GRM	16
Trippel et al [24]	SW AFL Fuzzer (2)	Byte Sequence	Verilator	Edge Coverage	AES,HMAC KMAC, Timer	SW crashes	0
HyperFuzzer [9]	SW AFL Fuzzer (2)	Series of bits	Verilator	High-level	SoC	Assertion	0
DirectFuzz [4]	HW Fuzzer (3)	Series of bits	Verilator	MUX	Peripherals, RISC-V	Assertion	0
TheHuzz [16]	HW Fuzzer (3)	Assembly	Synopsys VCS	FSM, Branch,toggle, conditional	RISC-V	GRM	10
Processor Fuzz [5]	HW Fuzzer (3)	Assembly	Verilator	Control path register, ISA-tranistion	RISC-V	GRM	8
SoCFuzzer [14]	HW Fuzzer (3)	Byte Sequence	Xilinx ISA	Randomness, target output, input coverage	SoC	Database	0
HyPFuzz [6]	HW Fuzzer (3)	Assertion Cover properties, Byte Sequence	Jasper Gold, Synopsys VCS	FSM, Branch,toggle, conditional	RISC-V	GRM	13
WhisperFuzz [3]	HW Fuzzer (3)	Assembly	Synopsys VCS	FSM, Branch,toggle, conditional	RISC-V	Trace Properties	12
SIGFuzz [21]	HW Fuzzer (3)	Assembly	Verilator	NA	RISC-V	Trace Properties	5

4 CHALLENGES

Nonetheless, these cutting-edge frameworks are not without certain challenges. Employing software fuzzers directly onto the RTL poses its own set of challenges. For instance, a common definition for a bug is an outcome marked by crashes. However, such scenarios do not exist in the hardware realm. Hardware, instead of crashing, issues often manifest as malfunctions or failures in the form of faulty output. Because of this, employing AFL directly on the hardware may fail to detect any hardware vulnerabilities [17] as shown in Table 1. To alleviate this, Verilator is used for translating hardware designs to software models such as binary executables on which AFL is deployed [24]. However, there arises a question on the equivalency between the hardware and the translated hardware [16]. The verilator fails to capture inherent hardware behaviors such as signal transitions, FSMs, and floating wires described in HDL languages to software construct. In addition, AFL instrumentations and coverage metrics are not supported for HDL constructs [16]. During the evaluation phase, fuzzing frameworks rely upon SVAs. The insertion of SVAs for CPU designs leads to instrumentation overheads and can only examine pre-defined conditions rather than uncover unknown vulnerabilities.

As a solution to the limitations with the latter, fuzzing hardware as hardware was proposed [5, 16] to retain the the hardware characteristics. Although this method was able to detect new bugs and vulnerabilities, these works detect bugs associated with the functional behaviors of the design but not for parametric behaviors such as temporal characteristics and side-channel attacks. In addition, the Huzz framework covers only 63% of the DUT's design space leaving one-third of the design space unexplored for a potential vulnerability. Although Hybrid Fuzzers [6], are claimed to be 11x faster than the Huzz framework, the increase in coverage is 1% (64%) still leaving the remaining of the design uncovered.

These fuzzing frameworks rely upon GRMs for evaluation. The availability of third-party GRMs and access to the source code and internal signals are limited GRMs. The other way to depend on the GRMs is through software models. The CPUs can be simulated in ISA simulators (i.e., software simulations) to verify the functionality of the processors. For instance, the RISC-V community has crafted ISA simulators for RISC-V processors. Even though the GRMs are carefully curated to eliminate bugs, these software are designed by developers and are still prone to human errors and bias.

Recently, to detect the bugs associated with parametric behaviors such as timing and side-channel attacks, SIGFuzz [21] and WhisperFuzz [3] were proposed. Though these works detect bugs with the timing behaviors, these frameworks require a deep knowledge of micro-architectural event details of the design. The current fuzzing frameworks are limited to fuzzing either a CPU or a standalone peripheral IP or an SoC. None of the fuzzing frameworks provide flexibility to be adaptable across the spectrum of IC designs. Hence, there is a need for a unanimous fuzzing framework to fuzz all the entities associated with hardware design considering all functional and timing behaviors.

5 FORWARD PATH FOR FUTURE FUZZING

The coverage metrics play a vital role for exploring uncovered regions of the designs to detect functional vulnerabilities and thus require meticulous planning for design-based coverage selection. Machine Learning (ML) has proved its prominence in bug detection capabilities at a reduced verification time. On the other hand, the current generation ICs are becoming complex by adopting third-party IPs and the unavailability of GRMs should not impede the fuzzing. Based on these factors, we lay out three promising future directions of hardware fuzzing.

Design-aware selection of coverage metrics: In the hardware fuzzing and verification, the coverage metrics play a prominent role in not only defining the input seed corpus but also in bug detection capability. The existing works solely consider logical metrics to define the coverage. The coverage metrics in the existing literature range from the simple multiplexer (MUX) toggling to the control path status register (CSR) values. On the other hand, capturing all the coverage metrics from all the coverage sites introduces monitoring and analyzing overheads. Considering the emerging attack patterns relying solely on logic functionality is inefficient, as such one needs to consider parametric properties (such as temporal behavior) of the system. Capturing the pre-defined coverage metrics from all or statically defined sites (gates or nodes) of DUT leads to large instrumentation overheads, redundancy, inefficient bug detection, and state explosion challenges. In other words, it is pivotal to consider relevant coverage metrics from coverage sites for efficient vulnerability detection with low overheads. To circumvent this, a design-aware selection of coverage metrics is a potential solution that one could explore.

ML-based Hardware Fuzzing: In the evolving landscape of hardware security and verification, the rising prominence of Machine Learning (ML) has opened avenues for minimizing human intervention in verification engineering, marking a recent stride in advancement. Hardware verification, executed at abstraction levels, necessitates precision in defining ML models to accurately encapsulate architectural specifications. ML algorithms can be leveraged to craft mutation engines for generating novel test seeds, thereby enhancing coverage and encompassing various functionalities. An intriguing avenue lies in the application of ML algorithms to fuzzing, presenting an unexplored domain ripe for exploration.

In the realm of hardware verification, recent strides have seen the integration of graph learning methods to represent circuits in terms of nodes. This innovative approach facilitates the deployment of hardware verification techniques for bug detection. The primary impetus behind embracing graph learning lies in its capacity to generalize, accommodate diverse topologies, exhibit scalability, and possess structural awareness. Given the inherent ability of graph learning to extract structural topologies, a thought-provoking idea emerges: deploying this methodology to ascertain design-aware coverage metrics, thereby pushing the boundaries of what is currently conceivable in the field.

Fuzzing in the Absence of Golden Reference Model: As discussed in earlier sections, the availability of GRMs and access to the source code and internal signals are impeding the efficiency and efficacy of hardware fuzzing. So, how to fuzz in the absence of GRMs is a thing one can explore. Can transfer learning be adopted in the absence of GRMs? Transfer learning has been used in different tasks, including in EDA applications. These are the possible research gaps still existing in the present hardware fuzzing frameworks.

6 CONCLUSION

In this survey paper, we have detailed the fundamental principle of hardware fuzzing inspired by software fuzzing and the frameworks of different fuzzing methodologies. In addition, we have illustrated the coverage metrics, target designs, EDA tools chosen by the fuzzing frameworks, and the number of bugs reported by each of them. This paper also states the limitations of each of the frameworks and their corresponding mitigations proposed to date. Lastly, we have also laid out the future research direction of hardware fuzzing for exploring uncharted territories in hardware design verification.

ACKNOWLEDGMENTS

This work was partially supported by the Commonwealth Cyber Initiative, an investment in the advancement of cyber R&D, innovation, and workforce development. For more information about CCI, visit www.cyberinitiative.org

REFERENCES

[1] Nitay Artenstein. 2017. BROADPWN: Remotely Compromising Android and iOS via a bug in BROADCOM'S Wi-Fi Chipsets. In *BlackHat USA*.

[2] A.Yen. 2012. Trends in the Global IC Design Service Market. In *Digitimes*. https://www.digitimes.com/news/a20120313RS400.html&chid=2 last accessed 3/4/24.

[3] 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. *arXiv preprint arXiv:2402.03704* (2024).

[4] 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)*. 529–534.

[5] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. Taylor, M. Egele, and A. Joshi. 2023. ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[6] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. HyPFuzz: Formal-Assisted Processor Fuzzing. 1361–1378.

[7] James C. Chen, Hsin Rau, Cheng-Ju Sun, Hung-Wen Stzeng, and Chia-Hsun Chen. 2009. Workflow design and management for IC supply chain. In *International Conference on Networking, Sensing and Control*. 697–701.

[8] Mingsong Chen and Prabhat Mishra. 2011. Property Learning Techniques for Efficient Generation of Directed Tests. *IEEE Trans. Comput.* 60, 6 (Feb 2011).

[9] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*. 51–65.

[10] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. Hardfails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Conference on Security Symposium*.

[11] Sai Manoj Pudukotai Dinakarrao, H. Yu, C. Gu, and C. Zhuo. 2014. A zonotoped macromodeling for reachability verification of eye-diagram in high-speed I/O links with jitter. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[12] Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin. 2017. Automatic Code Converter Enhanced PCH Framework for SoC Trust Verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 12 (2017), 3390–3400.

[13] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. 2015. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. *SIGPLAN Not.* 50, 4 (Mar 2015), 517–529.

[14] 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 *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[15] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *IEEE Symposium on Security and Privacy (SP)*.

[16] 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 *USENIX Security Symposium (USENIX Security)*.

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

[18] T Li, H Zou, Luo D, and Qu W. 2021. Symbolic simulation enhanced coverage-directed fuzz testing of RTL design. In *IEEE International Symposium on Circuits and Systems (ISCAS)*.

[19] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. 2020. HyperFuzzing for SoC Security Validation. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[20] Andreas Olofsson. 2017. Intelligent Design of Electronic Assets (IDEA) & Posh Open Source Hardware (POSH). https://www.darpa.mil/attachments/eri_design_proposers_day.pdf Last accessed: 3/3/2024.

[21] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. 2023. SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels.

[22] REDSCAN. 2021. 2021 has officially been a record-breaking year for vulnerabilities. https://www.redscan.com/news/nist-nvd-analysis-2021-record-vulnerabilities/#:~:text=2021%20was%20an%20especially%20difficult,50%20CVEs%20logged%20each%20day last accessed 3/2/24.

[23] TechInsights. 2017. Apple iPhone 15 Pro Teardown. In *TechInsights*. https://www.techinsights.com/blog/apple-iphone-15-pro-teardown last accessed 3/2/24.

[24] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *USENIX Security Symposium (USENIX Security)*.