GEmFuzz: Uncovering System-Level Vulnerabilities in SoCs via Emulation-Based Grey-Box Fuzzing

Shuvagata Saha, Ahmed Alhurubi, Tanvir Rahman, Hasan Al Shaikh, Sujan Kumar Saha,
Farimah Farahmandi, Mark Tehranipoor
Department of Electrical & Computer Engineering, University of Florida, Gainesville, United States
{sh.saha, aalhurubi, tanvir.rahman, hasanalshaikh, sujansaha}@ufl.edu, {farimah, tehranipoor}@ece.ufl.edu

***Abstract*—Security verification of modern System-on-Chip (SoC) designs is becoming increasingly challenging due to the growing integration of third-party IPs and the complexity of hardware-software (HW/SW) interactions. This escalating complexity broadens the attack surface, leading to a higher number of potential vulnerabilities and longer detection times. Consequently, verification engineers face increasing pressure to ensure robust security within tight development schedules. Traditional techniques such as formal verification and information flow tracking often suffer from poor scalability, state space explosion, and significant manual effort, necessitating expert-level design knowledge. Fuzzing-based methodologies, while promising, typically rely on the availability of a golden reference model and struggle to scale effectively, which limits their applicability. Furthermore, the increasing intricacy of HW/SW stacks in modern SoCs introduces new classes of system-level vulnerabilities that remain largely unaddressed by existing approaches. To address these challenges, we propose *GEmFuzz*, a hardware emulation-based grey-box fuzzing framework for SoC security verification. *GEmFuzz* uses a hardware emulation server to run the design under test (DUT) at near real-time speed, effectively addressing the scalability challenges. Also, it leverages a cost-function-guided fuzzer to generate intelligent input patterns for system-level vulnerability detection. We evaluate *GEmFuzz* on a RISC-V-based SoC and demonstrate its effectiveness in detecting a set of known system-level vulnerabilities. Additionally, it identifies two previously unknown vulnerabilities, highlighting the capability and promise of the proposed framework.**

***Index Terms*—System-on-Chip, Hardware Security, Fuzzing, Emulation, Security Verification**

I. Introduction

The design of modern System-on-Chip (SoC) architectures has become increasingly complex due to the integration of numerous third-party IPs and intricate hardware-software (HW/SW) interactions [1]. These systems often handle security-critical assets such as cryptographic keys, device configurations, personal data, etc. A breach of such assets can result in significant consequences, including reputational damage, personal harm, and malicious takeover of the SoC. Ensuring the confidentiality, integrity, and availability (CIA) of these assets is thus essential. However, as SoC designs evolve, so does the complexity of verifying their security. This poses substantial challenges to hardware verification engineers, especially under stringent time-to-market demands and manual verification workflows.

To address these growing challenges, there is a pressing need for security verification methodologies that are fast, scalable, automated, and require minimal manual effort or domain-specific expertise. Several approaches have been explored over the years, including formal verification [2]–[5] and information flow tracking [6]–[8]. While effective in principle, these techniques often suffer from key limitations such as (1) poor scalability, (2) state space explosion, (3) extensive instrumentation overhead, and (4) the requirement for expert-level understanding of the design under test (DUT). Consequently, the goal of a widely applicable, efficient security verification framework remains largely unmet.

Recently, fuzzing has emerged as a promising approach for identifying security vulnerabilities in hardware. Inspired by software security testing, fuzzing involves providing a DUT with unexpected or randomized inputs to trigger security violations while maximizing test coverage. Hardware fuzzing typically repurposes software-oriented tools such as American Fuzzy Lop (AFL) [9], LibFuzzer [10], and Honggfuzz [11], adapting them to operate in the hardware domain.

TABLE I: Comparison of Hardware Fuzzing Approaches

Method	Tgt.	Env.	GB	GRM	Coverage Metric	Vuln.
HyPFuzz [25]	SoC	Sim	–	✓	Branch	HW
TheHuzz [12]	CPU	Sim	–	✓	FSM, Stmt., Branch	HW
Cascade [13]	CPU	Sim	–	✓	FSM, Stmt.	HW
DifuzzRTL [15]	CPU	FPGA	–	✓	Control Reg.	HW
DirectFuzz [14]	IP	Sim	–	–	Mux	HW
FormalFuzzer [23]	SoC	FPGA	✓	–	Assertion, Cost Fn.	HW
HFL [19]	CPU	Sim	–	✓	FSM, Stmt.	HW
HyperFuzzing [26]	SoC	Sim	✓	✓	NoC, Biflip	HW
SoCFuzzer [22]	SoC	FPGA	✓	–	Cost Fn.	HW
MABFuzz [21]	SoC	Sim	–	✓	Branch	HW
PSOFuzz [20]	SoC	Sim	–	✓	Branch	HW
TaintFuzzer [24]	SoC	FPGA	✓	–	Cost Fn.	HW
Fuzzing HW as SW [27]	SoC	Sim	–	✓	Code/Branch	HW
ChatFuzz [16]	CPU	Sim	–	✓	Condition	HW
Genhuzz [17]	CPU	Sim	–	✓	Condition, Line, FSM	HW
GEmFuzz	SoC	Emu.	✓	–	Cost Fn.	SW+HW

Env.=Environment: **Sim.**=Simulation, **FPGA**=FPGA Prototyping, **Emu.**=Emulation, **GB**: Grey-box; **Stmt.** = Statement, **GRM**: Golden Reference Model needed; **Vuln.**=Vulnerability.
✓=Yes; –=No.

Several notable works have explored different facets of hardware fuzzing. TheHuzz [12] utilizes assembly-level instruction fuzzing to identify software-exploitable vulnerabilities in hardware designs. Cascade [13] employs asymmetric ISA pre-simulation to entangle control and data flows, thereby generating valid RISC-V programs that uncover deep design flaws. DirectFuzz [14] leverages a grey-box simulation-based approach to accelerate vulnerability detection. DifuzzRTL [15] combines register-coverage-based feedback with simulation and FPGA prototyping. ChatFuzz [16] and GenHuzz [17] explore the use of large language models (LLMs) to generate effective test inputs for simulation-based hardware fuzzing. SoCureLLM [18] extends this trend by introducing an LLM-driven framework for large-scale SoC security verification, combining automated vulnerability detection with policy generation to improve coverage and scalability. Additionally, reinforcement learning has been adopted in approaches such as HFL [19], PSOFuzz [20], and MABFuzz [21] to guide the fuzzing process more intelligently. SoCFuzzer [22] implements a cost-function-guided strategy on FPGA platforms to identify security issues at SoC level. FormalFuzzer [23] employs formal methods to reduce the input space and enhance fuzzing efficiency, while TaintFuzzer [24] uses taint analysis to infer promising seed inputs for faster convergence. Table I shows a comparative analysis of different hardware fuzzing approaches.

Despite these advances, existing fuzzing frameworks exhibit several drawbacks. Simulation-based methods often rely on the availability of a golden reference model (GRM), which may not exist for many

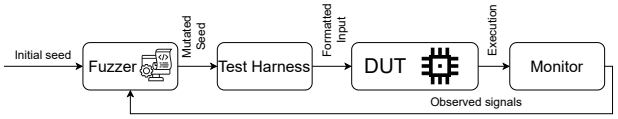


Fig. 1: Overview of a typical fuzzing framework.

designs. Moreover, their performance is heavily dependent on host machine speed, making large-scale simulations prohibitively slow. Besides, FPGA-based fuzzing approaches often depend on AFL variants that run on an operating system (OS) kernel executing on the DUT hardware, necessitating a custom OS built specifically for the DUT. Furthermore, both simulation and FPGA prototyping techniques typically require design instrumentation, which increases verification overhead, adds complexity, and demands considerable design expertise. Finally, many of these approaches primarily target hardware-level vulnerabilities and fall short of capturing vulnerabilities that arise from complex interactions across the HW/SW boundary.

To address these limitations, the next generation fuzzing methodology should be (1) applicable to arbitrary designs regardless of the required custom OS build or reference model availability, (2) capable of scaling across large SoC designs, (3) minimally invasive in terms of instrumentation, and (4) effective in identifying system-level vulnerabilities that stem from HW/SW interaction.

In this paper, we propose *GEmFuzz*, a scalable, emulation-based fuzzing framework tailored for uncovering system-level vulnerabilities in SoCs. To the best of our knowledge, this is the first work that utilizes a hardware emulation server in fuzzing-based security verification. *GEmFuzz* employs a security event-driven cost function to intelligently guide the fuzzing process without relying on a golden reference model or OS kernel. Our key contributions are as follows:

- We introduce *GEmFuzz*, a novel grey-box fuzzing framework that is implemented on a hardware emulation server.
- We propose an initial seed selection strategy to guide the fuzzer for early convergence.
- A lightweight, non-intrusive instrumentation strategy is developed to capture security events and send the feedback to the cost function of the fuzzer.
- We demonstrate the effectiveness of our approach through case studies on a RISC-V-based SoC that uncover both known and unknown vulnerabilities.

The remainder of this paper is organized as follows: Section II provides background information. Section III introduces the *GEmFuzz* framework. Section IV presents our experimental evaluation. Finally, we conclude the paper in Section V.

II. Background

A. Fuzzing

Fuzzing is an automated testing technique used to uncover vulnerabilities in hardware or software systems by continuously generating and injecting malformed or semi-valid inputs. As shown in Fig. 1, a typical fuzzing workflow begins with an initial seed, which is mutated to produce diverse input patterns. These inputs are then processed by a test harness that formats them into a valid transaction stream for the design under test (DUT). The DUT is instrumented to monitor relevant internal or output signals during execution. After each test run, the observed signal values are collected and analyzed by the fuzzer to assess coverage or detect anomalies. This feedback guides the mutation strategy for the next iteration, enabling the fuzzer to explore deeper and less deterministic execution paths to uncover hidden vulnerabilities.

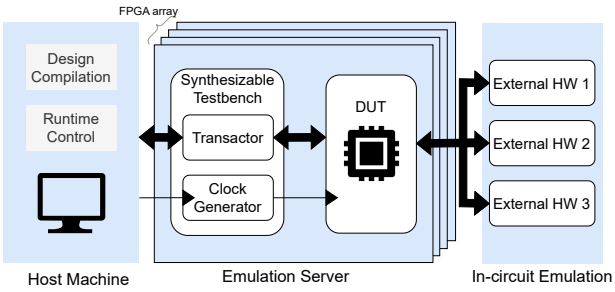


Fig. 2: Overview of Hardware Emulation.

The process repeats in a closed-loop fashion, progressively increasing the likelihood of uncovering subtle or corner-case bugs. In the hardware domain, fuzzing is typically categorized into two broad methodologies: (1) *fuzzing hardware as software*, where a software model of the hardware is built and fuzzed like a regular program [27]; and (2) *fuzzing hardware as hardware*, where the actual RTL design is exercised directly via simulation [12], [13], [19] or FPGA-based emulation [15], [22], [28]. The latter approach is particularly effective for revealing low-level hardware bugs that may not surface in an abstracted software model of the hardware.

B. Emulation

Emulation is a hardware-assisted verification technique that executes the design under test (DUT) on specialized hardware to achieve faster and more scalable verification. This process involves mapping and programming the SoC design onto the emulator, which typically consists of high-speed FPGA arrays. As illustrated in Fig. 2, the DUT can often be imported from the simulation environment with minimal modifications. Verification requires a synthesizable testbench comprising components such as a clock generator to control DUT timing and a transactor that manages communication with the DUT. The transactor, together with embedded probes, facilitates monitoring of I/O behavior and internal signals. The emulation process is controlled by the host machine, which performs two key tasks: (1) **design compilation**, including synthesis, place-and-route (P&R), and bitstream generation for the DUT; and (2) **runtime control**, wherein the bitstreams are loaded onto the assigned FPGAs in the emulation server to initiate and manage DUT execution. Modern emulators also support in-circuit emulation by communicating with external hardware in real-world environments, offering a circuit behavior close to a real system. This functionality supports various complex interfaces such as PCIe, CXL 2.0, Ethernet, USB, SATA, Display port, 5G testers, etc., which facilitates testing the DUT with real-world external devices. Due to their scalability, performance, and flexibility, emulation platforms are increasingly preferred over traditional simulation and formal verification approaches. Commercial emulation solutions, such as Synopsys ZeBu Server [29], Cadence Palladium Z2 [30], and Siemens Veloce CS System [31], enable verification of complex SoCs with significantly reduced turnaround time. As a result, emulation has become an essential tool in modern verification flow.

III. Proposed Framework: *GEmFuzz*

GEmFuzz is an emulation-based verification framework designed for security verification of RISC-V-based System-on-Chip (SoC) designs. The strength of *GEmFuzz* lies in ① detecting system-level vulnerabilities, ② scaling efficiently to verify large SoC designs, ③ monitoring security-critical events without relying on vendor-specific debug infrastructure (i.e., using Integrated Logic Analyzer (ILA) in Xilinx FPGAs) used in FPGA-based fuzzing approaches,

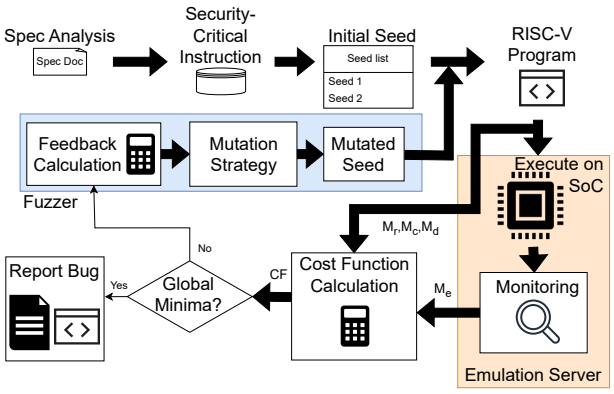


Fig. 3: *GEmFuzz* Framework.

and ④ operating without the need for a custom OS kernel. Unlike traditional coverage-driven verification approaches, *GEmFuzz* utilizes a cost-function-based feedback mechanism tailored specifically for security. Operating under a gray-box verification model, *GEmFuzz* assumes the verification engineer has limited internal knowledge of the design and does not require a golden reference model. Hence, the DUT is instrumented minimally, as only a set of selected security-critical signals is monitored. The framework performs fuzzing at the instruction level, directly stimulating low-level control and data paths. Additionally, specification-guided analysis is used to extract a set of security-critical instructions, which serve as the basis for generating initial fuzzing seeds. This targeted seeding accelerates convergence by biasing the fuzzer toward semantically meaningful program behavior. Once the fuzzed instructions are generated, these are executed on the DUT, and the outputs and monitored signal values are collected. The overall execution and feedback loop of the framework is illustrated in Figure 3, highlighting the key stages from specification analysis to vulnerability detection. Feedback derived from the cost function informs the mutation engine, guiding the generation of subsequent test inputs. After each execution cycle, the cost function is evaluated to determine progress toward reaching its global minima, which indicates the potential triggering of a security violation.

A. Spec Analysis

The first stage of the *GEmFuzz* framework is specification inspection. Alongside the RTL model, most SoC designs are accompanied by documentation detailing their intended functionality. Using these documents, a set of instructions are identified that are fundamental to execute security-critical operations such as access control mechanisms, privilege level checking, or memory protection. These are referred to as *security-critical instructions*. In the context of RISC-V architectures, security-critical instructions may include privileged CSR access instructions, protected load-store operations, etc. These *security-critical instructions* serve two key purposes: ① these are essential for selecting the initial seeds, as described in III-B, and ② these serve as critical reference points during cost function evaluation, particularly in determining whether a vulnerability has been triggered after test execution. Concurrently, a set of security-critical signals and registers is identified to be monitored at runtime. These typically include internal registers such as `mstatus`, `mtvec`, `mepc`, etc., as well as relevant bus transaction signals. This instrumentation enables the framework to monitor critical state transitions while maintaining compatibility with the gray-box model. Once the security-critical instruction set and signal list are defined, the framework proceeds to the initial seed selection stage.

B. Initial Seed Selection

The *GEmFuzz* framework begins its execution with the selection of an initial seed. This is a critical step that directly influences the efficiency of the fuzzing process. Rather than relying on a purely random seed, as is common in traditional fuzzing frameworks, *GEmFuzz* adopts a semi-directed approach guided by specification analysis to create the security-critical instruction set. From this set of instructions, *GEmFuzz* extracts the opcodes and completes its encoding by assigning randomized values to the remaining fields, such as immediate values, register indices, or function codes. The resulting fully-formed instructions are used as the initial seeds for the fuzzing campaign. This strategy ensures that the framework begins its exploration in a semantically meaningful region of the input space, increasing the likelihood of reaching security-critical execution paths early in the fuzzing process and accelerating convergence toward potential vulnerabilities.

C. Test Program Generation

Following the seed selection stage, the framework identifies instructions that are highly likely to expose security vulnerabilities based on their semantics. However, to execute these instructions meaningfully on the SoC, they must be embedded within a complete program that sets up the necessary hardware states. The test program generation step is responsible for constructing a valid RISC-V binary in which the fuzzed instruction is injected at a specific location, typically after preparatory instructions that configure the system into a relevant operational state. This stage ensures that the generated program respects instruction ordering and that the execution context is favorable for triggering the intended secure behavior. It avoids wasting cycles on irrelevant or non-security-critical execution paths. The generated program structure is designed to enhance the observability of violations and maximize the utility of each fuzzing iteration. For example, to test access control vulnerabilities, a representative program may first lower the system's privilege level and then attempt to execute a privileged instruction. If the design correctly enforces privilege separation, such execution should trigger an exception. Embedding fuzzed instructions into this structured format allows *GEmFuzz* to effectively evaluate whether the system behaves as expected under constrained and potentially adversarial conditions.

D. Program Execution and Event Monitoring

After program generation, the compiled RISC-V binary is executed on the DUT using an emulation server. The emulation server is responsible for managing this execution process and for monitoring security-critical signals and registers as identified in subsection III-A. It comprises two main components: the DUT and a synthesizable harness that interfaces with the host machine running the fuzzer. The harness is tasked with managing binary loading, controlling input signals, orchestrating execution, and extracting monitored data. Upon receiving a new test binary from the host, the harness loads it into the appropriate memory location within the SoC and configures essential control signals, such as clock and reset, to initiate execution. Once the SoC completes execution, indicated by a dedicated completion signal, the harness captures both internal signals (e.g., CSR values) and output signals, and transmits them back to the host.

A key advantage of this harness-based design is its modularity and minimal invasiveness. It requires no modification to the SoC itself. This makes the *GEmFuzz* framework portable across different SoC platforms, as only the harness needs to be adapted for each new

design. Additionally, the harness formats the extracted signals into a structure compatible with the framework's cost function evaluation logic, enabling seamless integration with the host-side fuzzing and feedback components.

E. Cost Function

To guide feedback and prioritize the discovery of security violations, *GEmFuzz* employs a cost function tailored for security-oriented fuzzing. It evaluates each fuzzing iteration and steers the mutation engine toward input patterns likely to expose vulnerabilities. The cost function integrates four weighted metrics: (1) *input diversity*, (2) *target similarity*, (3) *input coverage*, and (4) *event monitoring*. Together, these capture input randomness, alignment with security-relevant behavior, space exploration, and runtime correctness.

At each iteration i, the cost is computed as:

$$CF_i = w_r M_{r_i} + w_d M_{d_i} + w_c M_{c_i} - w_e M_{e_i} \tag{1}$$

where w_r, w_d, w_c, w_e are metric weights, and $M_{r_i}, M_{d_i}, M_{c_i}, M_{e_i}$ are their scores at iteration i for input diversity, target similarity, input coverage, and event monitoring metrics, respectively which are described below.

Input Diversity: Ensures structural variation among instructions to avoid redundancy. Let N_{fuzz} be the number of fuzzed instructions in the binary, $I_{k,j}$ and $I_{k,i}$ are the fuzzed instructions at fuzzing location k for j^{th} and i^{th} iteration, and L is the instruction length. The diversity score at iteration i is:

$$M_{r_i} = 1 - \frac{1}{(i-1)\cdot N_{fuzz}} \sum_{k=1}^{N_{fuzz}} \sum_{j=1}^{i-1} \frac{H(I_{k,i}, I_{j,i})}{L} \tag{2}$$

where $H(a,b)$ is the Hamming distance between a and b. As the fuzzer explores a more diverse set of instructions, M_{r_i} decreases, thereby decreasing the cost function. This metric ensures that the fuzzer thoroughly exercises the system to uncover security vulnerabilities by generating programs with complex control and dataflow entanglements.

Target Similarity: Quantifies closeness to a set of N_T security-critical instructions. For each fuzzed instruction $I_{k,i}$ at iteration i and location k, target similarity is calculated using Eq. 3:

$$M_{d_i} = \frac{1}{N_T \cdot N_{fuzz}} \sum_{k=1}^{N_{fuzz}} \sum_{j=1}^{N_T} \frac{(H(I_{k,i}, I_{T_{k,j}}))}{L} \tag{3}$$

where $I_{T_{k,j}}$ is the j^{th} security-critical instruction at k^{th} fuzzing location. Lower M_{d_i} indicates alignment with known risky operations. Note that this metric encourages the fuzzer to generate instructions that closely align with security-critical instructions. Integrating this metric with input diversity ensures a balance between random instruction generation and guided exploration toward security-relevant behavior.

Input Coverage: Promotes exploration of the input space. Let N_{unique_j} be the number of unique instructions seen at j^{th} fuzzing location, and B_{excluded} the number of fixed bits in the instruction. Then, input coverage is calculated by Eq. 4:

$$M_{c_i} = 1 - \frac{1}{N_{fuzz}} \sum_{k=1}^{N_{fuzz}} \frac{\sum_{j=1}^{i} N_{unique_{k,j}}}{2^{(L - B_{\text{excluded}})}} \tag{4}$$

Lower M_{c_i} reflects increased coverage, thus lowering the cost function. With the introduction of new, unique fuzzed instructions, this metric decreases and drives the cost function to a lower value.

Event Monitoring: Validates runtime behavior against security policies. Let K be the number of monitored events, and $\delta(E_{e,j}, E_{o,j}) = 1$ if expected and observed behavior match, 0 otherwise. Then, the event monitoring metric is evaluated by:

$$M_{e_i} = \frac{1}{K} \sum_{j=1}^{K} \Big(1 - \delta(E_{e_{i,j}}, E_{o_{i,j}})\Big) = 1 - \frac{1}{K} \sum_{j=1}^{K} \delta(E_{e_{i,j}}, E_{o_{i,j}}) \tag{5}$$

Higher M_{e_i} reflects greater deviation from expected secure behavior.

For example, if a higher-privilege instruction is executed from a lower-privilege mode without triggering an illegal instruction exception, it constitutes a divergence from the expected secure behavior. This deviation results in a non-zero value for this metric, which in turn drives the cost function toward a lower value.

F. Feedback and Mutation Strategy

The host machine runs an in-house modified version of AFL++ [32], enhanced with a custom mutator. This mutator leverages feedback from the cost function to guide the selection of mutation strategies. After the cost function (CF_i) is computed after i^{th} iteration, *GEmFuzz* checks if the cost function has reached the global minima. If the CF reaches the global minima, the framework reports the corresponding vulnerability along with the last mutated binary that triggered it. Otherwise, the fuzzer uses the computed feedback to guide the next round of test generation.

To quantify progress, the fuzzer calculates the Cost Convergence Rate (CCR), which captures the change in cost values across iterations. The CCR is defined in Eq. 6 as:

$$CCR = -\frac{CF_{\text{i}} - CF_{\text{j}}}{i - j} \tag{6}$$

where CF_{i} and CF_{j} are the cost function values at the i^{th} and j^{th} iterations. A positive CCR ($CCR > 0$) indicates that the cost is decreasing and the fuzzer is converging, so the current mutation strategy is retained. Conversely, a negative CCR ($CCR < 0$) suggests a lack of progress or regression, prompting the framework to switch to a different strategy.

The fuzzer selects from eight predefined mutation strategies (bit flip, byte copy, multi-bit flip, bit sliding, byte swapping, arithmetic operation, crossover with security-critical instruction, RISC-V operand mutation). Initially, one mutation strategy is chosen at random, but subsequent decisions are driven by CCR feedback. After mutation, the resulting ELF binary is generated and transmitted to the emulation server for the next iteration of execution.

G. Illegal/Invalid Instruction Execution

A unique feature of *GEmFuzz* is its ability to operate directly on compiled binaries, enabling execution of both legal and illegal instructions. It broadens the verification scope by allowing injection of syntactically invalid or architecturally undefined instructions into the DUT. These are used to evaluate whether the hardware raises exceptions or handles undefined behavior per the ISA specification. To implement this, the framework introduces a weighting parameter, w_{illegal}, which controls the likelihood of selecting illegal or non-standard instructions. A higher value of w_{illegal} increases the probability of generating invalid inputs, facilitating stress testing of exception-handling mechanisms, while a lower value biases execution toward valid instructions for focused analysis of specified behavior. This weight parameter is used to determine how many illegal instructions will be run on the DUT over all instructions. By operating at the binary level, *GEmFuzz* uncovers a broad class of vulnerabilities, including those arising from improper decoding, exception-handling flaws, and hardware/software interface violations. Its dual-pronged approach reveals issues in both hardware logic and cross-layer interactions.

TABLE II: Description of the security vulnerabilities

Index	Description	Security violation	Source	Reference
SV1	The system state does not transfer to lower privilege when `MRET` is executed	Privilege escalation	HW	CWE-266
SV2	U-mode is not implemented though mentioned in documentation	Integrity, Confidentiality	HW	CWE-250
SV3	M-mode CSRs accessed before access control check is performed	Integrity, Confidentiality	HW	CWE-1280
SV4	Insufficient granularity of access control of M-mode CSRs	Integrity, Confidentiality	HW	CWE-1220, CWE-1222
SV5	`mtvec` register can be modified from user-space application	Access control, Integrity	SW	CWE-1231, CWE-1233
SV6	Accessing M-mode CSRs doesn't raise illegal instruction exception for some CSRs	Confidentiality, Integrity	HW	CWE-1262
USV1	The processor executes codes from outside instruction memory boundary	Confidentiality	HW	CWE-125
USV2	The processor stalls when loading data from an unimplemented memory region	Availability	HW	CWE-754

IV. Experimental Setup and Results

A. Experimental Setup

We implemented *GEmFuzz* on an Intel Xeon Silver 4416+ workstation configured with 80 CPUs and 512 GB of RAM. This machine served as both the host for compilation and the runtime controller for the emulation server. All emulation experiments were conducted using the *Synopsys ZeBu Emulation Server 3* [29], while *Synopsys VCS* was employed in the same host machine for simulation-based comparisons.

As the DUT, we selected NEORV32 [33], a RISC-V-based SoC. To assess the framework's effectiveness in uncovering security flaws, we curated a benchmark suite consisting of six vulnerabilities. These included both *known vulnerabilities* obtained from a system-level vulnerability database [34] and *synthetically injected vulnerabilities* crafted by following descriptions from the *Common Weakness Enumeration (CWE)* [35] database.

A representative example of such a vulnerability (SV-1) is shown in Listing 1. According to the RISC-V specification, when a processor returns from a trap, it should restore the prior privilege level using the `mstatus` CSR's `MPP` bits (line 2). However, if the return privilege level is incorrectly hardcoded to machine mode (M-mode) as shown in line 3, the processor remains in M-mode (higher privilege level) after returning from a trap, regardless of the originating privilege. This flaw enables an attacker to escalate privileges persistently, thereby compromising the system's confidentiality, integrity, or availability (CIA) guarantees.

```
1 if CPU_EXTENSION_RISCV_U then
2   -- csr.privilege <= csr.mstatus_mpp;-- Correct
3   csr.privilege <= priv_mode_m_c;-- Vulnerability
4   csr.mstatus_mpp <= priv_mode_u_c;
```

Listing 1: SV-1 vulnerability in HW

Table II provides a detailed summary of all six vulnerabilities, including their type, insertion point, and security impact. The table also includes a description of the unknown vulnerabilities uncovered by *GEmFuzz*.

To guide the fuzzing process, we defined a set of control parameters. Among them, w_r, w_d, and w_c are associated with the DUT's input interface, and w_e targets internal signal behavior and output-side observation. Note that the same weights are used for each iteration, so $w_{r_i} = w_r$, $w_{d_i} = w_d$, $w_{r_c} = w_c$, and $w_{e_i} = w_e$ for all i. The weights assigned to these parameters were carefully balanced to ensure equitable exploration of both input stimuli and observable outputs. Additionally, approximately 10% of the generated instructions were designed to be illegal, malformed, or semantically invalid, thus challenging the DUT's ability to detect and safely handle erroneous inputs under adversarial conditions (see subsection III-G). Table III summarizes the experimental parameters and weight settings used during evaluation.

TABLE III: Parameter configurations for *GEmFuzz*.

w_r	w_d	w_c	w_e	L	$B_{excluded}$	$w_{illegal}$
0.25	0.25	0.25	0.25	32	7	10%

TABLE IV: Impact of Harness size

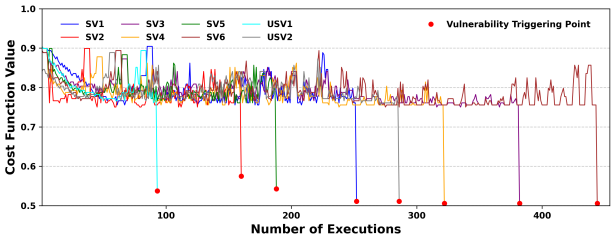
Design	Registers	LUT	LUT6	LoC
NEORV32	2257	2466	1019	21220
NEORV32 + Harness	2582	2775	1055	21399
Harness size w.r.t. DUT	14.4%	12.53%	3.53%	0.84%

B. Results

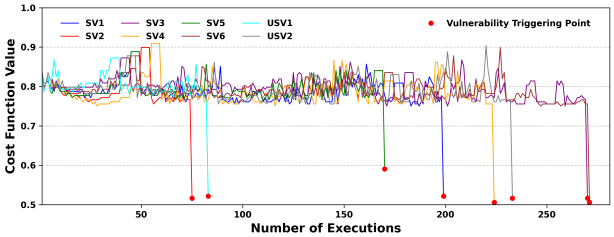
1) Impact of Test Harness: To enable emulation of the DUT, we developed a synthesizable testbench that manages its I/O interfaces. Within *GEmFuzz*, this testbench also functions as a harness, facilitating communication between the fuzzer running on the host machine and the DUT operating on the emulation server. Table IV presents a breakdown of the hardware resource utilization for both the SoC and the test harness. Specifically, the harness accounts for 14.4% of the DUT's register usage, 12.53% of its LUT usage, and only 3.53% of LUT6 resources. These utilization figures are obtained automatically through the emulation server's compilation tools. Given that the emulation server comprises multiple FPGA dies capable of hosting large-scale designs, the additional overhead introduced by the harness remains negligible. Moreover, the harness itself constitutes only 0.84% of the DUT's code size, thereby minimizing the setup effort required from the verification engineer.

2) GEmFuzz efficiency: Table V compares the performance of different *GEmFuzz* configurations with a baseline fuzzing approach without feedback, measured in terms of the number of execution cycles required to trigger each vulnerability. The second column reports the execution cycles for fuzzing without feedback, while the third column shows the results for *GEmFuzz* initialized with random seeds. The fourth column presents the performance of the complete *GEmFuzz* framework, enhanced by incorporating smart initial seed selection. For six out of the eight evaluated vulnerabilities, *GEmFuzz* demonstrates superior efficiency, achieving speedups ranging from 13.8× to as high as 100× over the baseline. For USV1, *GEmFuzz* performs comparably to the feedbackless approach, while for SV5, it performs worse. Notably, USV1 and USV2 represent previously unknown vulnerabilities, underscoring the framework's ability to uncover novel security flaws in addition to known ones. Fig. 4 shows the cost function values at different iterations while detecting the security vulnerabilities.

To better understand the cases where *GEmFuzz* performed comparably or worse than the feedback-less fuzzing approach, we examined the underlying causes of the SV5 vulnerability. In this case, the source of the vulnerability lies in the software stack: the value stored in the `mtvec` register is modified before control is transferred from machine mode to the user application. The `mtvec` register holds the address



(a) Cost-function convergence with *random* initial seeds.



(b) Cost-function convergence with *smart* initial seeds.

Fig. 4: Cost-function evaluation under two seeding strategies: (a) random seeds vs. (b) smart seeds.

TABLE V: Vulnerability detection speed for different approaches

SV	Fuzzing w/o FB	GEmFuzz$_r$	GEmFuzz	Improvement	Initial seed impact
SV1	3077	252	199	15x	26%
SV2	3077	160	75	41x	113%
SV3	7015	382	270	26x	41%
SV4	3366	322	244	13.8x	32%
SV5	15	188	170	0.08x	10%
SV6	8215	444	271	30.3x	63%
USV1	79	93	83	0.95x	12%
USV2	23349	286	233	100x	22%

FB = Feedback, GEmFuzz$_r$ = GEmFuzz with random initial seed

of the trap handler, which the processor uses to determine the next instruction to execute when a trap or invalid exception occurs. If an attacker alters the value of `mtvec` through the software stack, an illegal instruction executed from user space may redirect the processor to a malicious memory region. Since the processor fetches the next instruction from the address stored in `mtvec` upon detecting an exception, this manipulation results in a privilege escalation vector. In the absence of feedback, the fuzzer is more likely to generate invalid instructions with higher frequency, which explains its faster discovery of this vulnerability compared to *GEmFuzz*. Note that a GRM-dependent simulation-based approach would likely fail to detect this vulnerability without extensive manual review. This is because the software-level flaw corrupts the hardware state identically in both the reference model and the DUT, leading both to jump to the same compromised address, thereby concealing the mismatch.

In the case of USV1, *GEmFuzz* discovered that a specific sequence of instructions could force the processor to execute instructions outside the designated instruction memory region. Analysis of the generated binary revealed that this vulnerability was triggered through the use of jump instructions with carefully crafted immediate values. The feedbackless fuzzer, which relies on purely random input generation, is more likely to produce such large or malformed immediate values early in the execution, thereby increasing its chances of triggering this vulnerability quickly. While *GEmFuzz*'s feedback-guided instruction generation offers a more targeted and robust exploration strategy overall, it typically requires several initial execution cycles to begin exploring such corner-case behaviors. Notably, this vulnerability could only be uncovered through low-level, instruction-level fuzzing. If fuzzing were conducted at higher abstraction levels, the compiler and linker would likely optimize away or prevent such unsafe execution paths, rendering this class of vulnerabilities undetectable.

TABLE VI: Runtime comparison for Simulation vs. Emulation

SV	Execution Cycles	Simulation Time (s)	Emulation Time (s)	Improvement
SV1	199	105	11	9.5x
SV2	75	37	7	5.2x
SV3	270	152	19	8x
SV4	244	133	12	11x
SV5	170	84	10	8.4x
SV6	271	150	13	11.5x
USV1	83	43	7	6.1x
USV2	233	120	9	13.3x

3) Impact of Initial Seed Selection Strategy: Table V also illustrates the impact of initial seed selection on the performance of the *GEmFuzz* framework. Across all evaluated cases, the use of smart initial seeds led to performance improvements ranging from 10% to 113%, with an average gain of 29.9% (geometric mean). These results underscore the significance of effective seed selection in accelerated vulnerability discovery. The benefit is further visualized in Figures 4a and 4b, which show the number of execution cycles required to reach the global minimum of the cost function using both random and smart initial seeds. In *GEmFuzz*, the cost function reaches its global minimum when the DUT exhibits behavior that diverges from the expected secure specification, triggering the event monitoring metric (M_e). This divergence results in a sharp drop in the cost function, signaling the successful detection of a vulnerability.

4) Comparison with simulation: Although *GEmFuzz* is designed as an emulation-based verification framework, it can also operate in a simulation environment with minimal modifications to the test harness. Table VI presents a comparison of *GEmFuzz*'s performance in simulation vs. emulation, measured in terms of wall-clock time. On average, the emulation-based implementation achieves a speedup of 8.74× (geometric mean) in reaching the global minima of the cost function. Note that the emulation speed depends on several factors, including the number of monitored signals, execution granularity, and communication overhead with the host machine. By selecting optimal values for these parameters, the emulation framework can achieve performance that is orders of magnitude faster than simulation in practical verification scenarios.

V. Conclusion

Existing hardware fuzzing approaches face significant shortcomings in speed and scalability when detecting system-level vulnerabilities arising from complex hardware-software interactions. To address this challenge, we proposed *GEmFuzz*, an emulation-based fuzzing framework designed to uncover system-level vulnerabilities. We demonstrated its effectiveness by detecting six known and two previously unknown vulnerabilities across a representative RISC-V SoC. These results highlight *GEmFuzz*'s potential to detect security flaws that manifest at the hardware–software boundary with greater speed and scalability compared to simulation and FPGA-based approaches. In the future, we plan to extend this framework by integrating artificial intelligence (AI) to further enhance automation and overall detection efficiency.

VI. Acknowledgement

The authors would like to acknowledge funding support through Semiconductor Research Corporation (Grant/Award No. 2022-HW-3124 / AWD13345).

References

[1] K. Z. Azar, M. M. Hossain, A. Vafaei, H. A. Shaikh, N. N. Mondol, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Fuzz, penetration, and AI testing for SoC security verification: Challenges and solutions," Cryptology ePrint Archive, Paper 2022/394, 2022. [Online]. Available: https://eprint.iacr.org/2022/394

[2] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Soc security verification using property checking," in *2019 IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–10.

[3] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, 2021.

[4] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999.

[5] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable soc trust verification using integrated theorem proving and model checking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016, pp. 124–129.

[6] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1691–1696.

[7] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner, "Gate-level information flow tracking for security lattices," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, pp. 1–25, 2014.

[8] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.

[9] M. Zalewski, "American Fuzzy Lop," https://lcamtuf.coredump.cx/afl/, 2014.

[10] K. Serebryany, "libfuzzer–a library for coverage-guided fuzz testing," *LLVM project*, p. 34, 2015.

[11] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," *URl: https://github.com/google/honggfuzz (visited on 06/21/2025)*, 2017.

[12] 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 *Proceedings of the 31st USENIX Security Symposium*. USENIX Association, 2022, pp. 3219–3236.

[13] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023.

[14] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 529–534.

[15] 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)*. IEEE, 2021, pp. 1286–1303.

[16] 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)*. IEEE, 2024, pp. 1–6.

[17] L. Wu, M. Rostami, H. Li, J. Rajendran, and A.-R. Sadeghi, "{GenHuzz}: An efficient generative hardware fuzzer," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1787–1805.

[18] S. Tarek, D. Saha, S. K. Saha, M. Tehranipoor, and F. Farahmandi, "Socurellm: An llm-driven approach for large-scale system-on-chip security verification and policy generation," in *2025 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2025, pp. 335–345.

[19] L. Wu, M. Rostami, H. Li, and A.-R. Sadeghi, "Hfl: Hardware fuzzing loop with reinforcement learning," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[20] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. J. Rajendran, "PSOFuzz: Fuzzing processors with particle swarm optimization," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[21] V. Gohil, R. Kande, C. Chen, A.-R. Sadeghi, and J. Rajendran, "MABFuzz: Multi-armed bandit algorithms for fuzzing processors," in *arXiv preprint arXiv:2311.14594*, 2023.

[22] 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)*. IEEE, 2023, pp. 1–6.

[23] N. F. Dipu, M. M. Hossain, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "FormalFuzzer: Formal verification assisted fuzz testing for SoC vulnerability detection," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 355–361.

[24] M. M. Hossain, N. F. Dipu, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "TaintFuzzer: SoC security verification using taint inference-enabled fuzzing," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2023, pp. 1–6.

[25] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-assisted processor fuzzing," in *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023, pp. 1361–1378.

[26] S. K. Muduli, G. Takhar, and P. Subramanyan, "HyperFuzzing for SoC security validation," in *2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2020, pp. 1–9.

[27] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *Proceedings of the 31st USENIX Security Symposium*. USENIX Association, 2022, pp. 3237–3254.

[28] 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)*. IEEE, 2018, pp. 1–8.

[29] "Zebu server 5," https://www.synopsys.com/verification/emulation/zebu-server.html, accessed: 2025-06-30.

[30] "Palladium emulation," https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html, accessed: 2025-06-30.

[31] "Veloce cs system," https://eda.sw.siemens.com/en-US/ic/veloce/, accessed: 2025-06-30.

[32] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[33] S. Nolting, "The neorv32 risc-v processor," https://github.com/stnolting/neorv32, accessed: 2025-06-30.

[34] H. Al Shaikh, S. Saha, S. K. Saha, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "Rethinking system-on-chip verification for secure cross-layer interactions," *IEEE Design & Test*, 2025.

[35] MITRE, "Common weakness enumeration (cwe)," https://cwe.mitre.org/, accessed: 2025-06-30.