SoCFuzzer: SoC Vulnerability Detection using Cost Function enabled Fuzz Testing

Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman,
Farimah Farahmandi, and Mark Tehranipoor
Electrical and Computer Engineering, University of Florida, Gainesville, Florida 32611, USA
{hossainm, arash.vafaei, k.zamiriazar}@ufl.edu, {fahimrahman, farimah, tehranipoor}@ece.ufl.edu

***Abstract*—The modern System-on-Chips (SoCs), with numerous complex and heterogeneous intellectual properties (IPs), and the inclusion of highly-sensitive assets, become the target of malicious attacks. However, security verification of these SoCs remains behind compared to the advances in functional verification, mostly because it is difficult to formally define the accurate threat model(s). Few recent studies have investigated the possibility of engaging fuzz testing for hardware-oriented vulnerability detection. However, they suffer from several limitations, i.e., lack of cross-layer co-verification, the need for expert knowledge, and the inability to capture detailed hardware interactions. In this paper, we propose *SoCFuzzer*, an automated SoC verification assisted by fuzz testing for detecting SoC security vulnerabilities. Unlike the previous HW-oriented fuzz testing studies, which mostly rely on traditional (code) coverage-based metrics, in *SoCFuzzer*, we develop (i) generic evaluation metrics for fuzzing the hardware domain, and (ii) security-oriented cost function. This relieves designers of making correlations between coverage metrics, test data, and possible vulnerabilities. The *SoCFuzzer* cost functions are defined high level, allowing us to follow the gray-box model, which requires less detailed and interactive information from the design-under-test. Our experiments on an open-source RISC-V based SoC show the efficiency of these metrics and cost functions on fuzzing for generating cornerstone inputs to trigger the vulnerability conditions with faster convergence.**

***Index Terms*—SoC Security Verification, Evolutionary Testing, Fuzzing, Cost Function**

I. Introduction

Modern System-on-Chips (SoCs) are coming to market every year under tight time-to-market constraints, shrinking the window for full-scale verification. Several IPs from across the globe are integrated into a single SoC and must work together effectively to meet the requirements. With this huge increase in communication and synchronization, the need for security verification is also becoming paramount. Moreover, some of these third-party IPs (3PIPs) are not trusted and may contain malicious functionality. So, integrating them to the SoC may introduce newer vulnerabilities even to the trusted IPs [1].

During inter-IP interactions, assets such as encryption keys, device configurations, protected memory regions, and many more can be compromised. Any threat to confidentiality, integrity, or availability should be addressed in every design abstraction. Many studies have focused on security verification of large SoCs, incorporating methods such as pseudo-randomized testing, formal methods, and concolic execution [2], [3]. However, as SoCs are getting larger, these methods become less effective. They usually require full (white-box) access to the design and logical proof of security properties requires expert knowledge. Also, they suffer from scalability issues, making them less effective for larger designs.

One approach that has recently attracted attention and can be applied both at post-silicon and pre-silicon stages is fuzz testing [4]–[10]. Originally, fuzz testing was used extensively by software (SW) designers for identifying SW vulnerabilities and weaknesses by pushing the SW to unexpected inputs. From generation-based fuzzing for SW (a.k.a. black-box fuzzing) [11], [12] to coverage-guided mutation-based fuzzing (a.k.a. gray-box fuzzing) [13], [14], they engage dictionary-based sequence update (feedback-based mutation) for generating the test patterns. For instance, SiliFuzz [15] is the latest fuzz testing by Google on CPU defects that shows the high effectiveness of fuzzer for corner case bugs. All HW-oriented fuzz testing frameworks [4]–[9] have tried to simply re-use these SW fuzzing engines (e.g., AFL [14]) for exploiting vulnerabilities at HW-level. However, this re-usage suffers from important shortcomings: (1) It focuses on increasing the coverage (SW-based) metrics, which is not directly correlated to a significant portion of IP and SoC-based vulnerabilities; (2) Usually, they focus on specific categories of potentia bugs, such as side-channel or speculative execution bugs; and (3) They are not following any specific security property formulation to purposefully update the testing procedure (no security-driven mutation and feedback). These shortcomings affect the automation of vulnerability detection in the HW domain, and the need for expert contribution becomes inevitable.

To address these shortcomings, in this paper, we introduce *SoCFuzzer* that extends the popular fuzzing engine, AFL [14] for HW-oriented SoC security verification. In *SoCFuzzer*, we modify AFL so that it can be guided by a generic cost function that directs the fuzzing mutation towards unexplored corners of the design (w.r.t. security properties). Evaluation of the cost function requires partial feedback from the design that is done using an emulation-based integrated logic analyzer. *SoCFuzzer* is examined via real HW vulnerabilities from different Hackathons on a large enough RISC-V SoC. Our main contributions to this work are as follows:

(1) Building a fuzzing engine for SoC vulnerability detection.
(2) Metric formalization for evaluating fuzzing performance.
(3) Developing a generic cost function with feedback based on the vulnerability characteristic and SoC design specification for guiding fuzzing mutation engine.
(4) Evaluating vulnerability detection capability on a real SoC.

978-3-9819263-7-8/DATE23/© 2023 EDAA

II. Background and Prior Art

The term *security asset* refers to any design article, e.g., ports, memory, or signals, that are of value from a security standpoint. Security assets are not only limited to the primary ones identified by the designers. Communication protocols and configuration bits are also subliminal assets that are harder to pinpoint and could compromise the system's security. The assets may be leaked/altered either mistakenly due to a design bug or deliberately by inserted malicious logic (e.g., HW Trojan) [16]. Identifying the source of the leakage/alternation is the crucial role of security verification, which can be done by defining and evaluating security policies.

Different mechanisms have been used to realize these policies for verifying system security, such as formal methods [2], [17], information flow tracking (IFT) [18], [19], and symbolic or concolic testing (simulation) [3], [20]. In formal verification, the security policies are translated to assertions (e.g., SystemVerilog assertions), and the assertions will be evaluated by the formal tool (e.g., Cadence JasperGold). However, the definition of these assertions requires expert knowledge and mostly works as a white-box testing [2], [17]. In IFT-based verification, label-based propagation (tainting [21]) is used for building the policies, and the propagation of these labels determines the occurrence of vulnerabilities. Due to tainting, IFT suffers from excessive and unwanted propagation, resulting in poor accuracy and limited scalability. Additionally, for taint propagation, it follows white-box testing assumptions [18], [19]. Similarly, in symbolic and concolic (symbolic + concrete) testing, test cases are generated to effectively cover targeted timing paths. However, since it relies on the propagation of symbolic patterns, it suffers from scalability issues [3], [20].

Fuzzing or fuzz testing can outperform the above-mentioned mechanisms as (i) it supports both black-box and gray-box testing and (ii) it controls the propagation workload by defining feedback. Fuzzing originally (in SW) is the process of generating unexpected (related to corner cases) test patterns to trigger bugs and vulnerabilities. As summarized in Fig. 1, it accomplishes the testing based on some initial seed, the mutation engine, and feedback. With high efficacy in SW, few recent studies investigated applying SW fuzzing to HW (RTL) [4]–[9]. The SW fuzzing can be applied to the HW in two variants: (1) Fuzzing the HW as SW: In this variant, the HW is first translated to its SW representation, and then any SW fuzzer can be invoked [5], [8]. (2) Directly fuzzing the HW: In this variant, the fuzzer (in the form of a HW simulator) will be applied directly to the HW [4], [6], [7], [9].

These approaches are promising but have several important shortcomings: (i) Even for direct fuzzers on HW, there exists a total dependence on SW-based metrics for deciding on security policies. Almost all HW-oriented fuzz testing studies rely on coverage (code-based) metrics, and there exist no clear mapping between (SW) fuzzer assertions and HW-oriented bugs. (ii) There exists no clear relation between HW vulnerabilities and coverage metrics (illegal access to a security asset). (iii) SW-related metrics lead the verification towards a specific category of possible bugs (e.g., speculative execution or side-channel-related). (iv) None of the fuzzers on HW establishes a self-evolutionary cost-function formulation to provide a better sense of proximity for the fuzzer so that the fuzzing process tunes itself to bring the execution closer to vulnerabilities.

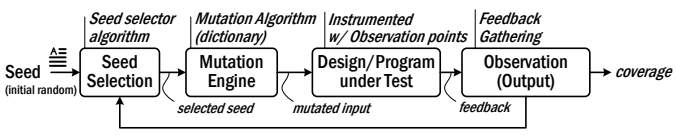

Fig. 1: Fuzz Testing Overview.

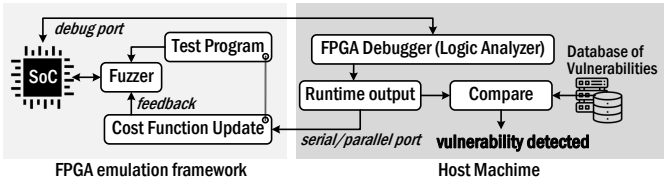

Fig. 2: A High-Level Diagram of *SoCFuzzer* Framework.

III. Proposed Framework: SoCFuzzer

A. Overview

SoCFuzzer is a dynamic verification framework based on cost function enabled fuzzing that detects vulnerabilities in SoCs autonomously. A high-level overview of *SoCFuzzer* framework is shown in Fig. 2. *SoCFuzzer* relies on FPGA-based emulation enabled with real-time internal HW signal monitoring. It accomplishes instrumentation on the SoC-under-test based on the set of security policies (cost functions). Instrumentation in *SoCFuzzer* consists of adding observation points to the SoC making them accessible in real-time in the emulation. The proposed fuzzer runs program(s) (with fuzzed inputs) on the CPU of the SoC w.r.t. the vulnerability of interest. Following the gray-box model, we assume the verification engineer has limited knowledge about the behavior of the SoC (no golden model). We developed metrics for evaluating the performance of the fuzzing engine in real time. In *SoCFuzzer*, the feedback is based on the value of the cost function (w.r.t. the security policies), and the most-fitted mutation technique will be selected based on the feedback allowing to have a faster convergence for triggering the vulnerabilities.

B. SoCFuzzer Evaluation Metrics

The ultimate goal of fuzzing is to generate smarter-than-random test inputs (based on the feedback), e.g., bit-flipping, permutation, etc., to obtain better coverage with faster convergence. Hence, in *SoCFuzzer*, to attain smart inputs w.r.t. the security properties, we develop 4 distinct metrics:

① Metric 1 - Randomness (M_1): This metric estimates the randomness in a particular set of fuzz-generated inputs. Considering that the complete randomness is not satisfactory for fuzzing[1], we expect the fuzzing tool to generate inputs far away from the average hamming distance (HD) of 50% (ideal HD in complete randomness) while generating unique inputs

[1]Fuzzing with no feedback generates complete random test inputs, which provides lower coverage compared to feedback-enabled fuzzing.

for each execution. The average HD must not be zero to avoid the case where the fuzzing tool generates duplicated inputs. Eq. 1 is the mathematical equation used in *SoCFuzzer* to estimate the randomness of the inputs. The lower f_r indicates less randomness in the generated fuzzed inputs which are not generated arbitrarily but rather crafted intelligently, thus making them more satisfactory for fuzzing.

$$f_r = \frac{2}{r(r-1)} \sum_{j=1}^{r-1} \sum_{k=j+1}^{r} \frac{h(i_j, i_k)}{l_i}, f_r \neq 0 \quad (1)$$

In Eq. 1, r: total number of fuzzed inputs, i_j: fuzzed input value for j^{th} run, i_k: fuzzed input value for k^{th} run, l_i: input length (bits), and $h(i_j, i_k)$: HD of j^{th} and k^{th} fuzzed inputs.

② Metric 2 - Output Activity (M_2): This metric evaluates the impact of the inputs on the output to see whether the circuit functions or sticks at a dead/idle state. If the program gets stuck (stall in state transition) even with newly fuzzed patterns, this will not instigate the program to trigger the vulnerability. For instance, the ciphertext must be different for various plaintext in an AES module. This metric also allows enabling/disabling IPs in the SoC via fuzzer. For example, assume that a `done` signal as the output of $\texttt{IP}_1$ is used as the input for $\texttt{IP}_2$. For verification, the output from $\texttt{IP}_1$ must be always valid so that the fuzzer can target the verification of $\texttt{IP}_2$. Eq. 2 is the mathematical expression developed to measure the output activity. In Eq. 2, σ is a constant parameter to be decided from the specification[2]. The increment in f_a indicates increasing changes in the output (higher HD).

$$f_a = \frac{\sigma}{\sum_{j=1}^{z} l_z} \sum_{k=1}^{z} (h(o_{r-1,k}, o_{r,k})) \quad (2)$$

In Eq. 2, $\sigma \in \{0, 1\}$, z: total number of output signals, l_z: length of z^{th} output signal in bits, and $h(o_{r-1,k}, o_{r,k})$: HD of output k^{th} values between r^{th} and $(r-1)^{th}$ iteration.

③ Metric 3 - Input Coverage (M_3): This measures the traversed input space of the SoC-under-test. With more iterations, *SoCFuzzer* generates more unique inputs to fuzz the program. Therefore, the input space coverage increases over time. Eq. 3 shows how this metric is calculated in *SoCFuzzer* for more efficient traversing input space for fuzzing. Since there might be some bits that should have a fixed value for fuzzing, e.g., *enable/reset* signals, these bits are excluded in this metric.

$$f_c = \frac{u_i}{2^{N-d}} \quad (3)$$

In Eq. 3, u_i: total unique inputs (traversed), d: number of fixed bits (exclusion), N: total number of input bits of SoC.

The increasing value of f_c indicates the fuzzer covers more input space and hence getting closer to trigger of the vulnerability (corner cases with high possibility of bugs). This metric also states the overall coverage of the input space compared to the brute-force verification (worst-case). When the fuzzing tool tries many fuzzed inputs and is unable to detect any vulnerability, we can make conclusions about the confidence in absence of vulnerabilities in the SoC as: $C_0 = f_c \times 100\%$.

④ Metric 4 - Target Output Behavior (M_4): This metric observes the output to distinguish the malicious (unexpected) behavior. Observing a particular subset of SoC output signals, the framework decides whether the fuzzer has reached any vulnerability conditions. This metric gives the percentile of how many output signals achieve the corresponding asset value or behavior that collectively indicates the triggering of a vulnerability, as calculated by Eq. 4. For example, reading the AES key from the bus as ciphertext suggests information leakage, where the AES key is the expected value. For some cases, such as accessing protective memory segments, the expected output behavior is the raise of an exception.

$$f_o = \frac{1}{m} \sum_{k=1}^{m} (o_{r,k} == o_{t,k}) \quad (4)$$

In Eq. 4, *m*: number of target output signals/behavior, $o_{t,k}$: expected value/behavior of k^{th} target output signal, and $o_{r,k}$: runtime observed value or behavior of k_{th} target output signal.

C. SoCFuzzer Cost Function

The cost function is developed based on the proposed fuzzing evaluation metrics described above. As these metrics quantitatively measure the fuzzing performance at run-time, improving these metrics in fuzzing indicates the faster convergence of the cost function to hit the vulnerability (global minima). Therefore, the cost function is developed based on these normalized metrics. Among all metrics, decreasing of f_r is expected for efficient fuzzing as opposed to f_a, f_c, and f_o (whose increasing is expected). In other words, increasing $(1 - f_r)$, f_a, f_c, and f_o are the satisfactory sign of efficient fuzzing. Averaging these and subtracting from 1 gives a mathematical expression of the normalized cost function as mentioned in Eq. 5, in which $n \in \{3, 4\}$ is the number of metrics considered in F_c[3].

$$F_c = 1 - \frac{(1 - f_r) + f_a + f_c + f_o}{n} = \frac{n-1}{n} - \frac{1}{n}(f_a + f_c + f_o - f_r) \quad (5)$$

f_a, f_c, and f_o are calculated after each execution while fuzzing. While f_r is calculated when the feedback is generated. All of these updated metric values are used in computing F_c. The decreasing value of the cost function indicates that the fuzzing tool is approaching the vulnerability triggering condition gradually. Finally, when F_c reaches the global minima, the vulnerability is (based on M_4) triggered and it can be seen in the final test patterns produced by the fuzzer.

D. SoCFuzzer Feedback

To understand and track the performance of *SoCFuzzer* and develop the suited feedback, we define a parameter named *cost function improvement rate (CFIR)* as Eq. 6, where F_{ci} is the cost function values for the r_i^{th} execution/iteration.

$$CFIR = -\frac{\delta F_c}{\delta r} = -\frac{F_{c2} - F_{c1}}{r_2 - r_1} \quad (6)$$

The positive value of *CFIR*, i.e., decrement in F_c, is a sign the fuzzer is performing better by developing more smart and cornerstone inputs (approaching to global minima

[2] σ becomes zero if the output does not change, otherwise one.

[3] $n = 3$ when M_2 is omitted due to σ is zero (see Eq. 2), otherwise $n = 4$.

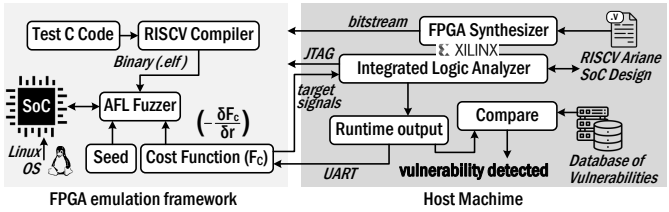

Fig. 3: Implementation of *SoCFuzzer* Framework.

of F_c). The negative value of *CFIR* is a sign the fuzzer is performing against the objective, thus *SoCFuzzer* switches the mutation algorithm. By using this metric, the fuzzing framework receives feedback from the cost function and triggers the vulnerability in significantly faster (keeping *CFIR* always high positive). *CFIR* can be calculated after each or a certain number of iteration(s), which is defined as the *frequency of feedback generation* (*FREQ*$_{fb}$). After each *FREQ*$_{fb}$ execution, the fuzzer may change the mutation strategy based on the positive/negative value of *CFIR*.

IV. Experimental Setup and Results

To evaluate the efficiency and performance of *SoCFuzzer*, we implemented the whole framework based on open-source RISC-V-based Ariane SoC [23]. Fig. 3 shows the details of implementation and how different components are connected in *SoCFuzzer*. To enable internal HW debugging, we integrated the Xilinx Integrated Logic Analyzer (ILA) core, and the emulation has been established on Genesys 2 Kintex-7 FPGA Development Board [24]. We enabled real-time monitoring via JTAG debug port, and the fuzzing engine, which is based on AFL [14], has been prepared on a Linux kernel directly mounted on the SoC (through SD card). The AFL has been modified (AFL v2.57b) for multiple reasons: (i) calculating the metrics M_1 and M_3; (ii) receiving metrics M_2 and M_4 from the host machine through UART; (iii) calculating F_c and *CFIR*; (iv) use feedback to change the mutation techniques of AFL and generate HW-oriented mutated inputs. To fuzz the SoC and detect the vulnerabilities, a set of high-level C code executables has been written to drive particular components of the SoC. These codes are compiled using the RISC-V compiler and the executable binary (*.elf*) is provided to the fuzzer on SoC with the required initial seeds. *SoCFuzzer* runs these codes via fuzzer, and by gathering the output activity through ILA, it calculates the metrics and feedback for guiding the fuzzer.

A. Vulnerabilities in Ariane SoC

We targeted a set of vulnerabilities in the Ariane SoC, and we restated the cost functions (w.r.t. the parameters) per vulnerability. The investigated bugs are a set of reported bugs in the MITRE CWE database [22]. For proof of concept, this study focuses on 5 different vulnerabilities listed in Table I. SV1 and SV2 are based on malicious Trojans introduced into the AES core of the Ariane SoC. These vulnerabilities are emerging due to the integration of 3PIP acquired from an untrusted vendor into the SoC. As per SV1, the AES key would be leaked, which is the security-critical asset of the AES IP. As per SV2, a timing violation in the AES crypto module may result in a denial of service for a time-critical application of the SoC. SV3 indicates that the instruction decoder (dec) in RISC-V processor does not ignore the *imm* and *rs1* fields of *FENCE.I* instruction and thus violating the RISC-V specification. An illegal rejection of execution of *FENCE.I* may lead to cache coherence issues. Both SV4 and SV5 are privilege-level escalation vulnerabilities. In accordance with the RISC-V ISA, there are three privilege modes, machine (M), supervisor (S) and user (U), for proper program execution with authorized permissions. Illegal execution of instruction from lower privileges (SV4) and unauthorized read/write operations in the control and status register (SV5) may allow untrusted execution of third-party programs and thus critically risk the continuous operations of the SoC. The attacker may escalate the privilege level by modifying the *mstatus_reg* and thus compromising the security assets.

B. Cost Function Development

The cost function for guiding the vulnerability detection framework is presented in Eq. 5. The parameters associated with the cost function of each vulnerability (based on the specifications) are listed in Table II. For instance, for SV1, a C program is written to drive the AES cryptography core in the SoC with $128-bit$ AES key and plaintext. According to the specification of AES IP, output as ciphertext (*C*) always changes with any change in the plaintext (*P*). Therefore, the fuzzer expects a change in the *C* even for a single bit of variation in *P*, which defines σ to be 1 for M_2. The effective length of the input is 128 bits $(N-d)$ as mentioned in M_3 because there is no bit to be fixed in the AES inputs. In this vulnerability verification, the output signal is the *C* $(m=1)$ which is monitored for any key leakage. The cost function is deduced to Eq. 7 with the above assigned parameters.

$$F_{c,SV1} = \frac{n-1}{n} - \frac{1}{n}[\frac{\sigma}{\sum_{j=1}^{z} l_z} \sum_{k=1}^{z} (h(C_{r-1,k}, C_{r,k})) + \frac{u_i}{2^{N-d}} + \frac{1}{m} \sum_{k=1}^{m} (C_{r,k} == AES_{key}) - \frac{2}{r(r-1)} \sum_{j=1}^{r-1} \sum_{k=j+1}^{r} \frac{h(P_j, P_k)}{l_P}] \quad (7)$$

As SV2 targets AES as well, a similar cost function will be defined. In the case of SV3 and SV4, the input is 32-bits of RISC-V instruction ((N-d) $\rightarrow$ 32) and the output behavior is either successful execution or an exception if the mutated instruction cannot be executed. As the CSR specifier is 12 bits in the 32-bits of CSR RISC-V instruction [25], the effective length is 12 for SV5. Unlike SV1 and SV2, the output is not contiguous in SV3, SV4, and SV5. So, the σ equals 0 for these vulnerabilities. Thus the parameters are derived for developing cost function for gray-box fuzzing based on the high-level characteristics of the target module in the SoC, not the in-depth source-code level information.

C. Results and Analysis

This section covers how the definition of the cost function, *CFIR*, and *FREQ*$_{fb}$ in *SoCFuzzer* affects the performance of the fuzzer for identifying the vulnerabilities.

TABLE I: List of Targeted Vulnerabilities based on CWE MITRE database [22] Inserted in Ariane SoC [23].

Index	Vulnerability	Location	Triggering Condition	Output Behaviour	Reference
SV1	Leaking the AES secret key through the common bus in the SoC	AES IP	Specific plaintext	AES Key leaks to PO	CVE-2018-8922
SV2	A Trojan injects a delay in the AES IP in cipher conversion	AES IP	Specific plaintext	Ciphertext not resulted in time	AES-T500
SV3	Incorrect implementation of logic to detect the FENCE.I instruction	CPU (dec)	$imm \neq 0$ & $rs1 \neq 0$	Illegal instr exception raised	CWE-440
SV4	Execute machine-level instructions from user mode	CPU (dec)	Execute "mret" instruction	No exception exhibited	CWE-1242
SV5	Access to CSRs from lower privilege level	Register file	*mstatus_reg* r/w from user space	No exception exhibited	CWE-1262

TABLE II: Parameters in Cost Function Development for Each Vulnerability.

Index	σ	Input		Output		Effective
		Data	Length	Signal	Length	Length (*N-d*)
SV1	1	Plaintext	128 bits	Ciphertext	128 bits	128 bits
SV2	1	Plaintext	128 bits	Control Register	32 bits	128 bits
SV3	0	Instruction	32 bits	Exception Raised	N/A	32 bits
SV4	0	Instruction	32 bits	No Exception	N/A	32 bits
SV5	0	CSR Ins.	32 bits	No Exception	N/A	12 bits

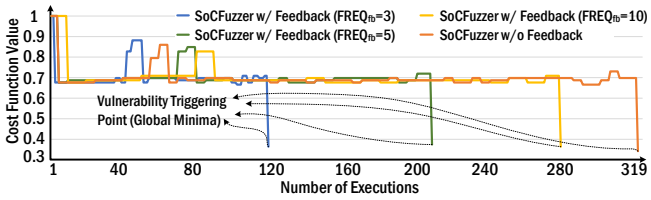

Fig. 4: Cost Function and Feedback Analysis in Detecting Vulnerability (SV3).

In *SoCFuzzer*, the impact of feedback on fuzzer and its mutation is heavily dependent on *CFIR*. In experiments, we observe that *SoCFuzzer* enabled with this feedback outperforms the conventional fuzzing framework without this feedback. In Fig. 4, we plot the run-time cost function in each execution of the test program for various $FREQ_{fb}$ while targeting SV3. Starting from the initial high value of F_c, based on the changes in F_c, the mutation is updated by *SoCFuzzer* per each $FREQ_{fb}$ number of iterations. In case F_c is increasing (fuzzing efficiency is degrading), *CFIR* becomes negative enforcing *SoCFuzzer* to change the mutation technique. As long as the F_c is decreasing, no change is required for the mutation technique. Although we witnessed more fluctuation of F_c at the early iterations of fuzzing, it becomes more stable while it finds the most appropriate mutation technique, leading to a fast convergence to the global minima. For any $FREQ_{fb}$ value, *SoCFuzzer* with feedback converges to the global minima significantly sooner than the system without the feedback. The run-time cost function for other vulnerabilities for an optimum $FREQ_{fb}$ is shown in Fig. 5, which confirms the efficacy of feedback on the performance of fuzzing per vulnerability.

As shown in Figs. 4 and 5, in *SoCFuzzer*, $FREQ_{fb}$ affects the convergence rate. Fig. 6 shows the impact of various values of $FREQ_{fb}$ on the performance of *SoCFuzzer* (number of executions needed for exploiting the vulnerabilities). When $FREQ_{fb}$ is small (e.g., 2 or 3), it indicates that the mutation technique must be updated rapidly (per 2-3 executions). This results in not evaluating a sufficient number of samples, thus taking a long time to reach the global minima. Contrarily, when

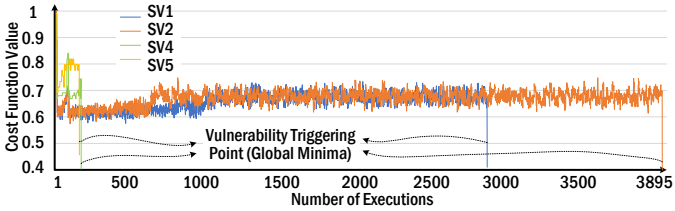

Fig. 5: Cost Function Analysis in Detecting SV1, SV2, SV4, and SV5.

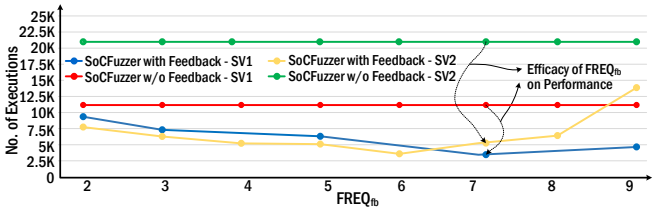

Fig. 6: Performance of *SoCFuzzer* with Feedback for Variation in $FREQ_{fb}$.

$FREQ_{fb}$ is large (e.g., 8 or 9), the fuzzer spends more (unnecessary) time on the same (inefficient) mutation technique before considering the new feedback. It also may slow down the fuzzing to reach the global minima. So, $FREQ_{fb}$ requires to be selected meticulously to provide the highest efficiency for *SoCFuzzer*. As shown in Fig. 6, we swept $FREQ_{fb}$ for vulnerabilities SV1 and SV2 and show the required executions to trigger them. For these vulnerabilities, the best performance is achieved when $FREQ_{fb}$ belongs to the range of 5→7, in which our *SoCFuzzer* can speed up the performance up to 2.83x (for SV1) and 6.1x (for SV2).

Also, the initial seed affects the fuzzing in triggering the vulnerability in terms of convergence rate [26]. When the HD between a seed and the fuzzed input for which ultimately the vulnerability got triggered is higher (i.e., highly dissimilar), we define the seed as *poor quality*. In some cases, detecting a vulnerability may take a significant time with a poor quality of seed. However, *SoCFuzzer* with cost function enabled feedback can converge to the global minima significantly faster even with poor quality seeds. Fig. 7 shows the required executions for triggering SV3 and SV4 for various quality seeds. In this case, we selected the seeds randomly which have a wide range of HD compared to the vulnerability triggering inputs. As shown in Fig. 7, when we have no feedback and the HD is high, more executions needed for exploiting the vulnerability compared to that of *SoCFuzzer* with feedback.

A summary of the verification results is shown in Table III with optimum $FREQ_{fb}$ for all vulnerabilities. Due to the randomness in the fuzzer, two identical runs may take different numbers of executions to trigger the vulnerability. So, we ran each experiment three times and presented the average results.

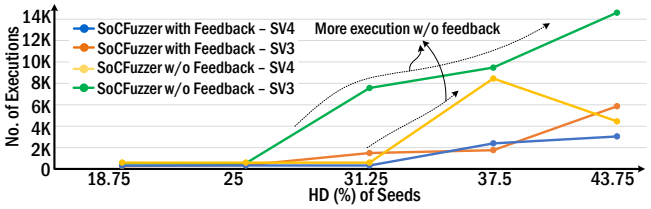

Fig. 7: Performance of *SoCFuzzer* for Various Quality Seeds.

TABLE III: Summary Results of Vulnerability Detection by *SoCFuzzer*.

Index	HD(seed, VTI)	$FREQ_{fb}$	No. of Executions		Speed up
			Fuzzing w/o CF_FB	SoCFuzzer	
SV1	62.5%	7	10968	2862	73.91%
SV2	62.5%	6	21319	2999	85.93%
SV3	25%	6	361	180	50.14%
SV4	43.75%	5	415	162	60.96%
SV5	66.67%	6	968	275	71.59%

HD(seed, VTI): HD of Seed and Vulnerability Triggering Input
CF_FB: Cost Function enabled Feedback $FREQ_{fb}$: Optimum $FREQ_{fb}$

Our proposed framework was able to detect all vulnerabilities in a reasonable time for a variety quality of seeds (higher quality for SV3 and SV4 and poor quality for SV1, SV2, and SV5). *SoCFuzzer* enabled with our proposed cost function enabled feedback saved at least 50% of verification time compared to the system without our proposed feedback (conventional fuzzing on HW with no cost function).

D. Comparison with Prior Art

The power of *SoCFuzzer* lies in its independence from golden models, use of an analog cost function, enabling cost function based feedback, and its generic metrics that are easy to incorporate and easy to evaluate. None of the state-of-the-art fuzzing engines on HW supports these features cumulatively. Table IV provides the key differences between our proposed *SoCFuzzer* vs. the state-of-the-art fuzzing techniques on HW. Our emulation-based framework enabled with cost function feedback improves the efficiency and performance of fuzzing allowing us to apply such a testing mechanism even on large and complex SoCs (with almost no scalability issues).

V. Conclusion

This study introduced *SoCFuzzer*, a fuzzing framework for SoC security verification in an autonomous fashion. *SoCFuzzer* integrates our proposed cost function enabled feedback system leveraging a real-time debugging platform. The feedback is developed based on HW-oriented generic evaluation metrics and cost function for the gray-box fuzzing model and guides the mutation engine of the fuzzer for faster convergence to the vulnerability triggering point. The experimental results proved the capability and efficiency of *SoCFuzzer* in terms of both vulnerability detection and verification time.

TABLE IV: Comparison of *SoCFuzzer* with Some HW Fuzzing Frameworks.

Method	Fuzzer Engine	Framework Target		Feedback to Mutation Engine	Golden Model
RFUZZ [4]	HW Fuzzer	FPGA	IP Designs	MUX coverage	Yes
DifuzzRTL [27]	HW Fuzzer	FPGA	CPU Design	Control-register Code coverage	Yes
Hyper fuzzing [5]	SW Fuzzer	SW Sim	SoC Design	NoC, Instruction and Bitflip Monitors	Yes
TheHuzz [9]	HW Fuzzer	HDL Sim	CPU Design	Statement, toggle, branch expression, condition, FSM	Yes
SoCFuzzer	Modified HW Fuzzer*	FPGA	SoC Design	Cost Function based (vulnerability-oriented)	No

FPGA: FPGA Emulation Sim: Simulation *: Based on Cost Function and Metrics

References

[1] K. Z. Azar *et al.*, "Fuzz, penetration, and ai testing for soc security verification: Challenges and solutions," *Cryptology ePrint Archive*, 2022.

[2] N. Farzana *et al.*, "Soc security verification using property checking," in *IEEE International Test Conference (ITC)*, 2019, pp. 1–10.

[3] X. Meng *et al.*, "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] K. Laeufer *et al.*, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *International Conference on CAD (ICCAD)*, 2018, pp. 1–8.

[5] S. K. Muduli *et al.*, "Hyperfuzzing for soc security validation," in *International Conference on CAD (ICCAD)*, 2020, pp. 1–9.

[6] S. Canakci *et al.*, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *DAC*, 2021, pp. 529–534.

[7] J. Hur *et al.*, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1286–1303.

[8] T. Trippel *et al.*, "Fuzzing hardware like software," in *USENIX Security Symposium*, 2022, pp. 3237–3254.

[9] R. Kande *et al.*, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *USENIX Security Symposium*, 2022, pp. 3219–3236.

[10] H. Al-Shaikh *et al.*, "Sharpen: Soc security verification by hardware penetration test," in *Asian and South Pacific Conference on DAC (ASP-DAC)*, 2023, pp. 1–6.

[11] X. Yang *et al.*, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN PLDI*, 2011, pp. 283–294.

[12] J. Shen *et al.*, "Native mode functional test generation for processors with applications to self test and design validation," in *International Test Conference (ITC)*, 1998, pp. 990–999.

[13] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *IEEE Cybersecurity Development (SecDev)*, 2016, pp. 157–157.

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

[15] K. Serebryany *et al.*, "Silifuzz: Fuzzing cpus by proxy," *arXiv preprint arXiv:2110.11519*, 2021.

[16] N. Farzana *et al.*, "Saif: Automated asset identification for security verification at the register transfer level," in *VLSI Test Symposium (VTS)*, 2021, pp. 1–7.

[17] X. Guo *et al.*, "Scalable soc trust verification using integrated theorem proving and model checking," in *HOST*, 2016, pp. 124–129.

[18] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 1691–1696.

[19] W. Hu *et al.*, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.

[20] R. Zhang *et al.*, "End-to-end automated exploit generation for validating the security of processor designs," in *MICRO*, 2018, pp. 815–827.

[21] M. M. Hossain *et al.*, "Boft: Exploitable buffer overflow detection by information flow tracking," in *DATE*, 2021, pp. 1126–1129.

[22] MITRE, "HW CWEs," https://cwe.mitre.org/data/definitions/1194.html.

[23] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[24] "Genesys 2 fgpga development board." [Online]. Available: https://digilent.com/reference/programmable-logic/genesys-2/start

[25] A. Waterman *et al.*, "The risc-v instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, 2014.

[26] A. Herrera *et al.*, "Seed selection for successful fuzzing," in *International Symposium on Software Testing and Analysis*, 2021, pp. 230–243.

[27] S. Nilizadeh *et al.*, "Diffuzz: differential fuzzing for side-channel analysis," in *Software Engineering Conference (ICSE)*, 2019, pp. 176–187.