InterConFuzz: A Fuzzing-based Comprehensive NoC Verification Framework

Samit Shahnawaz Miftah
Dept of Electrical and Computer Engineering
University of Texas at Dallas
Dallas, Texas, USA
samit.miftah@utdallas.edu

Hyunmin Kim
Technology Innovation Institute
Abu Dhabi, UAE
hyunmin.kim@tii.ae

Kanad Basu
Dept of Electrical and Computer Engineering
University of Texas at Dallas
Dallas, Texas, USA
kanad.basu@utdallas.edu

***Abstract*—Security verification of Network-on-Chip (NoC) systems is essential due to their intricate and high-concurrency structures. Traditional methods often fail to cover all scenarios or scale effectively, leading to prolonged verification and overlooked vulnerabilities. Our proposed solution, *InterConFuzz*, a hybrid hardware fuzzing technique, uses symbolic execution for extensive coverage. Developed on Universal Verification Methodology (UVM), *InterConFuzz* discovered five security flaws in the NoC architecture of the OpenTitan SoC—surpassing existing techniques by three—while reducing memory and computational needs by 24.4% and 29.5%, respectively. Furthermore, *InterConFuzz* furnished comparable functional coverage compared to existing NoC fuzzing approaches, proving its efficiency and robustness.**

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

I. Introduction

The increasing complexity of modern System-on-Chip (SoC) designs stems from the integration of a growing number of Intellectual Property (IP) blocks and sophisticated communication protocols that allow these components to interact efficiently. This complexity is managed by Network-on-Chip (NoC) systems, which are responsible for coordinating data flow, resource sharing, and communication across the SoC. While NoCs enhance performance and modularity, they also introduce potential security vulnerabilities due to the intricate interactions between multiple processing units. Flaws or weaknesses in NoC designs can expose sensitive data or lead to malfunctioning components, resulting in critical system failures. Such vulnerabilities and functional errors can have far-reaching consequences, including delays in product development and significant financial losses due to expensive silicon respins and redesigns, with over 70% of respins attributed to such issues [1], [2]. Therefore, comprehensive NoC verification is essential to identify and mitigate potential issues early in the design process, ensuring both the functional integrity and security of SoCs.

Despite advancements in fault-tolerant designs, the verification of NoC systems remains a significant challenge, particularly in ensuring that routing errors and communication faults are detected and mitigated. While simulation has been the traditional approach for verifying NoC designs, it is often insufficient for capturing the full scope of potential errors, especially as NoC systems become more intricate. Formal methods, although effective at verifying abstract models of NoC, struggle with the scalability and complexity of verifying Register Transfer Level (RTL) implementations. As routers interconnect to form larger networks, the verification of global NoC properties becomes exponentially more difficult, highlighting the need for more robust verification strategies.

This research was supported by the Technology Innovation Institute (TII), Abu Dhabi, UAE.

Nevertheless, given the limitations of formal methods in verifying larger and more complex NoC designs, simulation-based methodologies have become the preferred approach. In this context, the Universal Verification Methodology (UVM) is the industry-standard library for verification [3]–[5]. UVM provides a scalable and reusable framework that simplifies the generation of test cases and coverage analysis, thereby reducing the complexity associated with verifying intricate NoC architectures. However, the reliance on manually generated test cases still demands substantial human effort, making it challenging to achieve full coverage. As NoC designs continue to grow in complexity, the inefficiency of generating comprehensive test cases through manual methods becomes increasingly apparent.

Software fuzzing tools like libFuzzer [6] and AFL [7] have shown great promise in validating large and complex programs. When adapted for hardware verification, fuzzing has demonstrated significant potential [8]–[11]. However, hardware fuzzing techniques face several challenges that hinder their efficacy. These include the need for robust coverage monitoring to accurately measure and guide exploration, the challenge of efficiently directing the fuzzer to uncover new states without incurring excessive computational overhead, and the requirement for a reliable reset mechanism to ensure deterministic testing and manage the complex reset hierarchies of SoCs [11]. Furthermore, simulation tools like Verilator [12], which are often used in hardware fuzzing, present limitations in terms of scalability and compatibility with industrial verification environments. Another critical challenge is ensuring that the fuzzer can capture all hardware events, such as delays and clock generation in HDL, during simulation, as incomplete event tracking can lead to gaps in the verification process.

To address these challenges, we developed the *Interconnect Fuzzer* (*InterConFuzz*), a novel hybrid NoC fuzzing approach built on top of UVM that enhances coverage and operational efficiency while mitigating the limitations of existing simulation tools. Verilator [12], which cannot capture all hardware events, posed significant limitations. To overcome this, we opted for Vivado for simulation [13]. Although Vivado is slower than Verilator, it accurately simulates and captures critical hardware-specific events such as delays and clock generation, ensuring comprehensive event tracking during NoC architecture fuzzing. Furthermore, *InterConFuzz* employs a white-box verification model, enabling direct observation of internal variables within the NoC architecture, thus facilitating precise coverage monitoring and effective state tracking. A control flow graph (CFG) of the NoC is generated to capture branching information, guiding the placement of checkpoints that optimize state exploration. Integrating reset logic derived from the CFG into the RTL design reduces the need for frequent hard resets, ensuring deterministic tests while maintaining computational efficiency. *Moreover, InterConFuzz*

utilizes a two-step guidance mechanism that assesses the feasibility of reaching new states from the current state and employs the CFG to generate tailored seeds for exploring previously uncharted nodes of the NoC CFG. By addressing these challenges, including the need for comprehensive event tracking and the limitations of simulation tools, *InterConFuzz* significantly improves the scalability and effectiveness of NoC fuzzing in industrial verification settings. In this paper, our main contributions are:

- We present *InterConFuzz*, a fuzzing-based NoC verification methodology built upon UVM that systematically encompasses all states associated with the defined security properties. This facilitates its seamless integration into the hardware security development lifecycle (SDL).
- We develop a checkpoint mechanism using a reset operation, which reduces computational cost for executing the proposed NoC fuzzing approach. This enables partial reset to speed up the path exploration process through fuzzing.
- *InterConFuzz* is evaluated on the TL-UL NoC architecture used in the OpenTitan SoC [14], where it was able to identify five types of vulnerabilities, three more than the existing NoC verification methodology typically present in the NoC architectures.
- *InterConFuzz* demonstrates similar coverage gain with low variance compared to existing fuzzing-based verification approaches.

II. Background

A. Coverage-guided Fuzzing

Fuzzing generates tests to probe valid and invalid edge cases, identifying vulnerabilities by uncovering undefined behaviors [15], [16]. It uses mutation-based techniques, altering existing data, or generation-based methods, creating tests from protocols. Widely adopted, platforms like OSS-Fuzz and ClusterFuzz automate fuzzing, revealing security flaws in large codebases [17]–[22]. Coverage-guided fuzzing explores diverse program paths, leveraging run-time coverage data via instrumentation to refine inputs [23]–[26].

B. Symbolic Execution

Symbolic execution is a powerful technique for analyzing the relationship between program inputs and behavior [27]–[29]. Instead of concrete values, it uses symbolic representations for inputs, allowing the interpreter to create expressions that incorporate these symbols. Conditional branches are associated with constraints derived from symbolic values, representing the conditions under which different paths are executed. By solving these constraints, symbolic execution reveals how various inputs influence program execution, identifying which inputs lead to specific execution paths.

C. Universal Verification Methodology (UVM)

The UVM is a standardized framework based on SystemVerilog, designed to improve the verification process for complex digital designs, such as SoC architectures [3]. UVM promotes reusability, scalability, and flexibility by leveraging object-oriented programming principles, enabling the creation of reusable verification components like drivers and monitors. Its importance stems from its ability to streamline verification through constrained-random testing and functional coverage-driven techniques, ensuring comprehensive exploration of a design's behavior under various conditions. UVM's adoption across the verification industry is critical as it addresses the increasing complexity of modern designs while reducing development time and cost by reusing verification IPs. This aligns with industry demands for faster time-to-market and improved product reliability [3], [30], [31]. By offering a scalable and automated verification environment, UVM plays a crucial role in ensuring the functional correctness and robustness of semiconductor products. This is crucial since verification consumes a significant portion of the design flow.

III. Related Works

NoC architectures can be vulnerable to breaches with minimal impact [32], [33]. To secure SoC communication, methods like secure NoC fabric [34] and Design-for-Debug (DfD) features [35] are used. However, these security components can increase power and area costs, challenging the compact and energy-efficient design of SoCs.

Real-time detection and localization of denial of service (DoS) attacks have been proposed to prevent adversaries from exploiting the NoC architecture to execute DoS attacks on the SoC [36]. For example, network-based monitoring mechanisms utilizing unique hardware probes have been proposed to enable runtime observability of system-level security threats [37]. Furthermore, recent studies have explored methods for detecting NoC-based hardware Trojans at runtime [33], [38]. While various verification methods such as concolic testing [39], [40], fuzzing [8]–[10], information flow tracking [41]–[44], and runtime detection [45]–[47] have been developed for SoCs and IPs, few attempts have been made to develop a security verification method specifically tailored for NoC.

Despite these advances in ensuring security for NoCs, most techniques require architectural implementations to enforce security but lack verification strategies for assessing NoC security at the register-transfer level (RT-level). To address this gap, two methodologies have been developed for verifying NoC security: SeVNoC and NoC-Fuzzer [48], [49]. SeVNoC employs control flow graphs (CFGs) to identify vulnerabilities in inter-IP communication, while NoCFuzzer uses fuzzing techniques to verify the security of NoCs. However, these verification tools show limitations in coverage and bug detection capability as discussed in Section V.

IV. Proposed InterConFuzz Methodology

A. InterConFuzz Architecture

The *InterConFuzz* framework, illustrated in Fig. 1, integrates seamlessly into the UVM workflow for NoC security verification. The individual components are explained in the next paragraph; while a standard UVM verification environment consists of components ① to ⑥, *InterConFuzz* extends this with five additional blocks (⑦ to ⑪) to enhance the process through symbolic execution and input fuzzing. The fuzzing is introduced using UVM library functions, while symbolic execution is achieved with an SMT solver.

In a basic UVM verification workflow, the *Scoreboard* or ① is responsible for comparing the expected and actual outputs of the design under verification (DUV) ⑥ to verify its correctness. It typically receives predicted values from reference models and the actual values from the DUV, subsequently flagging any mismatches as errors. This ensures that the DUV behaves as intended according to the testbench specifications. The *UVM Sequencer* or ② generates and controls the flow of transaction-level stimuli to ⑥, managing the order and timing of test scenarios. It acts as a bridge between the testbench and the driver, ensuring proper data flow. The *UVM Driver* or ③ takes transactions from the sequencer and translates them into signal-level inputs for the DUV. It ensures that high-level commands are converted into physical stimuli, enabling the DUV to process the intended test cases. The *UVM Monitor* or ④ passively observes the DUV's interfaces, capturing signal activity and transaction data for analysis. It checks the DUV's behavior without interference and reports information for functional coverage

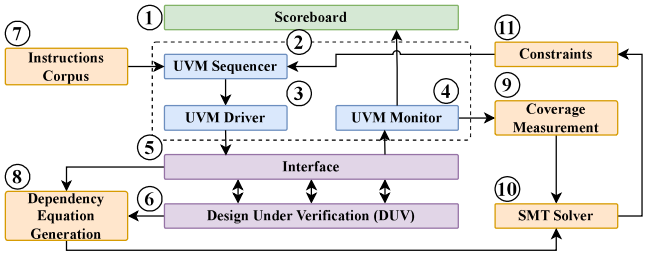


Fig. 1: *InterConFuzz* Architecture.

and checking, contributing to verification results. (6) is connected to (3) and (4) through the interface (5).

An instruction corpus (7), containing the necessary NoC function instructions, is compiled to verify the security of the NoC. This corpus is then linked with (2), enabling it to generate pattern sequences based on the corpus.

Dependency equations are extracted from the DUV through the Control Flow Graph (CFG) of the RTL design, which is generated using PyVerilog [50] to enable symbolic execution and enhance coverage. This process is illustrated by the connection of (8) with (5) and (6). These equations describe a Boolean relationship between the control registers and the I/O ports of the interface. The equations are then processed by the *z3-solver* [51] (10), an SMT solver, to generate and solve constraints (11). Additionally, a coverage measurement module (9) is linked to (4), evaluating the control register values to assess the achieved coverage. If the coverage remains stagnant for a user-specified duration, (9) triggers (10), which produces new constraints (11) to be integrated into (2).

B. *InterConFuzz Algorithm*

1) Defining UVM Sequencer and Driver: To randomize the instruction sequence for the NoC input, *InterConFuzz first defines a UVM sequencer tailored to the NoC architecture's specific instruction corpus.* The instructions are categorized based on their type, and each category's bit sequence is divided into components representing various commands, such as source address, destination address, opcode, flags, and more. For example, in the TL-UL bus protocol, the opcode is encoded in three bits, with an additional three bits allocated for parameters, a flag indicating data corruption, and such [52]. The protocol supports five message types: (1) *Get*, (2) *PutFullData*, (3) *PutPartialData*, (4) *AccessAck*, and (5) *AccessAckData*.

To execute specific functions, *InterConFuzz* begins by randomizing the input message sequences. Next, it identifies bit value packets such as addresses, data, and other fields. To perform fuzzing, *InterConFuzz* randomizes the values of these bit fields, generating diverse event scenarios while preserving the sequence structure. These modified inputs are then used by the UVM sequencer to drive the DUV with both functional and fuzzed input patterns.

To set up a complete UVM testbench, *InterConFuzz* first extracts the design interface from the NoC architecture, as described in detail in Algorithm 1, discussed in Section IV-B3. Upon defining the UVM sequencer, *InterConFuzz* attaches the sequencer to the UVM driver, and subsequently, the driver is connected to the design through the extracted interface. This integration establishes a fully functional UVM testbench, allowing for thorough testing of the NoC design.

2) Coverage Assessment: InterConFuzz uses control registers to measure coverage during fuzzing. Control registers dictate system behavior by controlling state transitions and monitoring specific operational parameters, enabling precise tracking in processes like

Algorithm 1: *InterConFuzz* Checkpoint Setup Algorithm.

Input: RTL Design, $\mathcal{D}$
Parameter: Interval, $\mathcal{I}$; Exit Threshold, Th
Output: Control Flow Graph, `CFG`; CheckPoints, `ChckPoints`

```
IF ← Extract(D)            /* IF = Interface */
CFG ←Generate— D;
Identify Branching Nodes;
Mark Control Registers;
for each Control Register, Reg_C do
    if branching at Reg_C > Br_th then
        append Reg_C to ChckPoints;
    end
end
Attach ChckPoints to CFG;
return CFG, ChckPoints
```

CFG traversal for coverage measurement. These registers offer a more accurate way to track coverage because they allow for precise monitoring of explored paths within the CFG, effectively capturing the test's progress and addressing coverage challenges commonly encountered in simulation.

Coverage is measured by running a simulation and analyzing the resulting dump file to see which paths were explored. Control registers are carefully monitored throughout this process as they define the sequence of states and transitions during the simulation. Since individual register values alone do not capture the full simulation dynamics—given the interactions between different control registers—it is essential to consider all possible combinations of these values. This approach allows a comprehensive measurement of coverage, with the total number of states calculated based on these register interactions.

To evaluate fuzzing coverage, *InterConFuzz* leverages control registers by generating a CFG from the RTL design. These control registers, which guide traversal through the CFG, enable precise coverage measurement, addressing associated challenges effectively. Following the simulation, the resulting dump file is analyzed to identify the paths explored during the test. *To ensure comprehensive state exploration, it is crucial to monitor these registers since they dictate the simulation's trajectory. Observing individual register values alone is insufficient due to potential interactions; therefore, all combinations of their values must be considered.* The total number of possible states is determined using the Eqn. 1.

$$No.\ of\ total\_states = \prod_{j=1}^{N} (n_j) \tag{1}$$

In Eqn. 1, N denotes the total number of control registers, with n_j representing the possible states for control register j. Coverage is measured by comparing the total number of possible states to the number of states that have been reached.

3) Simulation Setup: InterConFuzz utilizes Algorithm 1 to configure the simulation environment. In this process, *InterConFuzz* simulates the DUV with a specified interval, $\mathcal{I}$, and an exit threshold, Th. $\mathcal{I}$ determines the number of clock cycles over which the simulation runs before recording the simulation outcome as a VCD file. The exit threshold, Th, specifies the maximum number of consecutive simulation dumps during which the DUV can remain in the same

Algorithm 2: *InterConFuzz* CFG Traversal Algorithm.

Input: Interface, `IF`; Control Flow Graph, `CFG`; CheckPoints, `ChckPoints`;
Parameter: Interval, $\mathcal{I}$; Exit Threshold, Th
Output: Report, $\mathcal{R}$

1 **while** *All ChkPoints* ***not*** *Covered* **do**
2 $SimFile \xleftarrow{Dump\ VCD} Simulate(\mathcal{D})$;
3 $Coverage \xleftarrow{Record} Read(SimFile)$;
4 **if** *All states covered* **then**
5 **mark** current ChckPoint **as** covered
6 **end**
7 **if** $NoIncrement(Coverage) > Th$ **then**
8 **identify**(last covered state);
9 **find** (nearest ChckPoint);
10 **while** *No New State can be reached* **do**
11 **find** (previous ChckPoint in the `CFG`);
12 **end**
13 reset back to the ChckPoint;
14 Constraints ← SolveFor(newStateCondition);
15 $Sequencer \xleftarrow{Apply}$ Constraints;
16 **end**
17 **if** *Bug found* **then**
18 $\mathcal{R} \leftarrow B_{det}$ /* Bug Details, B_{det}, contain clock cycle time and violated property */
19 **end**
20 **end**
21 **return** $\mathcal{R}$

state; exceeding this limit prompts *InterConFuzz* to initiate symbolic execution for enhanced guidance. After completing the simulation, *InterConFuzz* produces a comprehensive bug report for the design.

InterConFuzz takes the RTL design of the NoC, $\mathcal{D}$, as its input. Initially, *InterConFuzz* extracts the I/O interface of $\mathcal{D}$ (line 1), enabling the attachment of the UVM driver to the RTL design. Following this, *InterConFuzz* generates the `CFG` of $\mathcal{D}$ (line 2) and identifies the branching nodes within this `CFG` (line 3). Subsequently, *InterConFuzz* identifies the control registers by determining those registers that influence the execution flow of the DUV (line 4). Using the `CFG`, *InterConFuzz* detects branch points and compiles a list of registers governing these branches. These registers are then classified as control registers and are employed to evaluate coverage metrics.

To avoid performing a full reset and ensure the fuzzer revisits specific points in the RTL design, InterConFuzz introduces checkpoints within the design's `CFG`. These checkpoints are strategically placed based on the design's branching structure. *InterConFuzz* selects nodes as checkpoints when the branching factor exceeds a predefined threshold, Br_{th}. Specifically, if a node and its subsequent sub-nodes generate more than Br_{th} branches, that node is marked as a checkpoint (lines 6-8). The threshold Br_{th} is configurable by the user: setting a higher Br_{th} results in fewer checkpoints, causing the fuzzer to revisit the same branches more often, while a lower Br_{th} increases the number of checkpoints, which demands more memory. Experimental results indicate that setting Br_{th} to three achieves an optimal balance for performance.

4) CFG Traversal and Constraint Generation: InterConFuzz follows Algorithm 2 to simulate $\mathcal{D}$, traverse through `CFG` and use symbolic execution to generate $Constraints$ to achieve higher coverage. Upon completion of the preliminary steps discussed in Section IV-B1, IV-B2, and IV-B3 *InterConFuzz* initiates a loop that continues until all checkpoints are covered (line 1). The initial step within this loop involves simulating until $\mathcal{I}$ is reached and subsequently generating a simulation record file (line 2). This record file is then processed by *InterConFuzz* to measure coverage using control register values (line 3). If the states associated with the current checkpoint are fully covered, *InterConFuzz* designates the checkpoint as "covered" (lines 4-6). Following this, *InterConFuzz* conducts a status check to assess the increase in coverage compared to previous checks.

In the absence of coverage improvement, InterConFuzz records the number of intervals during which coverage remains unchanged. When this count surpasses the designated exit threshold Th defined in Section IV-B3, InterConFuzz concludes that the fuzzer is experiencing stagnation and requires intervention to enhance coverage (line 7). To address this, *InterConFuzz* first identifies the most recent covered state by analyzing the latest simulation dump file and then determines the nearest checkpoint corresponding to this state (lines 8-9). Next, *InterConFuzz* initiates a procedure that navigates *upwards* in the CFG to locate the closest checkpoint leading to an unexplored state (lines 10-11). Here, *upwards* refers to moving toward the root node or initial default state within the CFG hierarchy. Once the appropriate checkpoint is found, *InterConFuzz* resets the simulation to this checkpoint (line 13) and uses the *z3-solver* [51] to establish constraints within the UVM sequencer (lines 14-15). These constraints guide the fuzzer's path, targeting the achievement of the newly identified state. By restricting specific sequences of NoC instructions, these constraints direct the UVM sequencer to produce input sequences that actively explore previously uncovered states. This guided approach allows *InterConFuzz* to generate instruction sequences with an increased probability of reaching unexplored states.

Conversely, if coverage improvement is detected, *InterConFuzz* continues fuzzing with the same setup. If a bug or property violation is identified during this process, *InterConFuzz* documents the specifics, including the clock cycle time and the violated property, and appends this information to the bug report, $\mathcal{R}$.

To execute checkpointing, *InterConFuzz* logs input pattern sequences as the fuzzer explores new states. When a checkpoint is reached, the sequence is recorded as the path to that checkpoint. If the fuzzer reaches a leaf node or a state with no further unexplored branches, *InterConFuzz* resets the design and replays the sequence to return to the checkpoint. Once all nodes from a checkpoint are visited, *InterConFuzz* removes the current checkpoint and promotes the previous one as the new target, repeating the process.

5) Bug Detection: InterConFuzz employs a property-based verification framework to identify security vulnerabilities in designs by leveraging a predefined set of properties that define key security requirements using the UVM monitor. During the fuzzing process, the simulator continuously monitors these properties for any violations. Upon detecting a breach, the UVM monitor generates a comprehensive report, including the name of the violated property, the precise simulation timestamp, the associated waveform, and the sequence of input vectors starting from activating the *reset* signal. This detailed information is then appended to $\mathcal{R}$.

V. Evaluation and Results

A. Evaluation Setup

We evaluated *InterConFuzz* using security vulnerability detection and coverage analysis. For security assessment, we compared *InterConFuzz* with SeVNoC [48] on the OpenTitan TileLink Unprivileged

TABLE I: Details for the Detected Bugs in the Benchmark SoC.

Bug No.	Bug Description	CWE Number	# of input vectors needed to detect the bug
01.	Omitted Wipe Operation After Transaction.	1331	5.85×10^5
02.	Modifying Destination.	668	4.39×10^5
03.	Modifying Source.	668	4.76×10^5
04.	False Transactions Leading to Denial-of-Service (DoS).	1315	7.38×10^4
05.	Corrupting Message Data	1331	6.34×10^5

(TL-UL) network [52]. For coverage evaluation, we benchmarked *InterConFuzz* against NoCFuzzer [49] on OpenPiton's [53] NoC.

1) Evaluation Platform Details: A server with AMD EPYC 7513 2.6GHz, 32C/64T, 128M Cache (200W) Processor, 16 DDR4-3200 64GB RDIMM, 3200MT/s, Dual Rank, 16 GB RAM, and four A100 SXM4 80GB 500W Nvidia GPUs was used to run the simulations. Xilinx Vivado is used as a simulator [13].

2) Parameter Setup: During fuzzing, the system is configured to dump simulation files at every third interval. An interval represents a user-defined number of clock cycles, which is 300 in this case. This setup strikes a balance between performance and resource utilization, ensuring *InterConFuzz* effectively saves the necessary simulation records for coverage analysis.

3) Evaluation Metrics: To assess the effectiveness of *InterConFuzz*, we concentrate on three metrics: (1) its bug detection capability, (2) coverage gain, and (3) computational resource consumption. We first compare the number of bugs *InterConFuzz* identifies against those found by existing fuzzing techniques. We then quantify the coverage achieved in terms of the number of coverage points–defined as the combined variations of control register values and input patterns generated. We also show the effect of symbolic execution and the performance in different parameter configurations, as well as the variance of achieved coverage to illustrate *InterConFuzz*'s efficacy. Moreover, we evaluate the computational resources and time required by *InterConFuzz* to verify the benchmarks in comparison to other methods, highlighting the efficiency of our approach.

B. Vulnerability Description

Table I summarizes the identified bugs, their CWE numbers, and the input vectors needed for detection. The following subsections provide detailed analysis and relevant RTL code snippets.

```
property reset_chk;
(@posedge clk_i) !rst_ni |-> !(|rdata);
endproperty
```

Listing 1: Security Property for clearing data.

a) ***Bug #1****:* This Bug prevents the system from clearing data in the buffers/registers after each transaction. As a result, residual data remains and gets overwritten with every new transaction.

Every new transaction overwrites the previous data, leading to potential data leakage. If the previous data was sensitive or critical, requesting another transaction can lead to storing the data in an unprotected memory address, leading to privacy violations. To detect this Bug, the property in Listing 1 is used, which dictates that upon a reset, `rdata` should be cleared.

b) ***Bug #2****:* This vulnerability allows alteration of a transaction's destination post-initiation. While the original transaction completes successfully, a copy of the data is covertly sent to an unauthorized location, like a UART port or another attacker-controlled channel. As shown in Line 2 and 3 in Listing 2, if the module's write data flag is raised, it sets the write request and data as adversary-defined.

```
1 if (!rst_ni) begin
2   wr_req_q <= (data_wr?) adv_req : 0;
3   wr_data  <= (data_wr?) adv_data : 0;
```

Listing 2: Writing Data to Another Destination.

This leads to a data breach, leaking confidential information to unauthorized locations. This major security threat was detected using the property shown in Listing 3, which requires `wr_req_q` and `wr_data` to be cleared upon reset.

```
property reset_chk;
(@posedge clk_i) !rst_ni |-> !(|wr_req_q)&&!(|wr_data);
endproperty
```

Listing 3: Security Property for Writing Data to Another Destination.

c) ***Bug #3****:* This security breach allows data injection into the system after a transaction starts. Initially, the system retrieves data from a legitimate source, but additional data is then injected from an attacker-controlled memory address. As shown in Line 3 and 4 in Listing 4, if the module's received flag is raised, it sets the receive source ID and source as the adversary's module ID and data size.

```
1 ...
2   reqid_q <= (data_r?) adv_id : 0;
3   reqsz_q <= (data_r?) adv_sz : 0;;
4   rspop_q <= AccessAck;
5   data_r  <= (@posedge clk_i) 0;
6 ...
```

Listing 4: Receiving Data from Another Source.

This can lead to data integrity issues, where unauthorized data is included in transactions. Such manipulation can result in incorrect or compromised data being sent, potentially disrupting operations or leaking sensitive information. This Bug was detected using the property in Listing 5, which asserts that `reqid_q` and `reqsz_q` should be cleared upon reset.

```
property reset_chk;
(@posedge clk_i) !rst_ni |-> !(|reqid_q)&&!(|reqsz_q);
endproperty
```

Listing 5: Security Property for Receiving Data from Another Source.

d) ***Bug #4****:* The bus initiates fake transactions, sending random values across the communication channel, causing congestion and delays for legitimate transactions, as shown in Listing 6. It initiates random write requests and data values if the adversary flag is raised.

```
wr_req_q <= (adv?) $RANDOM : 0;
wr_data  <= (adv?) $RANDOM : 0;
```

Listing 6: Initiating random write request to perform DoS attack.

This Bug leads to a Denial-of-Service (DoS) attack, in which the system's performance degrades significantly due to the time wasted processing these fake transactions. Legitimate tasks may experience delays or timeouts, reducing the overall system efficiency.

```
property reset_chk;
(@posedge clk_i) !rst_ni |-> !(|wr_req_q)&&!(|wr_data);
endproperty
```

Listing 7: Security Property for detecting DoS attack.

Here, the property in Listing 7 mandates that upon a reset signal, the `we_req_q` and `wr_data` should be set to Zero.

*e) **Bug #5**:* A rogue IP accesses the network and alters both inbound and outbound traffic, modifying data packets in transit and tampering with communication between components. As illustrated in Listing 8, the output write data register receives corrupted data before writing to the destination.

```
assign wdata_o=adv?`Mut_val^tl_i.a_data:tl_i.a_data;
```

Listing 8: Security property for error flag checking.

A rogue IP can manipulate exchanged data, compromising its integrity and introducing inaccuracies and security risks. This manipulation allows attackers to inject misleading data or responses, causing unintended operations. This property checks if `wdata_o` and `tl_i.a_data` are equal.

```
property reset_chk;
(@posedge clk_i) wdata_o == tl_i.a_data;
endproperty
```

Listing 9: Writing Data to Another Destination.

C. Comparative Analysis of InterConFuzz and SEVNoC

In this section, we present a comparative analysis of *InterConFuzz*'s detection capabilities relative to existing NoC security verification approaches, particularly SEVNoC [48]. Our findings reveal that while SEVNoC successfully detected certain bugs, it was unable to identify three significant issues: Bug #1, Bug #2, and Bug #3. *In contrast, InterConFuzz demonstrated a higher degree of reliability, detecting all five bugs within the NoC, indicating InterConFuzz's potential for comprehensive NoC verification.*

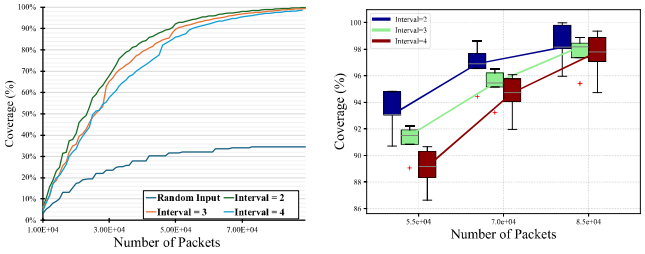
Our investigation found that SeVNoC overlooked Bug #1 because the buffers were often overwritten while evaluating the CFG of the SoC, resulting in an inability to detect mismatches due to not covering all corner cases. Furthermore, SeVNoC failed to identify Bug #2 and Bug #3 since SeVNoC considers a process safe if it completes functions without initially violating security properties. Consequently, these two bugs were not detected by SeVNoC. On the other hand, *InterConFuzz* detected all bugs by reaching the corner cases where the device was reset and initiated the transaction, causing previously written data in the buffer to be sent. Furthermore, *InterConFuzz* does not use a flagging technique to mark paths of the CFGs as secure. As a result, when the corner cases triggered Bug #2 and Bug #3, *InterConFuzz* flagged them as breaching security.

Finally, *our study found that InterConFuzz is more efficient, using 24.4% less memory and 29.5% less computational power compared to SeVNoC.* This demonstrates *InterConFuzz*'s enhanced effectiveness and efficiency in bug detection tasks.

D. Coverage Assessment

In this section, we provide a comparative study of the coverage furnished by *InterConFuzz* against existing approaches. To this end, we used control registers to assess coverage. Each branch of the control path is considered a coverage point for *InterConFuzz*. When multiple control path combinations are possible simultaneously, each combination of taken paths is considered a coverage point, as elaborated in Section IV-B2.

Randomness and Impact of Interval Parameter in Fuzzing: Figure 2 illustrates the coverage improvements of *InterConFuzz* and the variance in coverage gain. Figure 2a and Figure 2b show the coverage gain under different parameter configurations and the variance during three distinct fuzzing stages, respectively.



(a) Coverage comparison with existing fuzzing techniques.

(b) Variance of Coverage for Different Interval Configuration.

Fig. 2: Coverage comparison and variance profile.

Figure 2a underscores the impact of symbolic execution with different parameter configurations in guiding the fuzzing process. Random input generation without symbolic execution achieves only about 30% of the coverage attained with symbolic execution. *InterConFuzz* was tested with three configurations, altering the '$Interval$' parameter. Setting the '$Interval$' to 2, as shown in Figure 2a, results in better coverage but requires approximately 42.1% more processing power due to frequent SMT solver invocations. On the other hand, while setting the '$Interval$' value to 4 reduces processing power by 13.6%, the coverage gain is much slower. Therefore, realizing an '$Interval$' value of 3 provides the best outcome.

Figure 2b shows the variance in *InterConFuzz*'s coverage gain across different parameter setups. Within the selected data range, the impact of the initial randomness is settled down, and the fuzzing algorithm takes the primary effect. *InterConFuzz* displays a maximum coverage gain variance of 2.64%, indicating consistent performance. Notably, with an '$Interval$' value of 2, variance changes more frequently due to increased SMT solver invocations, leading to frequent coverage changes. Conversely, with an '$Interval$' value of 4, the coverage variance remains largely stable.

Comparison with NoCFuzzer: As shown in Figure 2a, the coverage improvement achieved by *InterConFuzz* aligns with the trends observed in the existing NoC fuzzing approach (NoCFuzzer) [49]. Moreover, *Figure 2b demonstrates the consistency of InterConFuzz's coverage, indicated by its low variance, confirming the reliability of InterConFuzz in delivering thorough and deterministic coverage.*

VI. Conclusion

This paper introduces *InterConFuzz*, a novel, coverage-directed hybrid hardware fuzzing framework specifically designed to enhance security verification for NoC architectures. By effectively combining coverage-directed fuzzing with symbolic execution, *InterConFuzz* strategically navigates the fuzzer into previously unexplored states of the NoC, optimizing both the efficiency and efficacy of the verification process. Integrated with UVM, *InterConFuzz* ensures seamless incorporation into existing security verification lifecycles, facilitating its adoption within industry practices. In evaluations conducted on a TL-UL bus network, *InterConFuzz exhibited superior bug detection capabilities, identifying three more bugs than current methodologies while reducing memory consumption by 24.4% and computational demands by approximately 29.5%.* Moreover, *InterConFuzz* achieved similar coverage when evaluated on OpenPiton's NoC architecture, underscoring its efficiency. These compelling results highlight *InterConFuzz*'s potential as a robust and efficient security verification framework for NoC architectures, positioning it as an exceptional option for integration into the security development lifecycle (SDL) of commercial SoC design flows.

REFERENCES

[1] P. Patra, "On the cusp of a validation wall," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 193–196, 2007.

[2] F. Farahmandi *et al.*, *System-on-Chip Security*. Springer International Publishing, Nov. 2019. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-30596-3_1

[3] S. V. Academy, "Universal verification methodology," Online, 2024. [Online]. Available: https://verificationacademy.com/topics/uvm-universal-verification-methodology/

[4] K. Salah, "A uvm-based smart functional verification platform: Concepts, pros, cons, and opportunities," in *2014 9th International Design and Test symposium (IDT)*. IEEE, 2014, pp. 94–99.

[5] W. Ni *et al.*, "Research of reusability based on uvm verification," in *2015 IEEE 11th International Conference on ASIC (ASICON)*. IEEE, 2015, pp. 1–4.

[6] "Libfuzzer – a library for coverage-guided fuzz testing. — llvm 12 documentation," accessed: 08/28/2024. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[7] "American fuzzy lop (afl) fuzzer." 2014. [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt

[8] K. Laeufer *et al.*, "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.

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

[10] T. Trippel *et al.*, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254.

[11] R. Kande *et al.*, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3219–3236.

[12] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[13] "Xilinx vivado," accessed: 05/19/2024. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug904-vivado-implementation.pdf

[14] "Opentitan | documentation," accessed: 05/19/2024. [Online]. Available: https://opentitan.org/book/doc/introduction.html

[15] R. Craig *et al.*, *Systematic Software Testing*, ser. Artech House ITS library. Artech House, 2002. [Online]. Available: https://books.google.com/books?id=2_gbZYZcZXgC

[16] M. Sutton *et al.*, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[17] X. Zhu *et al.*, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[18] A. Takanen *et al.*, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[19] K. Serebryany, "OSS-Fuzz - google's continuous fuzzing service for open source software," in *Proceedings of the 26th USENIX Security Symposium*. Vancouver, BC: USENIX Association, Aug. 2017.

[20] L. Dukes *et al.*, "A case study on web application security testing with tools and manual testing," in *2013 Proceedings of IEEE Southeastcon*. IEEE, 2013, pp. 1–6.

[21] W. Xu *et al.*, "Freedom: Engineering a state-of-the-art dom fuzzer," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 971–986.

[22] https://google.github.io/clusterfuzz/, accessed: 04/03/2024.

[23] Z. Michal, https://lcamtuf.coredump.cx/afl/, accessed: 04/03/2024.

[24] M. Böhme *et al.*, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[25] M. Beckmann *et al.*, "Coverage-guided fuzzing of embedded systems leveraging hardware tracing," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 362–378.

[26] A. Fioraldi *et al.*, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[27] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[28] R. Baldoni *et al.*, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[29] C. Cadar *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[30] H. Park *et al.*, "Strategies to maximize reusability of uvm test scenarios in soc verification," *DV-Con*.

[31] S. Vasudevan, *Practical UVM: Step by Step with IEEE 1800.2*. R R BOWKER LLC, 2020. [Online]. Available: https://books.google.com/books?id=LY9qzQEACAAJ

[32] D. M. Ancajas *et al.*, "Fort-nocs: Mitigating the threat of a compromised noc," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[33] T. Boraten *et al.*, "Mitigation of denial of service attack with hardware trojans in noc architectures," in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 1091–1100.

[34] A. P. D. Nath *et al.*, "Resilient system-on-chip designs with noc fabrics," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2808–2823, 2020.

[35] A. Basak *et al.*, "Exploiting design-for-debug for flexible soc security architecture," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[36] P. K. Mishra *et al.*, "Real-time detection and localization of dos attacks in noc based soc architectures," Oct. 24 2023, uS Patent 11,797,667.

[37] C. Ciordas *et al.*, "An event-based monitoring service for networks on chip," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 10, no. 4, pp. 702–723, 2005.

[38] V. Y. Raparti *et al.*, "Lightweight mitigation of hardware trojan attacks in noc-based manycore computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[39] L. Zhao *et al.*, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." Network and Distributed System Security Symposium (NDSS), 2019.

[40] 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.

[41] M. Tiwari *et al.*, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 189–200, 2011.

[42] X. Li *et al.*, "Caisson: a hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[43] D. Zhang *et al.*, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[44] K. Ryan *et al.*, "Augmented symbolic execution for information flow in hardware designs," *arXiv preprint arXiv:2307.11884*, 2023.

[45] M. Hicks *et al.*, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 517–529.

[46] S. R. Sarangi *et al.*, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 26–37.

[47] I. Wagner *et al.*, "Engineering trust with semantic guardians," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[48] X. Meng *et al.*, "Sevnoc: Security validation of system-on-chip designs with noc fabrics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 2, pp. 672–682, 2022.

[49] R. Ma *et al.*, "Nocfuzzer: Automating noc verification in uvm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[50] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42

[51] L. De Moura *et al.*, "Z3: An efficient smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[52] "Sifive tilelink specification," accessed: 08/28/2024. [Online]. Available: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[53] J. Balkind *et al.*, "Openpiton: An open source manycore research framework," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 217–232, 2016.