NoCFuzzer: Automating NoC Verification in UVM

Ruiyang Ma, Jiayi Huang, *Member, IEEE*, Shijian Zhang, Yuan Xie, and Guojie Luo, *Member, IEEE*

***Abstract*—Network on chip (NoC) has surfaced as a crucial interconnection strategy in modern digital systems, thereby demanding meticulous verification. Due to its multiple nodes and high concurrency, verifying an NoC is labor-intensive, making it complex to generate a multitude of test cases. Recently, hardware fuzzing has been identified as a promising automated approach for hardware verification. However, when we tried to apply these fuzzing techniques to our internally developed NoC design, we discovered that they were incompatible with the specificities of NoC. Additionally, they are also incompatible with the standard IC verification workflow and universal verification methodology (UVM) environment. In this work, we aim to automate our verification process of NoC with fuzzing. We propose a fuzzing strategy specifically tailored for industrial NoC UVM verification. We employ fuzzing in NoC verification at multiple levels, including router verification, network verification, and stress testing. As a case study we apply our approach to an open-source NoC component in OpenPiton. Remarkably, our fuzzing methods automatically achieved complete code and functional coverage in the router and mesh network. We also effectively detect injected starvation bugs with fuzzing. The evaluation results clearly demonstrate the practicability of our fuzzing approach to considerably reduce the manpower required for test case generation compared with traditional NoC verification.**

***Index Terms*—Automatic test generation, design verification, hardware fuzzing, network on chip (NoC).**

I. Introduction

IN MODERN digital systems, network on chip (NoC) plays an important role in serving as an interconnection solution, which supports high-level protocols and applications [1], [2], [3], [4]. Therefore, it is of utmost importance to ensure the correctness of NoC, as any undetected errors can result in the failure of the entire chip. However, due to the complexity of NoC systems, characterized by its multiple nodes and high concurrency, the verification of NoC is not only critical but also labor-intensive. The verification of NoC involves two primary techniques: 1) formal verification and 2) dynamic simulation.

Formal methods generally construct abstract models of NoCs in order to verify certain properties [5], [6], [7], [8], [9]. However, these methods often face significant challenges when it comes to verifying the RTL implementations of NoC due to the complexity of extracting NoC properties at this level. Additionally, scalability issues frequently arise. When routers are interconnected to form a network, the problem-solving complexity for some global NoC properties significantly escalates.

As a result, simulation-based methods have become a preferred choice for NoC verification [10], [11], [12]. In dynamic methods, numerous test cases are manually written, aiming to achieve full coverage in the test plan. This process heavily relies on skilled verification personnel to analyze coverage statistics and make real-time adjustments to test constraints. Within dynamic verification, the universal verification methodology (UVM) is a mainstream solution [13], [14], [15], [16], [17]. UVM provides a robust framework that promotes the development of reusable, interoperable, and scalable verification environments. This greatly simplifies the test case generation and coverage analysis process. However, despite these advancements, the generation of comprehensive test cases for NoC systems still necessitates substantial human intervention.

A variety of strategies have been proposed to automate the generation of test inputs. In recent years, hardware fuzzing has emerged as a popular and scalable solution [18], [19], [20], [21], [22], [23], [24], [25], [26]. Within this approach, coverage is typically used as guidance. New inputs are generated by mutating previously interesting inputs (i.e., the inputs that increase coverage), thereby effectively exploring hardware behaviors. Some software fuzzers, such as libFuzzer [27] and American Fuzzy Lop (AFL) [28], can function as heuristic mutation engines in hardware fuzzing.

However, applying these previous hardware fuzzing efforts to industrial NoC UVM verification has proven challenging. To accelerate the collection of coverage data within fuzzing loop, these efforts employ innovative coverage metrics as guidance, such as multiplexer coverage in RFUZZ [18] and software model coverage of DUT in HW-Fuzz [19]. However, these coverage metrics diverge significantly from those employed in

Manuscript received 30 January 2024; revised 12 May 2024; accepted 29 June 2024. Date of publication 17 July 2024; date of current version 26 December 2024. This work was supported in part by the Alibaba Innovative Research (AIR) Program; in part by the National Natural Science Foundation of China under Grant 62090021; in part by the National Key Research and Development Program of China under Grant 2022YFB4500500; in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2023A1515110353; in part by the Guangzhou Municipal Science and Technology Project under Grant 2024A04J4528; and in part by the Guangzhou-HKUST(GZ) Joint Funding Program under Grant 2024A03J0624. This article was recommended by Associate Editor H. Zheng. *(Corresponding author: Guojie Luo.)*

Ruiyang Ma and Guojie Luo are with the School of Computer Science and the Center for Energy-Efficient Computing and Applications, Peking University, Beijing 100000, China (e-mail: ruiyang@stu.pku.edu.cn; gluo@pku.edu.cn).

Jiayi Huang is with the Microelectronics Thrust, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou 510530, China (e-mail: hjy@hkust-gz.edu.cn).

Shijian Zhang is with the Computing Technology Lab, Alibaba DAMO Academy, Hangzhou 310023, China (e-mail: zsj269889@alibaba-inc.com).

Yuan Xie is with the Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong, China (e-mail: yuanxie@ust.hk).

Digital Object Identifier 10.1109/TCAD.2024.3430195

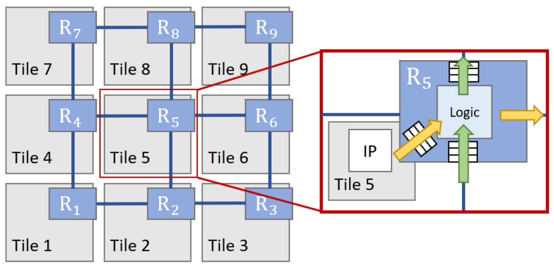


Fig. 1. NoC-based MPSoC.

industrial verification, including code coverage and functional coverage, thus leading to a significant departure from standard IC design and verification processes. Furthermore, these fuzzing methods rely on open-source hardware workflow. They make use of Verilator [29] as simulator, which offers limited support for SystemVerilog and UVM and differs greatly from industrial verification environments. Some research has attempted to create FIRRTL passes to facilitate fuzzing, which is also infeasible to most industrial DUTs written in Verilog.

In this article, we demonstrate the application of fuzzing in an industrial NoC verification process. We summarize four key contributions as follows.

1) We build an innovative *UVM fuzzing framework* that integrate software fuzzer AFL with conventional UVM verification environment.
2) We create an effective *multiport fuzzing grammar* to generate input transactions for NoC, which is also adaptable to other DUTs with multiple ports.
3) We assess the practicality of *employing fuzzing in an industrial NoC verification workflow*. Utilizing fuzzing metrics, we achieve coverage closure at both router and network verification levels. Additionally, we propose a novel fuzzing metric to detect overtime-related vulnerabilities in NoC systems.

To evaluate our fuzzing methodology, we conducted a case study using the NoC design within OpenPiton [30]. By employing fuzzing, we were able to achieve 100% code and functional coverage for both router and mesh network automatically. Our method has also successfully enabled the automatic detection of injected starvation bugs. Impressively, our approach outperformed industry-standard random approach and constrained random verification (CRV) [31], demonstrating the adaptability of fuzzing in practical NoC verification processes and its potential to reduce the need for human involvement.

In the instance of NoC fuzzing, our study also reveals novel insights for verifiers considering the application of fuzzing in industrial hardware verification processes. These knowledge includes the principles of hardware fuzzing grammar design, the implementation of functional coverage fuzzing, and the strategy for fuzzing seed selection. This study builds upon our preliminary research [32].

II. Background

A. Network-on-Chip Description

Fig. 1 shows an example of a 9-tile multiprocessor system-on-chips (MPSoCs) interconnected through a 9-router mesh-based NoC. As illustrated in the example, NoC is responsible for the exchange of data among the tiles. Packet is the basic data unit in NoC, which is subdivided into smaller units known as flits. A flit is not only the smallest manageable data unit in NoC but also the basic unit for bandwidth and buffer capacity allocation. A packet usually consists of one or more flits, which includes a header flit (containing destination address and other control information), payload flits (containing the actual data), and a tail flit (signifying the end of the packet).

The NoC comprises of a series of routers, each designed to route data from an input port to an output port. Neighboring routers are connected through bidirectional links. In a typical 2-D mesh topology, each router has five ports: 1) North; 2) East; 3) South; 4) West; and 5) Local. The port number of a router can differ based on the NoC topology. The router's structure consists of four primary components.

1) *Input/Output Buffers:* These buffers serve to store the data that request the router by one of its input ports.
2) *Route Computation:* This component is responsible for selecting the output port to redirect incoming data. It determines the path that packets will traverse through the network.
3) *Crossbar Switch:* This central switching mechanism links input ports to output ports, enabling multiple concurrent connections between various input and output ports.
4) *Arbitration Logic:* Arbiter decides which input buffer is permitted to utilize the crossbar switch at a given moment. In situations when multiple requests from different input ports target the same output port, the arbiter grants communication privileges to a single request.

In NoC Systems, certain properties, such as *starvation-free* and *deadlock-free*, hold paramount importance. The starvation-free property ensures that no input request gets stuck forever because of competition of other requests from different ports. The deadlock-free property, on the other hand, safeguards against the potential for packet stagnation caused by a cyclical wait for the release of resources.

NoC verification is an extremely labor-intensive process due to the inherent complexity of NoC systems. Ensuring that test cases cover all possible scenarios presents a significant challenge. NoC designs typically comprise *a multitude of input ports and switching nodes*. Every node and input port requires meticulous verification to ensure functional correctness. Moreover, the *concurrent nature of NoC designs* poses an additional verification challenge. Constructing test cases that can effectively evaluate concurrent scenarios is a demanding task, as it requires consideration of all possible concurrent events and their interactions.

B. Hardware Fuzzing

Fig. 2 shows the basic flow of hardware fuzzing. The fuzzing procedure starts with a random selection from the seed pool. This seed undergoes mutation to generate a new input, which is then compiled into a DUT-compatible stimulus by a preprocessor. An RTL simulator then simulates the DUT and collect coverage data. Inputs revealing new hardware states (i.e., new coverage) are retained as seeds for future fuzzing

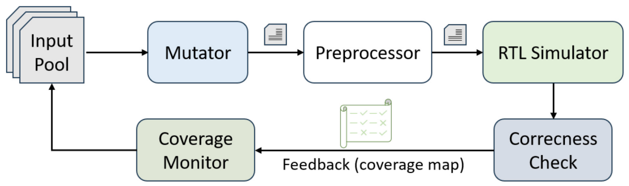


Fig. 2. Basic hardware fuzzing flow.

iterations. The process repeats until no new states emerge for a set number of iterations. Throughout, assertions or golden model cross-checks ensure DUT's correct behavior.

Since the collection of coverage is time-consuming, hardware fuzzing processes often take hours or even days to complete. To address the challenge of long runtime, novel coverage metrics have been designed as a speed-up strategy. For instance, RFUZZ [18] and DifuzzRTL [21] employ netlist-level coverage metrics, which enable FPGA acceleration during hardware fuzzing. HW-Fuzz [19] translates an HDL design into a software equivalent model using Verilator [29] and applies software fuzzing method directly.

However, these novel methods cannot be accepted in the industry. First, there exists a *major gap between novel and traditional coverage metrics*. Mapping these novel types of coverage to traditional coverage, such as line coverage and branch coverage, presents a significant challenge. Second, existing fuzzing methods *considerably deviate from mainstream standard IC verification process*. They rely on open-source simulators, which are not fully compatible with commonly used verification environments like SystemVerilog and UVM. In this study, we focus on the application of fuzzing in the traditional industry-standard NoC verification workflow.

C. AFL Fuzzer

In hardware fuzzing, an important part is using coverage to guide the input generation of subsequent fuzzing iterations. This process necessitates a fuzzer to select interesting seeds and perform mutations to heuristically explore design spaces. Many hardware fuzzing studies [18], [19], [20], including ours, use the AFL [28] as their mutation engine.

AFL is a well-established software fuzzer. When adapting AFL for hardware fuzzing, we bypass its built-in coverage instrument for software program, and use the coverage data collected from hardware simulator to guide AFL's heuristic algorithm. AFL generates binary test files as input, which necessitates a translation strategy from binary into valid hardware stimuli, specifically NoC packets in our work.

AFL's mutation strategy is grammar-agnostic and targets the binary files. Inputs are chosen from seed pools and are mutated using deterministic and nondeterministic methods. The deterministic stage mutate every position in the input, which includes operations, such as bitflip, arithmetic add/sub, trim, splice, etc. The nondeterministic stage are performed in the havoc stage of AFL. In each application of the havoc mutation, between 2 and 128 random mutations are performed on the parent input. Given AFL's maturity as a software fuzzer and its highly efficient heuristic strategies, it serves as an excellent mutation engine for hardware fuzzing, saving significant time compared to developing one from scratch.

D. UVM Basis

The UVM is a verification framework built primarily on the SystemVerilog class library. UVM constructs a verification platform framework composed of several key concepts: transaction, driver, monitor, agent, sequence, sequencer, reference model, and scoreboard. These components work together to accomplish hardware verification tasks.

Transaction: It is an abstract data unit in UVM testbench, modeling real-world events like data packets. It is also the basic unit for input generation in our NoCFuzzer framework.

Driver: The driver is responsible for converting abstract transactions into electrical signals. In our NoC testbench, it translates NoC packets into stimulus according to the NoC packet protocol.

Monitor: The main task of monitor is to observe the inputs and outputs of the DUT, transforming the DUT's electrical signal data into a transaction format.

Sequence: A sequence typically includes a series of transactions. In the process of fuzzing, NoCFuzzer generates a sequence for each NoC port during each iteration.

Sequencer: The main role of sequencer is to extract transactions from the Sequence and send them to the DUT at the appropriate times.

Agent: The task of agent is to centrally manage everything related to the DUT interface. An Agent typically includes driver, monitor, and sequencer.

Reference Model: It models the behavior of the DUT and offers a reference of the DUT's expected behavior.

Scoreboard: Scoreboard is used to compare the output data between the DUT simulation and the reference model to verify the correctness of DUT.

UVM allows verification engineers to construct functional verification environments with standardized hierarchical structures and interfaces using reusable components. In the following sections, we will introduce the process of constructing a UVM testbench for NoC and the integration of our NoCFuzzer into this environment.

III. Traditional NoC Verification

In this section, we use the NoC in OpenPiton [30] as the running example to explain the traditional NoC verification and highlight its shortcomings. Section III-A provides a comprehensive overview of the OpenPiton NoC, including a detailed explanation of the functional covergroup settings. Following this, Section III-B introduces the UVM environment constructed specifically for the OpenPiton NoC. Finally, Section III-C outlines the entire lifecycle of the traditional NoC verification process.

A. OpenPiton NoC

1) NoC Design: OpenPiton is the world's first open-source, general-purpose, multithreaded manycore processor [30]. The design is implemented in industry standard Verilog HDL. OpenPiton builds upon the industry OpenSPARC T1 cores [33] and is designed for scalability both intrachip and interchip. Intrachip, tiles are connected via NoCs in a 2-D mesh topology. Interchip, a chip bridge serves as an interface to off-chip logic. The NoC primarily facilitates

TABLE I
COVERAGE CLOSURE TARGET IN FUZZING

Coverage	Description	Pseudo Code	Point Count
Line Coverage	Number of Verilog basic blocks executed during testing.	-	733
Branch Coverage	Number of Verilog branch trace points executed during testing.	-	839
Router Covergroup1	Possible states of requests (enabled or disabled) from five input ports to a single output port arbiter. Each output port has one associated covergroup.	for output port x: *cross* req$_{n\_to\_x}$, req$_{e\_to\_x}$, req$_{s\_to\_x}$, req$_{w\_to\_x}$, req$_{p\_to\_x}$	800
Router Covergroup2	Possible states of request sending directions (free or directed towards NWESP) from five input ports.	*cross* input-req$_n$, input-req$_e$, input-req$_s$, input-req$_w$, input-req$_p$	7776
Router Covergroup3	Possible states of five output ports' granting (free or granted to NWESP input port). Some state combinations are impossible (*e.g.*, multi output ports granting to one input port).	*cross* output-gnt$_n$, output-gnt$_e$, output-gnt$_s$, output-gnt$_w$, output-gnt$_p$	1546
Mesh Covergroup1	Possible states of buffers usage for selected local input ports during data send request in 3×3 Mesh. We divide 16-size buffer into 6 states.	*cross* buffer-P$_{(0,0)}$, buffer-P$_{(0,1)}$, buffer-P$_{(1,0)}$, buffer-P$_{(1,1)}$	1296
Mesh Covergroup2	Possible states of buffers usage of five input ports of the center router during data send request in 3×3 Mesh. We divide 16-size buffer into 6 states.	*cross* buffer-N$_{(1,1)}$, buffer-E$_{(1,1)}$, buffer-S$_{(1,1)}$, buffer-W$_{(1,1)}$, buffer-P$_{(1,1)}$	486
Mesh Covergroup3	Possible states of request sending destinations (free or sent to other 8 routers) from selected local input ports in 3×3 Mesh.	*cross* input-req$_{(0,0)}$, input-req$_{(0,1)}$, input-req$_{(1,0)}$, input-req$_{(1,1)}$	6561

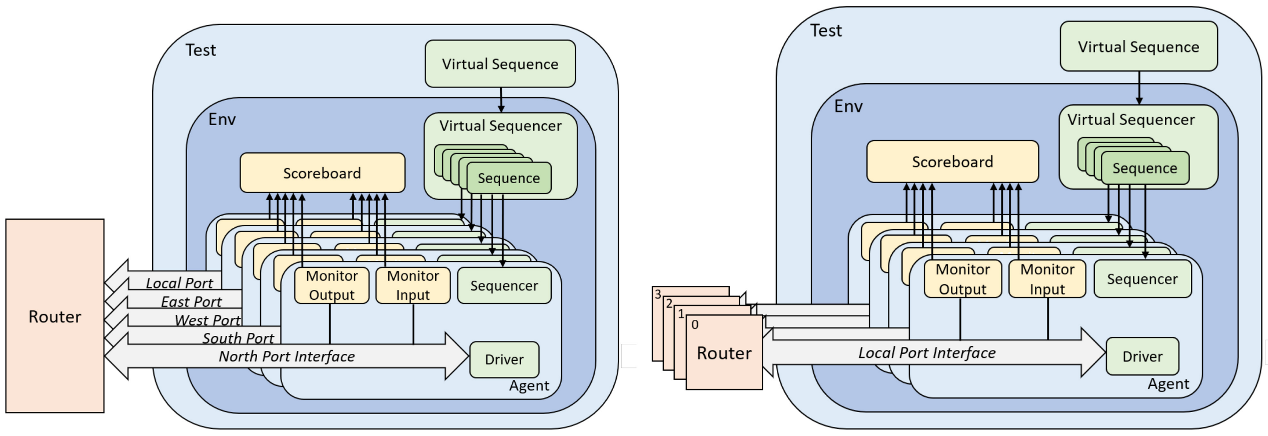


Fig. 3. NoC router UVM environment and network UVM environment.

a distributed, directory-based cache coherence protocol in OpenPiton.

In our research, we are not concerned with the high-level protocol operating on the NoC, as its content remains agnostic to the NoC itself. Instead, we view the NoC as a sophisticated component that provides service to high-level CPU applications, and our primary objective is to verify the functional correctness of the NoC. It should be noted that our focus is strictly limited to the aspect of intrachip NoC communication. Discussions pertaining to interchip communication involving the chip bridge and chipset is beyond our study.

OpenPiton NoC is dynamic, providing a packetized, fire-and-forgot interface. Each NoC consists of *two 64-bit uni-directional links*, one for each direction. The NoC relies on a *credit-based flow control* [34]. Packets are routed by *XY dimension-ordered wormhole routing* [34] to avoid deadlocks. Each hop takes one cycle when packets are going straight and one extra cycle for route calculation when a packet must make a turn at a switch. Within each router, there is a fully connected crossbar, which allows all-to-all five-way communication. Each NoC packet contains a header word denoting the *x* and *y* destination location for the packet along with the packet's length.

While there exist some other open-source NoC projects [35], [36], [37], none of them offer industry-level and comprehensive Verilog NoC designs like OpenPiton. Given that our work aims to demonstrate the feasibility of using fuzzing to automate industry-standard NoC verification, to maintain a strong correlation with practical applications, we select the NoC in OpenPiton as our primary test target in the following experiments.

2) NoC Functional Coverage: In the NoC testplan, verification of the router and network necessitates the closure of code coverage, such as line coverage and branch coverage. Additionally, functional coverage, which is determined by the verifier, is also crucial. Achieving functional covergroups closure proves to be the most challenging. Table I provides an overview of the coverage groups which we utilize as fuzzing targets in the test plan.

Approximately 80% of code coverage can be effortlessly achieved through random testing. Similarly, many cover-groups, especially those of smaller size or with simpler conditions, can be easily covered through the same method. We focus primarily on covergroups that cannot be fully completed by random testing. Additionally, we also select two covergroups, specifically `router covergroup3` and `mesh covergroup3`, which can be easily covered by random testing, to draw comparisons in the subsequent experiments.

For the router, we select three covergroups of varying sizes as examples. `router covergroup1` tests the arbiter's behavior, while `covergroup2` and `covergroup3` both concentrate on the router's behavior and the correct crossbar switching.

For network, we have chosen a 3×3 Mesh Network as our DUT for the upcoming experiment. We are primarily concerned with the global behavior of the NoC, including the states of the sending destination and buffer usage. To prevent the covergroup size from becoming excessively large, we have selected the top-left four routers in 3×3 Mesh Network for observation within `mesh covergroup1` and `covergroup3`. Given the symmetry of a Mesh Network, the states of these four routers can adequately represent the Mesh to a certain extent. We also observe the buffer usage of the center router with `mesh covergroup2`.

B. NoC UVM Testbench

In this section, we build the UVM environment for the NoC in OpenPiton. Given the complexity of NoC architectures, characterized by multiple ports, high concurrency, and diverse combination patterns, creating test cases directly in SystemVerilog proves challenging. UVM provides an effective solution with its strong abstraction and reusability capabilities.

We have implemented a two-step verification process using UVM, first focusing on the router, then expanding to the network system. This design strategy primarily draws inspiration from the study outlined in [10]. Fig. 3 shows an overview of our NoC UVM testbench.

In NoC UVM testbench, a transaction is referred to as an NoC packet, which is composed of several flits. The packet's properties in OpenPiton NoC include content, length, and destination address. Both router and network UVM environments utilize the same type of agent, encompassing a driver and monitor. These are responsible for converting NoC packets into actual hardware stimuli that comply with the OpenPiton NoC protocol. In both router and network verification, every input port is allocated a dedicated agent.

Because OpenPiton's NoC uses a mesh topology, the router UVM environment deploys five agents, each corresponding to the north, east, south, west, and local input ports. For network verification, the number of agents is contingent on the mesh size. Each local port of the router is assigned an agent.

To check the correctness of NoC, we use a scoreboard to collect transactions sent from input monitors, then observe whether each packet is dispatched to the corresponding output port. Each collected transaction is passed to a predictor to determine which output port should the packet be assigned to. In router testing, the predicted output port should align with the routing strategy. In network testing, the predicted output port should match the local port of the destination router. The scoreboard also collect transactions from output monitors. If all packets are matched correctly, this means that all packets were correctly sent and received through the DUT.

In addition to the basic UVM components discussed in Section II-D, *Virtual Sequence* is employed for NoC. It provides hierarchical levels of sequences. Virtual sequence do not send any transactions or sequence items to the DUT. They only initiate subsequences and assign these sequences to the corresponding agent of each NoC input port. Subsequences then transmit transactions to the driver and finally to the DUT. In our NoCFuzzer framework, a virtual sequence is generated for UVM testbench in each fuzzing iteration.

C. NoC Verification Lifecycle

Following conventional hardware verification principles, the verification of OpenPiton's NoC can be approached in three stages: 1) router verification; 2) network verification; and 3) stress testing [10], [38], [39].

Router verification, the first phase, validates the functionality of an individual router. This phase scrutinizes key router operations, including the arbitration strategy, routing computation, and crossbar functions.

Network verification, the second phase, integrates individual routers into the larger network structure. This phase assesses the global behavior of the network, including the effective communication between routers, network latency, and the accurate delivery of packets.

Stress testing, the final stage, is designed to uncover any latent bugs that may not be detected in the preceding stages. By subjecting the NoC system to extreme conditions and high traffic loads, this phase tests the resilience and stability of the system, potentially revealing hidden vulnerabilities, such as packet drops, starvation, deadlock, and livelock.

In the beginning of verification, the design undergoes *random testing* in UVM. Code coverage and functional coverage serve as indicators of the test sufficiency in this process, which has illustrated in Section III-A. However, using OpenPiton's NoC router as an example, random testing only achieves approximately 95% code coverage. The remaining 5% of coverage, equating to about 40 points, can only be addressed by manually written specific test cases one by one. Verifiers should thoroughly analyze the router's logic, and then write constraints or packet transaction sequences to cover these points. *Achieving coverage closure with manual test cases is the most labor-intensive part in NoC verification. In this article, we propose to automate this task with hardware fuzzing.*

IV. Fuzzing for NoC Verification

We advocate that hardware fuzzing development should put emphasis on its industrial practicality and its demonstrable

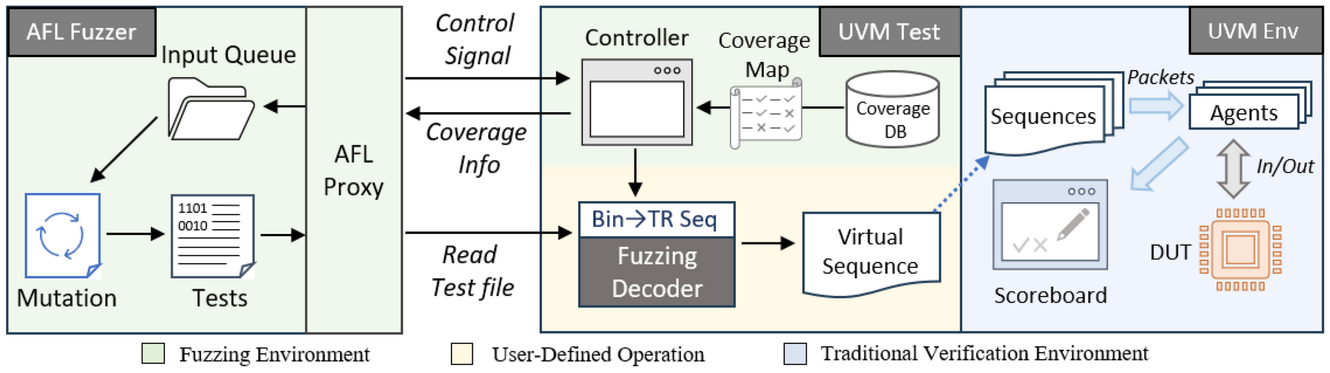


Fig. 4. Overview on UVM fuzzing framework for NoC verification.

ability to reduce human resource expenditure in the verification process. To successfully integrate fuzzing within our NoC industrial verification workflow, we put forth three fundamental prerequisites for our fuzzing methods.

1) Aimed at standard HDL (e.g., Verilog).
2) Primarily target at traditional coverage metrics.
3) Be compatible with UVM.

In this section, we outline the fuzzing framework and introduce its application into NoC verification. We still use OpenPiton NoC as a consistent running example to explain some key technical concepts. We first discuss the integration of fuzzing with the traditional NoC UVM testbench. Then, based on the NoC testplan, we analyze the potential areas in traditional NoC verification where fuzzing can enhance automation. In the final section, we explicate our approach to translate AFL generated input file into NoC stimuli.

A. UVM Fuzzing Framework

Fig. 4 provides an overview of our fuzzing framework, which consists of two parts: 1) the AFL fuzzer and 2) the simulation environment. These two components operate concurrently and interact with each other.

AFL generates binary-format input test files, and all its mutation operations are binary-based. It generates new inputs from the seed queue according to the coverage information. To facilitate communication with the simulation environment, we use *AFL proxy*, as proposed in JQF [40]. When AFL produces a new test file, the proxy sends a control signal to the simulation environment via a Linux pipe and writes the new test files to local Linux files. After each simulation loop, the simulation environment sends the coverage information back to AFL through the Linux pipe, allowing AFL to analyze the coverage and generate new inputs.

We utilize Synopsys VCS [41] to simulate the UVM testbench. A UVM test component is launched from the top testbench and connects to AFL through the AFL proxy. A controller orchestrates the basic logic of the fuzzing loop. Upon receiving a new input from AFL, the controller initiates the fuzzing decoder, which reads the input test file and converts the binary string into a valid virtual sequence. The decoding is enabled by user-defined *hardware fuzzing grammar*, which will be detailed in Section IV-C. Virtual sequence comprises multiple sequences, each of which is a series of UVM transactions. The virtual sequence also specifies the allocation of each sequence to different agents. Subsequently, the virtual sequencer allocates the sequences, resets the DUT, and the agents drive the transactions to the DUT. After each simulation iteration, we extract the coverage data from the coverage database using the unified coverage API (UCAPI) provided by VCS and send this information back to AFL.

While our fuzzing framework automates test generation, it does not eliminate the need for the creation of a basic UVM testbench. Verifiers must also develop a detailed test plan that accurately reflects the verification intent. Following this, verifiers should establish the hardware fuzzing grammar, taking into account DUT characteristics and interface protocols. The remaining fuzzing environments can then be reused for various DUTs and test plans.

B. Where Can Fuzzing Be Used in NoC Verification?

The verification process for NoC can be divided into three parts: 1) router verification; 2) network verification; and 3) stress testing. The fuzzing verification workflow discussed in this section are intuitively depicted in Fig. 5. For router and network verification, *Coverage Directed Fuzzing* helps achieve code and functional coverage closure. For stress testing, we introduce *Overtime Directed Fuzzing* to rapidly detect time-related vulnerabilities of NoC.

1) Router Verification: Achieving full code and functional coverage of router is needed in this stage. Unfortunately, random testing usually fails to reach numerous complex branches and functional points and writing test cases manually is both time-consuming and labor-intensive.

Code coverage, which measures the extent RTL code has been scrutinized, offers fine-grained, strongly correlated feedback and is ideal for directing fuzzing. There exists a strong logical correlation between various code elements. For instance, in branch coverage, mutating an input covered a particular branch is more likely to lead to an input that hits its subbranch. Therefore, mutation-based fuzzing appears to be a potent strategy for achieving closure of code coverage.

Functional coverage, as defined by verification experts, reflect the high-level functional intentions of the design. Achieving full coverage in functional coverage is particularly

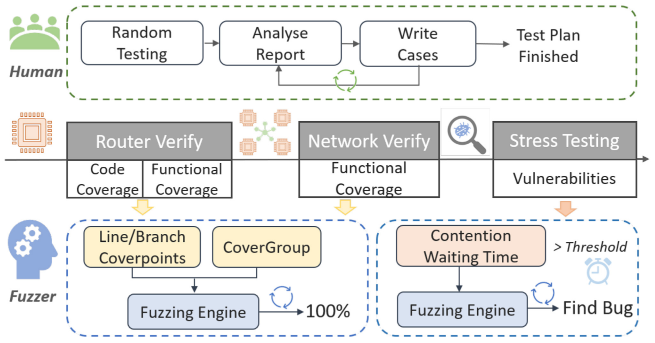


Fig. 5. Comparing manual testing and fuzzing in NoC verification.

challenging for some covergroups. For example, the covergroups of OpenPiton NoC, as outlined in Section III-A, reflect the NoC's diverse concurrent behaviors. The scale of these covergroups is substantial, making them hard to be fully covered by random testing. There exists a strong interrelation among these functional points, which makes mutation-based fuzzing potentially beneficial. However, previous fuzzing research has not focused on utilizing fuzzing to automate the functional coverage closure. In our research, we initiate a pioneering effort.

2) Network Verification: Once achieving a thorough verification of individual routers, they are interconnected for network-level testing. Given that full code coverage for each router has been achieved in the preliminary phase, it is not included in the test plan for the network, which is constructed from these routers. At the network level, the verifier establishes numerous functional coverpoints. These can be utilized as feedback in hardware fuzzing, aiming to achieve coverage closure automatically.

3) Stress Testing: This stage involves long-duration, high-pressure testing aimed at uncovering hidden vulnerabilities in NoC. Bugs in NoC typically arise from mishandled resource contention and often manifest as packet transmission delays. Generally, a timeout threshold is set. If a packet is not received within this threshold, a bug is triggered. However, triggering these bugs is not only reliant on input stress but also profoundly influenced by the order of input sequences. Using random testing is hard to find out such vulnerabilities. We hypothesize that by employing the packet waiting time, which could be induced by contention, as a guiding feedback for fuzzing, the mutation could progressively generate specific input sequences that lead to timeouts, thereby rapidly find the bug. This hypothesis leads us to introduce *OverTime Directed Fuzzing*, a method for detecting time-related vulnerabilities. This will be further discussed in Section VI-C.

C. Multiport Fuzzing Grammar Design

In each fuzzing iteration, to translate AFL-generated binary-format test files into valid input transaction sequences to DUT, a mapping strategy is necessary. In NoC verification, we construct the *NoC fuzzing grammar*. Our NoC fuzzing grammar first segments the AFL-generated input test file into *NoC packet instructions*. Subsequently, it allocates these

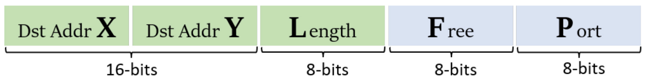


Fig. 6. NoC packet instruction.

instructions to the UVM sequences of different NoC ports. Each instruction is then translated into NoC UVM transaction and passed to the UVM driver, which generates the final stimuli for the NoC.

To accommodate the mutation operations of AFL, we format our grammar in a compact binary representation and use byte as a basic unit. As depicted in Fig. 6, NoC Packet Instructions are abstracted from a standard format of an NoC packet, that contains: 1) 16-bit destination address field; 2) 8-bit packet length field; 3) 8-bit free flag field; and 4) 8-bit port field. The destination address encapsulates x and y addresses, each being 8 bits. In random testing, the port state, either free or active, is controlled by a predefined free probability. We replace that probabilistic method with a free flag embedded in the NoC packet instruction. This allows fuzzing to generate the flag, thereby controlling the port's state directly. If the free flag is false, the testbench translate the NoC packet instruction to an NoC packet, which is then allocated to a departure port. If the free flag is true, the instruction is still allocated to a port, but it keeps the port idle for a period that corresponds to the length of packet.

The aforementioned fields are restricted to the generation of NoC packet transactions. However, both the router and network have multiple input ports, which prompts the question of how to distribute them effectively across these ports. To accommodate the multiport feature of NoC, we also introduce a port field in the NoC instruction for packet distribution. We propose two potential strategies for determining the value of this port field.

1) Dynamic Pattern: We refer to a field as dynamic when its value is generated from an AFL test file. If the port field is dynamic, the departure port of the corresponding packet is determined by the fuzzing-generated test files. The value of this field is mapped to one of all possible input port IDs, and the packet is then allocated to the input sequence of the corresponding port. For a router in Mesh topology, there are five input ports. For a network, the number of input ports is equal to the number of routers.

2) Stable Pattern: We refer to a field as stable when its value is defined by a rule, not generated from input test files. In the stable pattern, the value of port field does not depend on fuzzing. We aim to assign a fixed order to the instruction sequence for their departure. Instructions are cyclically allocated to the input queues of different ports. For instance, in a five-port router in Mesh topology, each generated instruction is dispatched sequentially to the `North`, `East`, `South`, `West`, and `Local` input ports. For a 3×3 mesh network, each generated instruction is systematically sent to the `Local` input ports of routers R_0 to R_8.

We map the AFL binary test files to each dynamic field of the instruction byte by byte, aligning them with their respective valid definition domains. Given an NoC size of $M \times N$, a maximum packet length of $L_{\max}$, and a desired port free possibility of p. The number of input ports in this verification environment is denoted as S. Consequently, the final behavior of an NoC instruction is determined by these parameters

$$\begin{cases} X_{\text{pkt}} = X_{\text{byte}} \mod M \\ Y_{\text{pkt}} = Y_{\text{byte}} \mod N \\ L_{\text{pkt}} = L_{\text{byte}} \mod L_{\max} \\ \text{idle} = F_{\text{byte}} < \text{byte\_max} * p \\ P_{\text{src}} = \begin{cases} P_{\text{byte}} \mod S, \text{ if dynamic} \\ \mathit{inst\_id} \mod S, \text{ if stable.} \end{cases} \end{cases} \tag{1}$$

With AFL binary test files evolve through successive fuzzing iterations, a variety of NoC instruction sequences are generated. These instructions can be translated into diverse NoC traffic patterns, each characterized by unique packet contents and lengths. The fuzzing process strategically guides the test generation using coverage feedback and thereby accelerates the closure of NoC testplan.

Our fuzzing framework is adaptable to any NoC topology. The heuristic algorithm of fuzzing is topology-agnostic, focusing only on the number of NoC ports. The NoC fuzzing grammar segments the AFL input test file into NoC instructions, which are then assigned to different ports in either a dynamic or stable manner. AFL observes the relationship between the coverage result and the input file, heuristically enhancing the generation of high-quality inputs for a specific topology.

In summary, to facilitate the translation of AFL-generated test files into NoC stimulus, we have designed the NoC packet instruction and developed two NoC fuzzing grammars: 1) dynamic and 2) stable. A detailed comparison between these two grammars will be presented in Section V-A.

V. Case Study: Fuzzing NoC in OpenPiton

In our case study, we use the router in OpenPiton NoC to address some key questions related to fuzzing. Section V-A contrasts the dynamic and stable port mapping strategies. In Section V-B, we further explore the treatment of functional coverpoints during fuzzing. Finally, in Section V-C, we research the influence of initial seed length to hardware fuzzing.

A. Fuzzing Grammar Comparison

Section IV-C introduces two multiport fuzzing grammars for NoC. The dynamic pattern utilizes fuzzing to determine the source input port of an NoC instruction, while the stable pattern sequentially assigns instructions to each input port within the NoC verification environment. In this section, we employ the functional covergroups of a router as a case study to draw a comparison between the stable pattern and dynamic pattern.

In these experiments, we focus on the center router in a 3×3 Mesh Network. We set the NoC maximum packet length of 20 flits and the expected idle probability to 20% for both random testing and fuzzing. We perform each trial ten times and calculate the average. We use ten initial seeds, which is generated randomly and is 100 bytes long. For the router with five input ports, this corresponds to an average of 5 input instructions for stable pattern fuzzing and 4 input instructions for dynamic pattern fuzzing per port within a fuzzing loop. We allot 10 000 000 packets for each router port, generated either randomly or by fuzzing, and then compare the resulting functional coverage. The results are presented in Table II.

We observe that fuzzing with dynamic fuzzing covers significantly fewer points than fuzzing with stable pattern, and even underperforms random testing. We attribute this result to the *dynamism* nature of dynamic pattern. A minor modification in the port field of dynamic pattern's instruction triggers a substantial transformation in associated port packet sequences, leading to a complete shift in the router's behavior.

Fig. 7 provides a representative example. Before mutation, in the dynamic pattern, the north input port receives instructions `1`, `2`, and `5` and the south input port receives instructions `3` and `4`. In the stable pattern, the port field is disregarded and instructions are distributed cyclically. After mutation, the port field of the first instruction changes from North to South. In the dynamic pattern, instruction `1` is then sent to the south. The instructions following instruction `1` in the north port all shift forward one place, and the instructions in the south port all move back one space, resulting in a reordering of all input packets to the north and south ports. However, the evolutionary algorithm assumes that mutations derived from previous inputs will explore the uncovered regions of DUT incrementally. Otherwise, it could degenerate into randomness, or even worse. Contrary to dynamic pattern, a mutation operation in stable pattern affects only a single input packet, leading to relatively minimal changes to DUT, which makes fuzzing work. *Based on this comparison, we will employ the stable pattern in the following experiments.*

TABLE II
Performance Comparison Between Dynamic and Stable Fuzzing Grammar on Router Covergroups

Metric	Coverage Achieved After 10,000,000 Packets		
	covergroup1	*covergroup2*	*covergroup3*
Dynamic Fuzz	63.5%	49.3%	92.0%
Stable Fuzz	99.8%	88.9%	100%
Random Test	80.3%	78.2%	99.8%

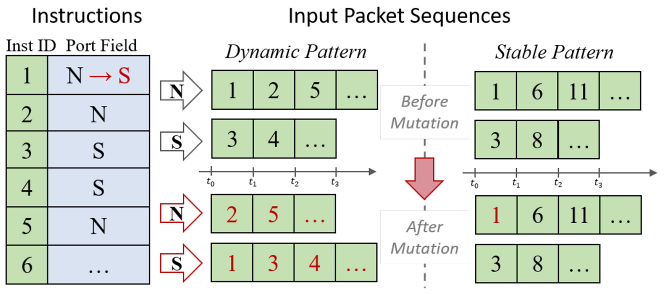


Fig. 7. Comparison of two fuzzing grammars. Dynamic pattern's high dynamism results in huge behavior changes of DUT.

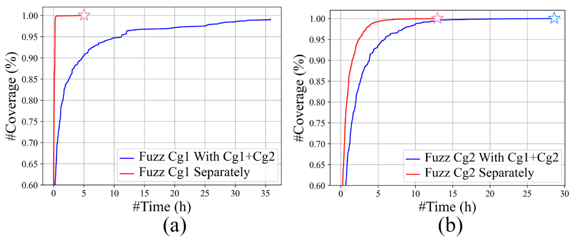


Fig. 8. Comparison of different fuzzing targets for functional coverage fuzzing. (a) Router covergroup1. (b) Router covergroup2.

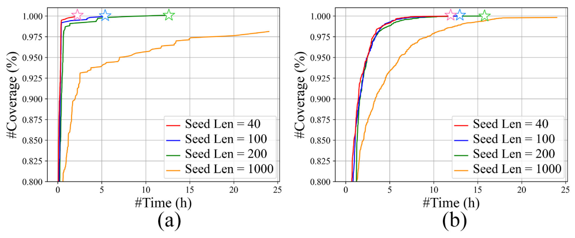


Fig. 9. Comparison of different seed length for fuzzing. (a) Router covergroup1. (b) Router covergroup2.

Key Insight: Fuzzing Grammar's *dynamism* needs restriction. Instruction mutations must not trigger drastic, untrackable changes in DUT's behavior.

B. Functional Coverage Fuzzing

Our research represents a pioneering effort in using hardware fuzzing techniques to automate functional coverage closure. When we select large-scale covergroups as our target, a question arises: How should we use covergroups as feedback in fuzzing. Should we dedicate a specific fuzzing task to each covergroup, or can we merge different covergroups together and attempt to cover them within a single fuzzing task.

An intuitive notion is that a covergroup represents a space where each coverpoint is strongly related to the others, while the coverpoints in different covergroups may vary significantly. However, combining multiple covergroups could enhance the diversity of feedback to fuzzing. We cannot rule out the possibility that different covergroups might stimulate the generation of more efficient test cases when interacting with each other.

So we use `router covergroup1` and `covergroup2` as a case study. We first fuzz `covergroup1` and `covergroup2` separately in two fuzzing tasks, where each task uses only the corresponding covergroup information as feedback. Then we combine them in a single fuzzing task and use both covergroups as feedback. The experimental setup remains consistent with previous settings outlined in Section V-A. For each methods, if one thirds of the fuzzing trials achieve coverage closure, an end flag is plotted.

Fig. 8 presents the results. The results indicate that merging covergroups within a fuzzing task significantly diminishes the task's pertinence, thereby slowing down the coverage closure time. This effect is particularly pronounced when a smaller covergroup is fuzzed alongside a larger one. *Based on this comparison, we will fuzz each covergroup separately in subsequent experiments.*

Key Insight: In a fuzzing task, targets must be closely related. Otherwise, fuzzing becomes irrelevant and less efficient.

C. Seed Length Selection

Previous Hardware fuzzing studies have often neglected the impact of initial seed length on the fuzzing process. The initial seed length not only determines the length of subsequent inputs in fuzzing iterations but also influences the simulation time, fuzzing iteration frequency, and the time required for fuzzing operations, including mutation and coverage collection.

Initial seed with shorter length can decrease the simulation time, thereby increasing the frequency of fuzzing iterations. This results in more time spent on fuzzing operations, and reduces the total length of tested input within a specific period. Despite this, the simplicity of mutation with shorter seeds may enhance evolutionary outcomes. In contrast, longer seed provide more tested input. However, this also broadens the scope and complexity of mutation targets, which could pose a challenge for the software fuzzer to achieve effective evolution. In this section, we conduct an experiment to investigate the influence of seed length on our NoC fuzzing framework.

The coverage of some router branches is related to its current state. For example, for the round-robin arbitration of the router, the grant to the next packet depends on the current grant states of the output port. And for some functional covergroup, it is related with the current buffer usage. Therefore, it is infeasible to assign just one instruction for an input port in a fuzzing loop to achieve coverage closure. Consequently, we select the minimal seed length of 40 bytes, equivalent to two instructions for an input port in a fuzzing loop. For comparison, we also set seed lengths of 100, 200, and 1000 bytes. The experimental setup remains consistent with that in Section V-A.

Fig. 9 illustrates the results. For both `router covergroup1` and `covergroup2`, a seed length of 40 bytes yields the best performance. As the seed length increases, the time required to achieve coverage closure slows down. For the longest seeds of 1000 bytes, both `covergroup1` and `covergroup2` fail to reach closure even after 24 h. The experimental data suggests a strong correlation between seed length and the efficacy of fuzzing. It is clear that the selection of initial seeds cannot be arbitrary. Our study indicates that a smaller seed length enhances NoC fuzzing performance. *Based on this comparison, we will choose small seed length in subsequent experiments.*

Key Insight: Keep fuzzing seeds small to facilitate easier evolution. However, ensure they are sufficiently long to represent the behavior space of DUT.

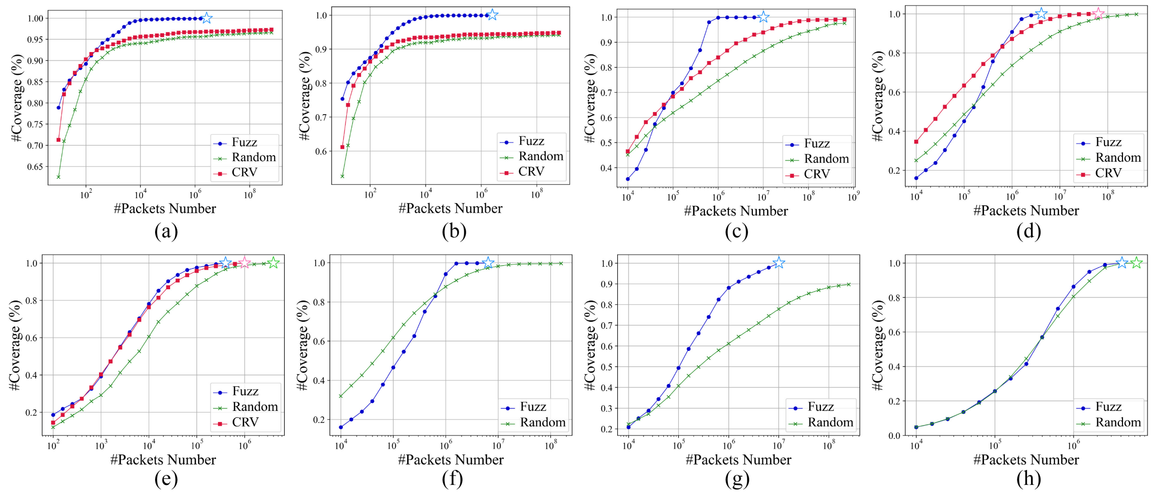


Fig. 10. Comparison between fuzzing, CRV, and random testing on router coverage closure (a)–(e) and network coverage closure (f)–(h). (a) Router line coverage. (b) Router branch coverage. (c) Router covergroup1. (d) Router covergroup2. (e) Router covergroup3. (f) Mesh covergroup1. (g) Mesh covergroup2. (h) Mesh covergroup3.

VI. Evaluation

In the evaluation part, we explore the practical efficiency of hardware fuzzing in the verification flow of NoC. We address the question: *Can fuzzing be utilized to reduce human or computational costs in a UVM environment?*

Despite our best efforts to use previous fuzzing research as a baseline, we found no suitable candidate that could be incorporated into our current VCS and UVM fuzzing framework. Most of hardware fuzzing researches either use Verilator [19] or are based on FIRRTL tool chains [18], [20], [21], [22]. For example, RFuzz [18] uses mux coverage as fuzzing guidance and targets Chisel designs. However, VCS does not support such netlist-level coverage collection. Moreover, integrating our Verilog NoC design into their fuzzing framework is currently impractical due to the difficulty in translating Verilog to FIRRTL.

Our primary objective is to evaluate whether fuzzing can automate an industrial NoC verification workflow, rather than comparing with novel but impractical fuzzing methods. Accordingly, we benchmark against commonly used industrial verification techniques, specifically, random testing and CRV methods. If random testing fails to achieve coverage closure, but fuzzing succeeds, then the human effort involved in writing test cases or specific constraints to reach coverage closure is effectively saved.

We set the experiment based on the key insights of Section V. First, we employ the stable pattern among the two NoC instruction distribution strategies. Second, we initiate a separate fuzzing task for each code coverage and functional covergroup. Third, we use the shortest possible initial seed length for router fuzzing and network fuzzing.

A. Router Fuzzing

In this section, we use a single router as DUT and employ the router UVM testbench mentioned in Section III-B. We make a comparison among fuzzing, random testing, and CRV. Our goal is to achieve coverage closure automatically through fuzzing and thereby reduce human effort. In fuzzing, We use line coverage, branch coverage, and router covergroups as feedback, which are detailed in Table I. In random testing, the input packets for each port are completely random, both in terms of seed length and destination. In CRV, we take into account the dimension-order routing strategy of the router, and establish a general constraint for each packet's destination field. This helps balance the load and maximizing the coverage within the router's logic regions.

We focus on the center router in a 3 × 3 Mesh Network. The initial seed files used for router fuzzing are 40 bytes in length, corresponding to two instructions for each of the five input ports on the router. For tests that do not achieve coverage closure, we set the maximum testing time to 120 h. Given the lengthy duration of the fuzzing process and the incrementally slower progress in coverage, we use the logarithm of the packet number as the *x*-axis to clearly illustrate the quality of input generated by different methods. This is depicted in Fig. 10, where an end flag is marked once over a third of trials reach full coverage. Additionally, Table III compares the time efficiency of different methods.

Fig. 10(a)–(e) and Table III show the router fuzzing results. For both line and branch coverage, fuzzing reaches full coverage within 3 h, surpassing the performance of random and CRV methods, which struggle to improve coverage beyond 95%. For `router covergroup1` and `covergroup2`, fuzzing can also outperform random and CRV methods in terms of both coverage and time efficiency. However, for the easy `covergroup3`, CRV demonstrates superior performance, achieving coverage closure in the shortest time.

B. Network Fuzzing

In this section, we use a 3 × 3 Mesh Network as DUT and employ the network UVM testbench mentioned in Section III-B. In network verification, our objective is to achieve functional coverage closure. Given the complexity of the NoC network, it is challenging to identify a general and effective constraint in CRV similar to the constraint outlined

TABLE III
TEST GENERATION METHODS EFFECTIVENESS COMPARISON

Method	Result	Router					Mesh Network		
		Line	Branch	Covergroup1	Covergroup2	Covergroup3	Covergroup1	Covergroup2	Covergroup3
Random	*Time (h)*	120	120	120	120	0.55	120	120	**0.33**
	Cov (%)	96.7	94.8	97.3	98.6	*max*	99.8	89.7	***max***
CRV	*Time (h)*	120	120	120	23.6	**0.17**	-	-	-
	Cov (%)	97.3	95.2	98.8	*max*	***max***	-	-	-
Fuzz	*Time (h)*	**2.6**	**2.4**	**3.3**	**10.8**	1.0	**9.2**	**6.8**	5.1
	Cov (%)	***max***	***max***	***max***	***max***	*max*	***max***	***max***	*max*

in Section VI-A for router verification. We only use fuzzing and random testing for packet generation.

For a 3 × 3 Mesh Network with nine input ports, we set the initial seed length to 72 bytes, which corresponds to two instructions for each local input port. Fig. 10(f)–(h) and Table III show the experiment results. For both `mesh covergroup1` and `covergroup2`, fuzzing achieves coverage closure while random testing fails to complete. However, for the less complex `covergroup3`, although fuzzing is capable of reaching closure, it is slower than random testing.

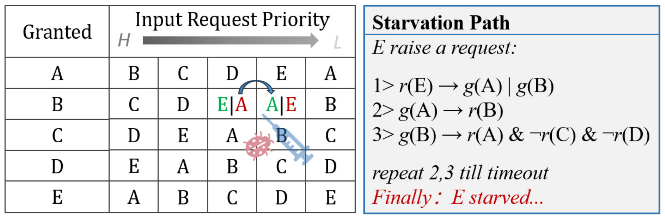
C. Fuzzing for Vulnerability Detection

Using random testing to identify vulnerabilities is challenging during stress testing in NoC verification. Therefore, we adopt *overtime directed fuzzing*, which utilizes packet waiting time as fuzzing feedback, to generate effective NoC packet sequences and uncover potential overtime-related bugs.

We have manually inserted a starvation bug into the router's arbitration logic. The arbiter in OpenPiton's router employs a starvation-free round-robin arbitration strategy. Every port will be granted the highest priority cyclically. Fig. 11 provides the priority table and the details of the injected bug. When an output port is assigned to input port `A`, the next highest priority input request in the subsequent arbitration is given to input port `B`. Similarly, when the output port is granted to `B`, the following highest priority input request is directed to `C`, and so on. We inject the bug by switching the priorities of input ports `A` and `E` when the output port is allocated to `B`. This could cause the starvation of `E` under certain conditions.

Fig. 11 also depicts the path that can trigger the bug. Here, $g(X)$ represents the output port is being granted to input port X. $r(X)$ represents input port X is requesting for the output port. To trigger the bug, we must avoid granting the output port to `C` or `D`, as this would allow `E` to get high priority and be granted, causing the bug to be missed. Therefore, when `E` makes a request, the output port must be granted to either input port `A` or `B`. If it is granted to `A`, then `B` should be active, ensuring that the next arbitration grants the output port to `B`. If it is granted to `B`, the following arbitration must grant the output port to `A`, requiring ports `C` and `D` to be inactive. This implies that the input sequences from different ports must always meet complex conditions over an extended period, long enough to surpass the overtime threshold, which is quite challenging for random testing.

In our experiment, we compare random testing and overtime directed fuzzing. To evaluate the necessity of our time-based



Granted	Input Request Priority H → L				
A	B	C	D	E	A
B	C	D	E|A	A|E	B
C	D	E	A	B	C
D	E	A	B	C	D
E	A	B	C	D	E

Starvation Path
E raise a request:
1> $r(E) \rightarrow g(A) \mid g(B)$
2> $g(A) \rightarrow r(B)$
3> $g(B) \rightarrow r(A)$ & $\neg r(C)$ & $\neg r(D)$
repeat 2,3 till timeout
Finally : E starved...

Fig. 11. Injected starvation bug and starvation path.

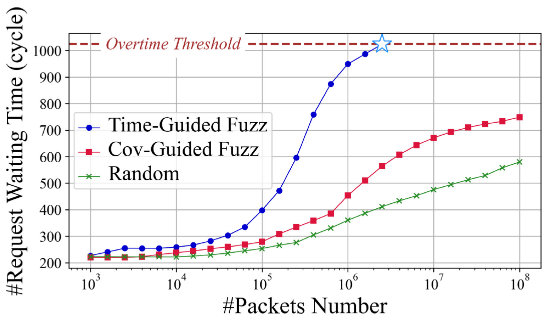


Fig. 12. Comparison between random testing and fuzzing on starvation bug detection.

fuzzing approach, we also incorporate line-coverage-guided fuzzing for a comprehensive comparison. We use the router fuzzing environment and set the overtime threshold of 1024 clock cycles, approximately five times the normal waiting time. To ensure the input sequences are long enough to trigger this threshold, we set initial seed file length to 100 bytes, equivalent to five instructions for each input port.

Fig. 12 illustrates the maximum waiting time during verification with different methods. Despite running for 12 h, random testing could not detect the bug. In contrast, overtime directed fuzzing could identify the bug in less than half an hour. We also demonstrate that using time-based feedback is essential for uncovering overtime-related vulnerabilities. On the other hand, using coverage as feedback cannot provide relevant guidance for such bugs and does not speed up the bug-finding process.

The experimental results in Section VI-A–VI-C show that for verification targets inaccessible through random testing or CRV in NoC verification, fuzzing can achieve them automatically. With the analysis of coverage or overtime feedback, fuzzing can generate test inputs of higher quality. This could reduce the human labor required for writing high-quality inputs and automate the NoC verification process.

VII. Discussion

A. *Formal Verification of NoC*

Our research is an initial attempt to automate NoC verification at the RTL level. We also attempts using SymbiYosys [42], a popular open-source formal verification tool, to verify the starvation bugs detailed in Section VI-C. However, the formal tool cannot automatically solve that property within 24 h. We encountered a state explosion due to the time depth involved. It is important to note that key properties in NoC like starvation and deadlock are all liveness properties. For a router, it may take dozens of clock cycles to send or receive a packet, and potentially even longer if waiting is required. This necessitates that formal methods induct for hundreds of clock cycle steps to trigger the bugs, leading to a significant time depth and resulting in a state explosion that cannot be effectively addressed by formal methods.

Generally, formal methods are employed at an abstract modeling level to verify the correctness of NoC algorithms, such as routing and arbitration algorithms [5], [6], [7], [8], [9]. These formal methods cannot be directly applied to the RTL level due to state explosion. Consequently, at the RTL level, simulation-based methods, such as fuzzing, are indispensable for bug detection. Despite formal methods can guarantee an algorithm's correctness, they cannot ensure that the algorithm is accurately implemented at the RTL level. Thorough testing at the RTL level is also necessary. In this context, NoC fuzzing can serve as an approach to automate the NoC verification.

B. *Time–Cost Distribution in Hardware Fuzzing*

The execution speed of fuzzing is considerably slower than random testing. When testing `router covergroup1` with 800 points, fuzzing's execution speed is approximately twice as slow as that of random testing. This discrepancy becomes more pronounced with larger coverage number.

We have examined the time expenses associated with each phase in a fuzzing process. Fig. 13 provides a visual representation of the time distribution for each part. We use an initial seed length of 100 bytes when evaluating the influence of different coverage sizes on fuzzing in the first histogram. We set coverage size of 1024 when comparing the impact of varying seed lengths on fuzzing in the second histogram.

The first histogram underscores a scalability issue related to the size of coverage during fuzzing. The collection time for coverage increases substantially with the growth in coverage size, thereby impeding the efficiency of fuzzing. This is primarily due to the slow speed of VCS's coverage collection mechanism. Currently, when fuzzing requests coverage data, it instructs VCS to dump the coverage into its database in a structured and hierarchical manner. Subsequently, fuzzing retrieves the data through UCAPI. However, fuzzing only requires a coverage vector reflecting the hit count of each coverpoint, making current process of extracting a detailed report inefficient. The scalability issue could be addressed if the simulator provides a direct coverage interface for fuzzing. This solution is beyond the scope of this article and is earmarked for future work.

As depicted in the second histogram, a longer initial seed length results in increased simulation time, potentially testing

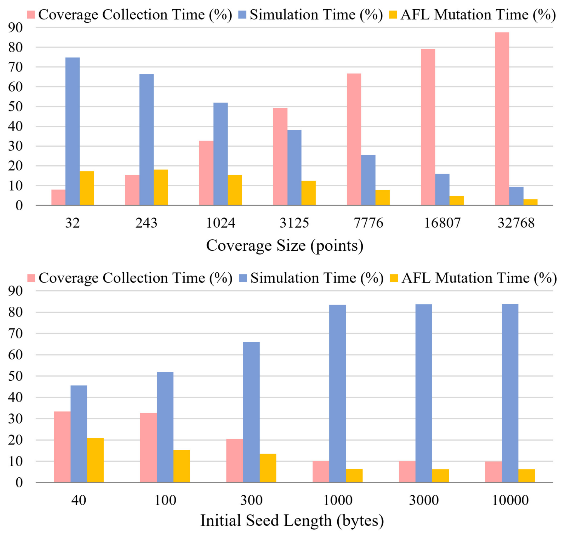


Fig. 13. Compare the time partitioning across different coverage sizes and seed lengths in router fuzzing.

more packet sequences within a single fuzzing loop. When the seed length exceeds 1000 bytes, the simulation time ceases to increase. This is attributed to the effective trimming operation of AFL fuzzer. From a time-efficiency perspective, a longer seed proves more beneficial as it lessens the number of fuzzing loops and saves on coverage collection time. However, our previous experiments have indicated that, even with the current low-efficiency coverage collection mechanism, the use of shorter seed lengths is justified. Although it increases the frequency of fuzzing loops and spends more time in coverage collection, the benefits outweigh the costs.

C. *Comparison of Software Network Fuzzing*

There are three related verification problems for fuzzing network systems: 1) verifying a single network component; 2) verifying the network system; and 3) verifying the overall functionality of network according to some application protocols. Our research on NoC primarily addresses the first two levels. We focus on the NoC packet transfer protocol and fuzz the NoC router and the network. Our research does not include the potential high-level application protocols of NoC.

Numerous fuzzing studies have been conducted to verify software network protocols (e.g., FTP [43], RTSP [43], SNMP [44], and TCP [45]). Some studies use grammar-based fuzzing [46], [47], [48]. They utilize hard-coded or user-defined grammar specifications to guide test case generation. These specifications define data structure or field types of packets to be generated. Several recent approaches use stateful protocol fuzzing [43], [49], [50]. They learn the state models of network protocols to enhance seed selection and mutation.

In our research, we adapt the principles of grammar-based fuzzing to NoC verification. We define an NoC-specific grammar to generate a diverse array of NoC packets as stimuli. Given the simplicity of NoC packet transfer protocols, which are hardware-limited and state-light compared to software protocols, we have not incorporated a state model. Our primary verification challenge lies in ensuring RTL correctness, which is considerably more complex and prone to errors compared to software coding. Once we have tested the basic functionality of

NoC packet transfer, to verify the overall functionality of NoC, including its high-level application protocols, some advanced software network fuzzing techniques, such as state model, could provide valuable insights. This could be considered as a promising avenue for our future research.

D. Future Work

In Section V, we present three key insights about fuzzing, derived from a small number of experiments with a router as our DUT. We recognize that these conclusions are lack the support of extensive experimental evidence. Conducting more experiments with a greater variety of DUTs and fuzzing targets in the future could help solidify these findings.

In our fuzzing framework, we use AFL as a black box, employing it as the primary mutation engine. However, upon closer examination, we observe that not all mutation mechanisms in AFL are beneficial for our NoC fuzzing. Our experiments find that the majority of beneficial mutations are generated by AFL's havoc and splice operations, contributing to 47% and 53% of all useful mutations, respectively. This observation implies a redundancy in most of AFL's operations. Therefore, future research could focus on understanding the reasons behind this redundancy and developing a more efficient mutation engine specifically tailored for fuzzing hardware designs.

At present, our efforts are primarily focused on automating the industrial verification of NoC. It is necessary to extend the applicability of our method to a wider range of DUTs. Regrettably, there are no open-source or commercial tools that enable the implementation of hardware fuzzing for industry-standard IC verification workflows. Therefore, further development of a comprehensive hardware fuzzing tool oriented toward UVM and general hardware designs is an essential pursuit.

VIII. Conclusion

In this article, we show how fuzzing methods can be used to automate the industrial NoC verification process. Specifically, we developed a hardware fuzzing framework that integrates the widely used software fuzzer AFL into NoC UVM testbench. Our results suggest that fuzzing can greatly reduce manpower demands and accelerate the verification process. For NoCs, we have devised a unique hardware fuzzing grammar for multiport DUTs. We utilized fuzzing at various stages of NoC verification, including router and network verification, as well as vulnerability detection. Our fuzzing methodology was evaluated through the case study of the OpenPiton NoC. The evaluation results show that our approach can automatically achieve complete line and branch coverage and complete coverage closure of some complex functional covergroups. Fuzzing was also successful in uncovering hidden starvation bugs, demonstrating its superiority over random testing.

References

[1] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surveys*, vol. 38, no. 1, pp. 1–50, 2006.

[2] A. V. Bhaskar and T. Venkatesh, "Performance analysis of network-on-chip in many-core processors," *J. Parallel Distrib. Comput.*, vol. 147, pp. 196–208, Jan. 2021.

[3] N. A. Kumar, A. Kavitha, P. Venkatramana, and D. Nandan, "Architecture design: Network-on-chip," in *VLSI Architecture for Signal, Speech, and Image Processing*. Apple Acad. Press, Palm Bay, FL, USA, 2022, pp. 147–165.

[4] J. Ahn et al., "Network-on-chip microarchitecture-based covert channel in GPUs," in *Proc. Int. Symp. Microarchit.*, 2021, pp. 565–577.

[5] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz, "A generic model for formally verifying NoC communication architectures: A case study," in *Proc. 1st Int. Symp. Netw.-Chip*, 2007, pp. 127–136.

[6] Y.-R. Chen, W.-T. Su, P.-A. Hsiung, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen, "Formal modeling and verification for network-on-chip," in *Proc. Int. Conf. Green Circuits Syst.*, 2010, pp. 299–304.

[7] V. A. Palaniveloo and A. Sowmya, "Application of formal methods for system-level verification of network on chip," in *Proc. Comput. Soc. Annu. Symp. VLSI*, 2011, pp. 162–169.

[8] A. Zaman and O. Hasan, "Formal verification of circuit-switched network on chip (NoC) architectures using SPIN," in *Proc. Int. Symp. Syst.-Chip (SoC)*, 2014, pp. 1–8.

[9] F. Boutekkouk, "Formal specification and verification of communication in network-on-chip: An overview," *Int. J. Recent Contrib. Eng., Sci. IT*, vol. 6, no. 4, pp. 15–31, 2018.

[10] A. S. Eissa et al., "A reusable verification environment for NoC platforms using UVM," in *Proc. Int. Conf. Smart Technol.*, 2017, pp. 239–242.

[11] F. Vitullo et al., "A reusable coverage-driven verification environment for network-on-chip communication in embedded system platforms," in *Proc. Workshop Intell. Solu. Embedded Syst.*, 2009, pp. 71–77.

[12] S. Saponara, L. Fanucci, and M. Coppola, "Design and coverage-driven verification of a novel network-interface IP macrocell for network-on-chip interconnects," *Microprocess. Microsyst.*, vol. 35, no. 6, pp. 579–592, 2011.

[13] J. Bromley, "If systemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language," in *Proc. Forum Specification Design Lang.*, 2013, pp. 1–7.

[14] Y.-N. Yun, J.-B. Kim, N.-D. Kim, and B. Min, "Beyond UVM for practical SoC verification," in *Proc. Int. SoC Design Conf.*, 2011, pp. 158–162.

[15] N. Harshitha, Y. P. Kumar, and M. Kurian, "An introduction to universal verification methodology for the digital design of integrated circuits (IC's): A review," in *Proc. Int. Conf. Artif. Intell. Smart Syst.*, 2021, pp. 1710–1713.

[16] J. Wang, N. Tan, Y. Zhou, T. Li, and J. Xia, "A UVM verification platform for RISC-V SoC from module to system level," in *Proc. Int. Conf. Integr. Circuits Microsyst.*, 2020, pp. 242–246.

[17] W. Ni and J. Zhang, "Research of reusability based on UVM verification," in *Proc. Int. Conf. ASIC*, 2015, pp. 1–4.

[18] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*, 2018, pp. 1–8.

[19] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *Proc. USENIX Security Symp.*, 2022, pp. 3237–3254.

[20] 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 *Proc. Design Autom. Conf.*, 2021, pp. 529–534.

[21] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential fuzz testing to find CPU bugs," in *Proc. Symp. Security Privacy*, 2021, pp. 1286–1303.

[22] B. Fajardo, K. Laeufer, J. Bachrach, and K. Sen, "RTLFUZZLAB: Building a modular open-source hardware fuzzing framework," in *Proc. Workshop Open-Source EDA Technol.*, 2021, pp. 1–11.

[23] D.-L. Lin, Y. Zhang, H. Ren, B. Khailany, S.-H. Wang, and T.-W. Huang, "GenFuzz: GPU-accelerated hardware fuzzing using genetic algorithm with multiple inputs," in *Proc. Design Autom. Conf.*, 2023, pp. 1–6.

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

[25] C. Chen et al., "HyPFuzz: Formal-assisted processor fuzzing," 2023, *arXiv:2304.02485*.

[26] S. Canakci et al., "ProcessorFuzz: Processor fuzzing with control and status registers guidance," in *Proc. Int. Symp. Hardw. Oriented Security Trust*, 2023, pp. 1–12.

[27] "LibFuzzer: A library for coverage-guided fuzz testing," 2018. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[28] M. Zalewski, "American fuzzy lop (2.52b)." 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[29] W. Snyder, "Verilator," 2024. [Online]. Available: https://www.veripool.org/wiki/verilator

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

[31] A. B. Mehta and A. B. Mehta, "Constrained random verification (CRV)," in *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technol. Methodologies*. Cham, Switzerland: Springer, 2018, pp. 65–74.

[32] R. Ma, H. Zhao, J. Huang, S. Zhang, and G. Luo, "An endeavor to industrialize hardware fuzzing: Automating NoC verification in UVM," in *Proc. Design, Autom. Test Europe Conf. Exhibit.*, 2024, pp. 1–2.

[33] *OpenSPARC T1 Microarchitecture Specification*, Sun Microsyst., Inc., Santa Clara, CA, Tech. Rep., 2006.

[34] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Amsterdam, The Netherlands: Elsevier, 2004.

[35] J. Zhao, A. Agrawal, B. Nikolić, and K. Asanović, "Constellation: An open-source SoC-capable NoC generator," in *Proc. 15th IEEE/ACM Int. Workshop Netw. Chip Archit. (NoCArc)*, 2022, pp. 1–7.

[36] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis, and J. Shalf, "OpenSoC fabric: On-chip network generator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2016, pp. 194–203.

[37] M. Braun, J. Pendlum, and M. Ettus, "RFNoC: RF network-on-chip," in *Proc. GNU Radio Conf.*, vol. 1, 2016, pp. 1–7.

[38] P. James, *Verification Plans: The Five-Day Verification Strategy for Modern Hardware Verification Languages*. New York, NY, USA: Springer, 2011.

[39] M. Mintz and R. Ekendahl, *Hardware Verification With System Verilog: An Object-Oriented Framework* (International Federation for Information Processing). New York, NY, USA: Springer, 2007.

[40] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-guided property-based testing in Java," in *Proc. SIGSOFT Int. Symp. Softw. Test. Anal.*, 2019, pp. 398–401.

[41] "VCS: The industry's highest performance simulation solution." 2024. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[42] S. Eda, "SymbiYosys (SBY) documentation," 2023. [Online]. Available: https://symbiyosys.readthedocs.io/en/latest/index.html

[43] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *Proc. IEEE 13th Int. Conf. Softw. Testing, Validation Verification (ICST)*, 2020, pp. 460–465.

[44] Z. Wang, Y. Zhang, and Q. Liu, "RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing," *KSII Trans. Internet Inf. Syst.*, vol. 7, no. 8, pp. 1989–2009, 2013.

[45] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 21)*, 2021, pp. 489–502.

[46] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, Oct. 2016, pp. 1492–1504.

[47] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ICS protocol," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5S, pp. 1–22, Oct. 2019.

[48] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "ICS protocol fuzzing: Coverage guided packet crack and generation," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[49] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *Proc. USENIX Security Symp.*, Jan. 2020, pp. 2523–2540.

[50] J. Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proc. USENIX Security Symp.*, Aug. 2015, pp. 1–14.