CHIMERA: Fuzzing P4 Network Infrastructure for Multi-Plane Bug Detection and Vulnerability Discovery

Jiwon Kim
Purdue University
kim1685@purdue.edu

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

Benjamin E. Ujcich
Georgetown University
bu31@georgetown.edu

***Abstract*—Programmable network data planes, such as P4, offer flexibility in defining network forwarding behavior. However, such programmability introduces a new attack surface for bugs and security vulnerabilities. Most P4 security research has focused solely on the data plane, overlooking its integration with the control plane. We investigated past bug reports in open-source P4 implementations across both control and data planes, and we observed that many P4 network bugs and vulnerabilities arise from the interplay between these planes.**

We present CHIMERA, a comprehensive P4 fuzzer that targets bugs requiring multi-plane inputs and impacts. Unlike existing network fuzzers that operate separately on each plane, CHIMERA uses concolic execution to capture control-data plane interactions. CHIMERA introduces two novel input mutation strategies to exploit interdependencies across both planes and P4 programs: parser-aware packet mutation (PAPM) and header-guided rule generation (HGRG). Evaluating CHIMERA on ONOS, Stratum, and BMv2, we discovered 7 new bugs, including 3 security-critical vulnerabilities, 2 bugs triggered by multi-plane inputs, and 2 cross-plane bugs. CHIMERA outperforms state-of-the-art single-plane fuzzers with higher coverage and a 3.5× higher bug detection rate.

1. Introduction

Software-defined networking (SDN) has changed the paradigm of designing computer network architectures, enabling more flexibility and agility compared to traditional networks. Such flexibility arises from the programmable nature of both the network's control and data planes.

Programmable data planes are increasingly deployed to enable more efficient and complex logic in line-rate forwarding. In particular, the P4 programming language [28] has enabled network operators to define how packets are processed within the data plane. Unlike prior approaches (*e.g.,* OpenFlow [64]), P4 is not tied to specific network protocols (*e.g.,* TCP/IP) or even implementation targets[1] (*e.g.,* SmartNICs [89], FPGAs [87], ASICs [32]). Developers can specify a network protocol and its processing operations within a P4 program. Given a program, a compiler generates the API that the P4-aware control plane uses to communicate with the data plane.

Despite the benefits of programmable data planes, programmability brings new security challenges. Prior work has given more attention to finding bugs in the program compilation process [20], [73], detecting semantic bugs within programs [35], [37], [60], [76], [80], [90], [95], and validating data plane switches [21], [56], [66], [72]. However, less attention has been paid to the bugs associated with *the network infrastructure*, which inherits vulnerabilities from both the control and data planes. Both planes in a P4-enabled network often change dynamically in runtime, particularly as the control plane makes changes to the data plane configuration and how the data plane processes traffic.

Overview. In this paper, we first seek to understand how known bugs and vulnerabilities occur in the broader "full-stack" P4-based network infrastructure. We posit that the programmability of the network infrastructure where the P4 programs are deployed can cause complex and interdependent challenges across both the control and data planes. We explore the dataset of publicly-known bug reports for various open-source P4 network implementations to understand (1) in which plane bugs and vulnerabilities occur; and (2) how often bugs relate to the various runtime configuration components (*i.e.,* data plane packets, control plane packets, and control plane rules[2]). We find that *over 80% of P4 network infrastructure bugs relate to control plane inputs* and discover that 20.8% of bugs in both planes can be detected only under specific configurations of control plane rules and data plane packets matched to such rules.

However, state-of-the-art bug detection tools for programmable networks often focus on a specific plane without considering multi-plane interactions. More importantly, reflecting the inter-plane dependency during fuzzing is more complex than combining a control plane fuzzer with a data plane fuzzer. For instance, existing control plane fuzzing tools [34], [46], [51], [52], [57], [58], [59] lack understanding of P4 semantics, and data plane fuzzing tools [76], [90] rely on user-provided control plane inputs. Motivated by our findings on known bugs and vulnerabilities, we posit that siloed approaches to bug and vulnerability detection that

1. In this paper, we refer to "targets" (a P4 terminology), "switches," and "forwarding devices" synonymously unless otherwise noted.

2. We refer to "control plane rules," "table entries" (a P4 terminology), "table rules," and "flow rules" (an SDN terminology) interchangeably.

existing tools use cannot easily discover complex and subtle bugs that involve interactions between the two planes.

We introduce CHIMERA, a comprehensive multi-plane fuzzing framework for P4 network bug and vulnerability discovery to address these challenges. CHIMERA tests both the control and data planes simultaneously by fuzzing packets in the data plane and rules in the control plane. To detect abnormal behaviors, CHIMERA uses concolic execution to determine the expected output and compare it against the actual target's output. CHIMERA also leverages concolic execution to identify invalid control plane rules and measure a P4 program's coverage (*e.g.,* P4 action coverage).

To further explore the interdependencies among multi-plane input spaces and P4 programs, we introduce new mutation policies for packets and rules. To allow mutated packets to fit into a state space specified in the P4 program, we devise parser-aware packet mutation (PAPM) that leverages a packet mutator directly translated from the P4 program. We propose header-guided rule generation (HGRG) to guarantee that the given input packet matches mutant rules by mapping the packet's header fields to the corresponding fields of rules. We plan to open source CHIMERA.

We evaluate CHIMERA with popular open-source P4 infrastructure implementations such as the ONOS SDN controller [25], the Stratum switch operating system [18], and the BMv2 switch [5]. We discovered 7 new bugs, 3 of which are security-critical vulnerabilities that tamper with data plane integrity and induce denial of service and resource exhaustion. Control plane inputs are essential for finding all 7 bugs, while 2 bugs can only be triggered by interdependent multi-plane inputs. Moreover, we uncovered two *cross-plane* bugs[3], which can only be detected by testing both planes as a whole. We perform an extensive performance evaluation to demonstrate the advantages of CHIMERA over two state-of-the-art programmable network fuzzing tools.

Contributions. Our contributions are the following:

- We conduct a comprehensive study of publicly known bug reports across popular P4 implementations. We find 83.3% of P4 network bugs are related to control plane rules, and 20.8% of bugs in both planes need multi-plane triggers.
- We design and implement CHIMERA, a multi-plane fuzzer for P4 networks that overcomes the limitations in existing approaches by mutating control and data plane inputs simultaneously. We leverage concolic execution of P4 programs to create a bug oracle without manually defined assertions, to verify control plane rules, and to measure P4 program coverage.
- We propose two new approaches in input mutations, which leverage interdependencies among multi-plane inputs and P4 programs: parser-aware packet mutation (PAPM) and header-guided rule generation (HGRG).
- Applying CHIMERA, we find 7 new bugs, 3 of which are security-critical vulnerabilities. All 7 bugs require control plane inputs, and 2 bugs need interdependent multi-plane inputs, including 2 *cross-plane* bugs that require fuzzing at both planes. Compared to prior approaches tackling each plane individually, CHIMERA provides the highest coverage in both the data plane (1.16×) and the control plane (1.1×), achieving a 3.5× higher bug detection rate.

3. Cross-plane bugs, which we discuss in detail in Section 8, are caused by semantic gaps between the control and data plane and demonstrate how one plane can indirectly affect another plane [83].

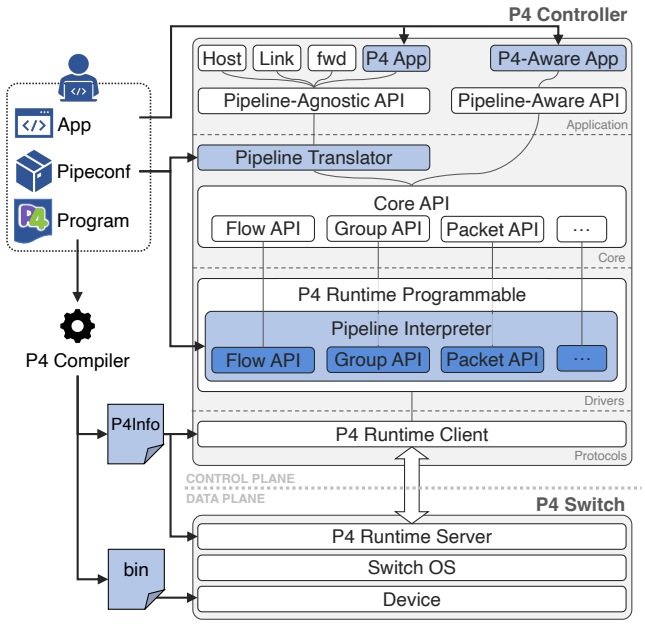


Figure 1: Overview of a P4-enabled network. Developers or operators implement the blue-colored components: a P4 program, a P4 pipeconf package, and P4-aware applications.

2. Background

Programmable networks. Software-defined networking (SDN) decouples the network into two domains: a *control plane* that decides how the network should be configured and a *data plane* that performs traffic forwarding. Unlike traditional legacy networks, SDN enables network operators to manage a complex network of switches in a centralized view through an *SDN controller*. The SDN controller and switches communicate using a *southbound API*.

OpenFlow [64], an early southbound API, specifies a standard set of header fields to *match* against incoming packets and a standard set of *actions* to take (*e.g.,* forward, drop); however, OpenFlow suffers from flexibility issues with new network features and protocols [45], [86].

P4. P4 [28] overcomes the inflexibility issues of OpenFlow by enabling programmers to define how data plane devices should process packets in a *P4 program*. A P4 program specifies three logical components: a *parser* that consists of parser states to extract header metadata, a *control* that defines match-action tables such that operators can install table entries at runtime, and a *deparser* that sends an output packet. According to specific *targets* (*e.g.,* a P4-capable switch, SmartNIC), a P4 program can have different

pipelines (*e.g.,* ingress and egress) and external functions (*e.g.,* hash functions, checksums).

A *P4 compiler* produces two outputs from P4 programs to be deployed on targets: a target-specific configuration binary (`json` and `bin`) and a communication protocol between the control and data plane (`p4info`). The P4Runtime protocol [39] supports P4 program installation and acts as the southbound API. P4 switches read the binary and process packets based on both the P4 program logic and the control plane's runtime configuration, such as table entries.

The *P4 controller*, implemented as a standalone SDN controller or subsystem within an SDN controller, communicates with P4 switches that run a P4 program. We focus on the P4Runtime protocol since it is a standard proposed and used by the P4 community [39].

Figure 1 shows a P4-enabled network using the P4Runtime within the ONOS SDN controller [25]. We choose ONOS as a representative implementation of the P4 controller since ONOS supports P4 in its latest version and has been used in carrier-grade deployment [44]. ONOS supports different P4 programs using Pipeconf, which allows existing network applications to translate ONOS's APIs into P4 APIs. ONOS Pipeconf includes an interpreter that translates control plane objects into P4 entities, such as matches, actions, and packets destined for the data plane (*i.e.,* `PACKET_OUT`). Besides ONOS, OpenDayLight [65] has a proposed plug-in to support different P4 programs, similar to ONOS Pipeconf. The security concerns that we discuss are generally applicable to all P4 implementations.

Fuzzing. Fuzzing automates bug funding at scale. *Fuzzers* generate random inputs and monitor a target program to see if such inputs trigger bugs. Fuzzers can leverage feedback from the target to generate input efficiently, such as code coverage guidance [1], [36], [93], state machine analysis [22], [23], or white-box techniques (*e.g.,* taint analysis and concolic execution) [30], [38], [54], [79], [92]. Fuzzers utilize diverse bug detection mechanisms to uncover *syntactic errors* (*e.g.,* crashes) or *semantic errors*. *Sanitizers* can detect memory errors, including stack/heap overflows, use-after-free, *etc.* [75]. For semantic errors, fuzzers can leverage user-provided assertions [31], differential testing [63], or domain-specific methods [52], [53], [96]. If fuzzing targets require a program as an input, such as compilers or architectures, fuzzers can use input programs as oracles for model-based testing [47], [73].

3. P4 Infrastructure Bugs and Vulnerabilities

While P4 enables data plane network programmability, such programmability not only brings security challenges from new components (*e.g.,* P4 programs) but also creates new attack surfaces stemming from the growing complexity of the programmable networking stack. Inputs from both the control and data planes can change the network configuration directly and indirectly, which attackers can exploit.

To systematically understand the new attack surface and its security impacts, we conduct a comprehensive exploration of publicly disclosed bug reports within several open-source programmable networking projects involving P4 (Section 3.1), revealing that bugs can impact all infrastructure components. We consider existing bug detection tools (Section 3.2) and present our threat model (Section 3.3). We analyze a security vulnerability we discovered (Section 3.4) to illuminate the limitations of existing programmable network fuzzing tools. We identify challenges that hinder efficient bug and vulnerability discovery (Section 3.5), which motivates our design of CHIMERA.

TABLE 1: Number of bug reports in P4 network infrastructure components based on triggers.

Triggers	Switch	Controller
Data plane packet	4	0
Control plane packet	1	1
Control plane rule	1	12
Control plane rule + Data plane packet	4	1

3.1. P4 Infrastructure Bug Study

To understand the space of existing bugs in the P4 network and which components of the P4-enabled network they originate from and affect, we collected all of the bug reports on open-source projects within the P4 ecosystem.

Approach. We selected five open-source projects:

1) BMv2 [5], [6], the reference implementation for the P4 software switch;
2) PI [15], [16], an implementation framework for a P4Runtime server that runs on a data plane;
3) Stratum [18], [19], a silicon-independent switch operating system that supports SDN interfaces, including P4 and OpenConfig;
4) ONOS [11], [25], a production-quality SDN controller that is the basis for proprietary network operating systems used in production environments [44]; and
5) fabric-tna [8], [9], a P4 program that realizes a network fabric tailored for 5G-connected edge clouds and is designed to run on Stratum and ONOS.

We collected 130 reports related to P4 from the projects' bug trackers [6], [9], [11], [16], [19] and filtered out those caused by misconfigurations, non-reproducible cases, and bugs caused by non-P4-related components (*e.g.,* testing, configuration, UI, *etc.*). Among the remaining 68 bugs, 41 bugs (60.3%) are triggered by packets and rules, which form the test inputs of existing P4 testing tools, whereas 27 bugs are triggered by other inputs: P4 program input (6 bugs), meter and counter (9 bugs), OpenConfig southbound APIs (5 bugs), intent interface (1 bug), and device connection (6 bugs). Of the remaining 41 bugs, 15 were caused by P4 programs, 10 were caused by the switch, 14 were caused by the controller, and 2 were caused by network applications. We excluded the 15 P4 program bugs and 2 application bugs that are logical bugs, as these require user-provided assertions that depend on semantics. 24 bugs originated from the switch and the controller and relate to implementation errors such as specification mismatch, translation errors, rule mismanagement, and crashes.

TABLE 2: Comparison of the features of state-of-the-art programmable network fuzzing tools across the data plane (**DP**) and the control plane (**CP**). Targets include the data plane switch (**S**) and the control plane controller (**C**). Coverage indicates which code base and granularity of code access the tool investigates. Code base includes **P4** programs (*e.g.,* P4 actions (**PA**)), data plane code (**DC**), and control plane code (**CC**). Code access granularity in fuzzing can be black-box (■), grey-box (◩), or white-box (□). Test inputs can be APIs to test controllers (*i.e.,* northbound API (**NB API**) and southbound API (**SB API**)) or entities for the target network (*i.e.,* packets and rules). OF stands for OpenFlow [64]. In SB API, parenthesis means that the test input is an indirect input sourced from another input (*e.g.,* control plane rules via OF or P4Runtime [39]).

	Target		Prerequisite	Coverage			Feedback	Test Input				
	S	C		P4	DC	CC		NB API	SB API	CP Rules	CP Packets	DP Packets
PAZZ [77]	✓	✗	-	N/A	■	N/A	✗	N/A	(OF)	△ (User)	✗	Mutation
P6 [76]	✓	✗	P4, Rule, Query	□	■	N/A	Bug Reward (bool)	N/A	(P4Runtime)	△ (Fixed)	✗	Mutation
FP4 [90]	✓	✗	P4, Assert	◩	■	N/A	PA, Rule	N/A	(P4Runtime)	△ (User)	✗	Mutation
DELTA [58]	✗	✓	-	N/A	N/A	■	✗	App API	OF	✗	✗	Generation
BEADS [46]	✗	✓	-	N/A	N/A	■	✗	✗	OF	✗	✗	Generation
AIM-SDN [34]	✗	✓	Service URLs	N/A	N/A	■	✗	REST	(OF)	Random	✗	✗
AudiSDN [57]	✗	✓	-	N/A	N/A	■	✗	App API	(OF)	Generation	✗	✗
Spider [59]	✗	✓	-	N/A	N/A	◩	Performance, Semantic	Event Sequences	(OF)	✗	✗	✗
Intender [52]	✗	✓	-	N/A	N/A	■	Intent-State Transition	REST & App API	Topology, (OF)	Generation	✗	Generation
Ambusher [51]	✗	✓	-	N/A	N/A	□	Cluster Protocol State	East-West Interface	(OF)	✗	✗	✗
CHIMERA	✓	✓	**P4**	□	◩	◩	**DC, CC**	**App API**	**(P4Runtime)**	**Mutation**	✓	**Mutation**

Triggers. Table 1 shows the number of bugs of each component triggered by different input types among 24 bugs under study. We categorized input types as data plane packets, control plane packets (*i.e.,* `PACKET_OUT`), and control plane rules. In the last row, we separately collected bugs triggered by *multi-plane inputs*: (1) bugs triggered by data plane packets with specific rules and (2) bugs triggered by control plane rules but detected by the data plane.

Key findings. A large majority (83.3%) of P4 infrastructure bugs occur with control plane inputs, compared to 8 of 39 bugs (20.5%) triggered by only data plane packets. Control plane inputs trigger all controller bugs, 85.7% of which are caused by control plane rules. Multi-plane inputs are required for a non-negligible number (20.8%) of bugs in both planes; 40% of switch bugs need *both* data plane packets and control plane rules to be discovered.

3.2. Programmable Network Fuzzing Tools

Table 2 shows existing fuzzing tools for programmable networks with data plane fuzzers [76], [77], [90] and control plane fuzzers [34], [46], [51], [52], [57], [58], [59]. P4-based data plane fuzzers improve packet mutations by reducing header spaces from P4 static analysis [76] or generating packets at line rate with a high-throughput programmable switch [90] but typically require user-defined control plane rules and assertions to identify abnormal program behavior. Early control plane fuzzers [46], [58] investigated OpenFlow-based controllers by mutating southbound APIs but are tightly coupled to OpenFlow and cannot be leveraged for P4 networks. Other control plane fuzzers that test controller internals (*e.g.,* datastores [34], network policy [57], or intent subsystem [52]) can apply control plane rules but are limited to generating purely random values rather than fine-grained rules defined by the underlying P4 program.

Key findings. Based on Table 2, current bug and vulnerability discovery fuzzing tools for a specific plane do not consider or simplify interactions with the other plane, despite our findings that many bugs manifest across multiple components and require multi-plane inputs to discover.

3.3. Threat Model

We assume two attacker models for adversaries exploiting P4 network infrastructure based on the capability to inject multi-plane inputs: data plane attackers and control plane attackers. In both models, we trust the data plane implementations (*i.e.,* a switch and a P4 program), the controller, and core applications running on the controller (*e.g.,* host provider and link provider).

Data plane attackers. We assume that the attacker in an end host can send malicious packets to the victim switch. The attacker can deliver data plane packets to the controller and applications if the controller installs a flow rule that sends packets to the controller (*e.g.,* DHCP relay [17]).

Control plane attackers. We assume that the attacker installs a third-party application that can inject control plane packets and rules, as shown in existing work [29], [81], [82]. Operators can deploy third-party applications, readily available on open-source platforms (*e.g.,* Github) as part of their system integration, potentially exposing them to software supply chain attacks. The controller implements access controls to confine an application's capabilities [91]. For example, ONOS has a set of read and write permissions for different network objects (*e.g.,* {`READ`, `WRITE`} for {`PACKET`, `FLOWRULE`, `HOST`, `LINK`, *etc.*}). We assume that the malicious application does not have the write permission for network topology (*e.g.,* `HOST_WRITE`) since misused topology API can paralyze the managed network by removing all devices and hosts [2]. Also, the application only has `FLOWRULE_WRITE` without `FLOWRULE_READ` to prevent it from learning existing rules installed by others.

3.4. Motivating Vulnerability Example

We describe a newly discovered P4 network infrastructure vulnerability that requires both control and data plane

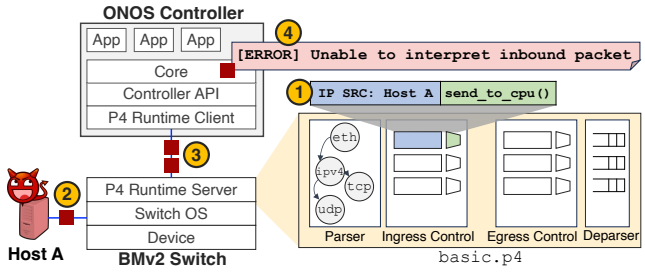


Figure 2: CVE-2024-53423 attack. ONOS rejects Host A's malformed packets, which causes resource exhaustion (steps 1–4).

inputs to be triggered. We highlight what makes this attack succeed and why the aforementioned state-of-the-art tools are insufficient for discovering such vulnerabilities.

Discovered vulnerability. Figure 2 shows a data plane attacker exploiting BUG 6, which causes resource exhaustion in ONOS and leads to denial of service (CVE-2024-53423). We used the `basic.p4` program, but this vulnerability can occur in different P4 programs (*e.g.,* `fabric.p4`) since the vulnerability source is the network infrastructure. (1) ONOS installs a flow rule with `send_to_cpu()` action to receive data plane packets. In `basic.p4`, the `send_to_cpu()` action sets the egress switchport as the CPU port (*i.e.,* controller port) and delivers packets to the ONOS controller. (2) Host A sends malformed IP packets to the network. The switch handles received packets based on the installed P4 program, `basic.p4`. These packets match the rule (1) with the `send_to_cpu()` action. (3) Since the switch does not validate the malformed IP packets, packets will be sent to the controller as `PACKET_IN` messages. After receiving messages, the controller parses packets and forwards them to applications that listen to `PACKET_IN` events. (4) However, the controller fails to deserialize the malformed packets with an error.

Preliminary exploitation. We ran two virtual machines in Google Cloud Platform with 4 vCPUs, 32 GB memory, and 120 GB balanced persistent disk. The network bandwidth between two virtual machines is 7.6 Gbps. We ran ONOS v2.7.1 on one VM and Stratum-BMv2 with emulated hosts on the other VM.

The attacker in the emulated host floods malformed IP packets with `tcpreplay` at high bandwidth (1174.66 Mbps). After 5 seconds, another host tries to send ARP packets, and the controller should register this host. However, ONOS only discovers the new host 7 seconds after the first ARP packet because the controller wastes its resources on parsing flooded malformed IP packets. When the attacker floods malformed packets for longer, ONOS can find the new host after several minutes. The attacker can exploit such degradation to block new benign hosts from communicating.

Security impacts. Since the high-performance P4 switch does not filter out malformed IP packets, the controller has to process flooded packets in a slow path, which causes resource exhaustion and denial of service of legitimate requests for new flow rules by other hosts on the network. This new cross-plane vulnerability occurs due to the semantic gap between the controller and the switch with the P4 program, affecting the whole network.

3.5. Challenges in Fuzzing P4 Networks

Based on our bug study, our analysis of the gaps in existing tools, and our motivating vulnerability example, we establish several research challenges in discovering vulnerabilities and bugs in P4 network infrastructure, which we tackle with our solutions in CHIMERA.

Challenge (C1): Complex bug and vulnerability discovery needs to consider a "full-stack" P4 network infrastructure. As shown in Table 2, existing programmable network fuzzers only test a specific component or plane without accounting for the behavior of the other plane. Our bug investigation in Section 3.1 shows that multi-plane inputs trigger a non-negligible number of bugs in the P4 network. Also, the semantic gap between the control and data plane can cause the example vulnerability in Section 3.4. As a result, a comprehensive P4 network fuzzing tool needs to test a whole P4 network infrastructure. Furthermore, simply combining a data plane fuzzer with a control plane fuzzer is insufficient towards discovering such vulnerabilities, given that the inputs of both planes are necessary for finding such complex bugs and vulnerabilities.

Our solutions: We propose a full-fledged fuzzing framework for a P4 network to test both the control and data plane implementations (Section 5.3). We show our advantages by describing security case studies (Section 7.1) and performance evaluations (Section 7.2).

Challenge (C2): P4 network fuzzing needs to consider interdependencies between P4 programs and multi-plane inputs. The motivating vulnerability example in Section 3.4 shows that a bug in the controller occurs when there are both a *specific* rule and a *matched* packet. If the P4 network runs a complex P4 program with multiple match-action tables, a bug may require a set of specific actions for tables, a packet matched to table entries, and specific metadata fields read and written during the P4 pipeline. Random values for either the packet or the rule do not guarantee a way to satisfy interdependencies among packet headers, match fields, and P4 programs. Therefore, a P4 multi-plane network fuzzing tool needs to leverage not only P4 program grammar but also dependencies between input spaces, which cover more code of the P4 program and have a more significant impact on the P4 network.

Our solutions: We propose two novel mutation policies for packets and rules (Section 5.2) and show that our policies perform better than random mutations (Section 7.2).

Challenge (C3): Validation should not depend on manual effort. As shown in Table 2, existing fuzzing tools that typically focus on program semantics rather than infrastructure require user inputs for a bug oracle, validation checks, and a feedback mechanism, such as annotations, assertions, or queries. Such an approach requires non-trivial manual work to write verification rules, which must be updated and

maintained whenever the P4 program is updated. User inputs are error-prone and may be incomplete, undermining bug-finding efficiency in fuzzing.

Our solutions: We propose concolic execution to leverage P4 programs as a bug oracle (Section 5.1). Based on the expected output, we provide a verification method to compare against the actual output (Section 5.3).

4. Overview

We present CHIMERA, a multi-plane bug and vulnerability detector for P4 networks. Figure 3 shows the overview of the CHIMERA framework. CHIMERA broadly consists of two phases: (1) the preparation and (2) the runtime phase.

Preparation. ❶ CHIMERA parses a given P4 program into a P4 intermediate representation (IR) language. ❷ The IR is the only input for P4CE, the concolic execution tool for our bug oracle, which will validate mutant rules and calculate the expected output from a mutant input packet in the runtime. ❸ Users can selectively run a P4 test generation tool [66], [72] to create test cases, which CHIMERA uses as seed inputs. CHIMERA supports multi-plane fuzzing by processing each test case with an input packet, control plane configurations (e.g., table rules), and an expected output, such as an egress packet or drop. ❹ The translator for parser-aware packet mutation (*PAPM translator*) converts the P4 program into a packet mutator. During fuzzing, the translated mutator will guarantee that mutant packets are within a state space specified in the P4 program.

Runtime. CHIMERA executes seeds and stores interesting seeds in the *seed corpus*. ① CHIMERA picks a seed from the corpus to mutate its input packet and rules. ② CHIMERA runs input mutators, including our new approaches: parser-aware packet mutation (PAPM) and header-guided rule generation (HGRG). For HGRG, CHIMERA receives generated rules from P4CE (*table entry generator*). ③ As CHIMERA mutates the input packet and rules, altering the expected output, CHIMERA provides the mutant to P4CE.[4]

In the emulation step, P4CE runs concolic execution on the P4 program with the given mutant seed (*i.e.,* mutant input packet and mutant rules) to verify rules and calculate the expected output. ④ P4CE validates the mutant rules by checking whether the rules' names, types, and values match the P4 program (*table evaluator*). ⑤ P4CE performs concolic execution with the given packet (*packet evaluator*) to step through P4 program statements, mark coverage, and evaluate the expected output. Finally, P4CE returns the mutant rules with validity marks and the expected output, serving as a reference model for rule and packet verification.

In the verification step, CHIMERA executes rule installations and packet testing in the target P4 network based on the result received from P4CE. ⑥ The *P4 rule validator* checks whether the mutant rules are correctly installed via the *application agent*, reads the mutant rule states, and compares the validity marks. ⑦ The *network manager* marshals a source to send an input packet and a destination to receive the expected output packet, with the destination comparing the received and expected outputs. ⑧ CHIMERA checks whether the test case causes any errors by leveraging bug detectors, including P4CE's expected output.

CHIMERA checks the test case's result to log errors. CHIMERA gathers all coverage data from P4CE, the controller, and the data plane switches for feedback. If the test case finds new coverage, CHIMERA stores it in the seed corpus. CHIMERA repeats steps ① – ⑧ until terminated.

4. Given an input packet, CHIMERA requests P4CE once to calculate the expected output from random rules or generate both rules and the output.

5. Design

We set out several design goals for CHIMERA based on the technical challenges **C1–C3** in Section 3.5:

Goal (G1): Multi-plane P4 network testing. To address bugs that occur in both planes and are triggered by multi-plane inputs, a P4 network fuzzer should include both the controller and the switch as a whole target with the underlying P4 program **(C1)**.

Goal (G2): Multi-plane input mutations based on inter-dependency. To trigger complex bugs, a P4 network fuzzer should cope with interdependencies among data plane packets, control plane rules, and P4 program grammar **(C2)**.

Goal (G3): Bug oracle without human involvement. To support diverse programs without relying on program semantics, the fuzzer should only need users to provide a P4 program **(C3)**. The fuzzer should correctly compare testing results to expected behavior by referring to the P4 program.

Goal (G4): Abstract design. The fuzzer should provide developers with abstraction layers, allowing testing for multiple controller implementations and diverse switch targets under different P4 programs.

5.1. P4CE: Concolic Execution as a Bug Oracle

A P4 program defines how data plane devices process packets depending on control plane rules installed by the controller, which means that the underlying P4 program reflects the expected behavior of the target P4 network. Thus, we leverage concolic execution on the P4 program as a bug oracle of multi-plane P4 network fuzzing **(G1)**.

CHIMERA includes a P4 concolic executor (P4CE) to extract necessary information from the underlying P4 program. P4CE consists of the *table evaluator* validating random rules and the *packet evaluator* calculating the expected output from a random input packet. P4CE leverages concolic execution on the P4 IR generated by target-specific compilers **(G3)**. We employ concolic execution on top of P4Testgen [72] using concrete values for rules and packets. P4Testgen supports diverse P4 implementations (*e.g.,* Tofino and eBPF), so P4CE can support them with minimal modification **(G4)**. Notably, although P4Testgen can generate test cases, P4Testgen is limited in that it does not provide any decision of correctness of a test case on its own; hence, we design two bug oracles:

Bug oracle for rule verification. CHIMERA incorporates a static analysis tool, the *table evaluator*, to verify whether

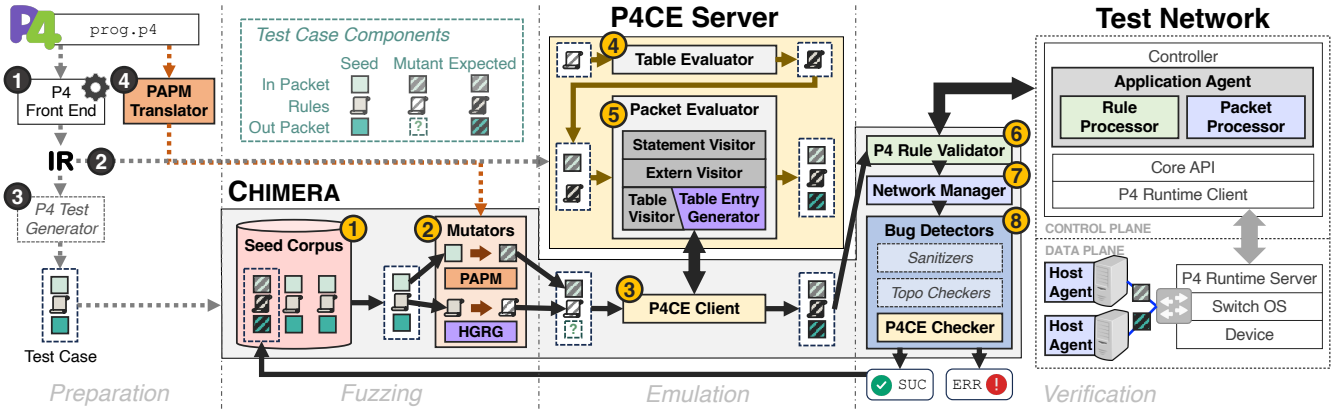


Figure 3: Overview of the CHIMERA framework with two phases: preparation (black-colored numbers) and runtime (yellow-colored numbers). CHIMERA executes fuzzing, emulation, and verification steps with P4CE and the test network during the runtime phase. New components are highlighted in colors.

random table entries are valid in the P4 program. The validity marks of table entries are utilized not only as a measure for rule verification but also as a way to filter out invalid rules before running the subsequent concolic execution in the packet evaluator. To this end, the table evaluator first collects all existing tables from the P4 IR. For each table, the table evaluator checks names and identifiers of match, action, and parameter fields. If the table entry's field exists, the table evaluator compares its type and bit length. If the given value is within the range of bit length described in the P4 IR, P4CE marks the validity bit of the table entry as valid. Such validity will become a measure of the success or failure of the rule installation.

The table evaluator's results become the bug oracle of control plane rule testing in Section 5.3. CHIMERA can detect an error if the state of the installed rule does not align with what was calculated by P4CE. For example, assume that a P4 program defines a match-action table that has a longest prefix match (*e.g.,* 10.0.0.1/24) with a key of an IPv4 destination address (*e.g.,* `ipv4_dst`) and specific actions (*e.g.,* `drop` and `send_to_cpu`). P4CE can distinguish an invalid rule with incorrect values, such as a wrong name for its match field (*e.g.,* `ipv4_src`), a wrong match type (*e.g.,* exact type such as 24), an out-of-range value (*e.g.,* 300.0.0.1/24 with 33 bits), or a wrong name for the action (*e.g.,* `drop_to_cpu`). The rule testing will succeed if CHIMERA successfully installs valid rules or fails to install invalid rules.

Bug oracle for packet verification. CHIMERA incorporates the *packet evaluator* to compute the expected output (*e.g.,* an egress packet or drop) based on the input packet and valid table entries. The packet evaluator executes the P4 program's *parser* to derive in-range header values of the input packet. The *statement visitor* steps through the program *control*'s statements and conditional branches while measuring covered statements. For externs, the *extern visitor* executes target-dependent methods. For tables, the *table visitor* checks each valid entry of the table in priority order and measures covered actions. If the packet matches the valid entry, the table visitor executes its action; otherwise it chooses the table's default action.

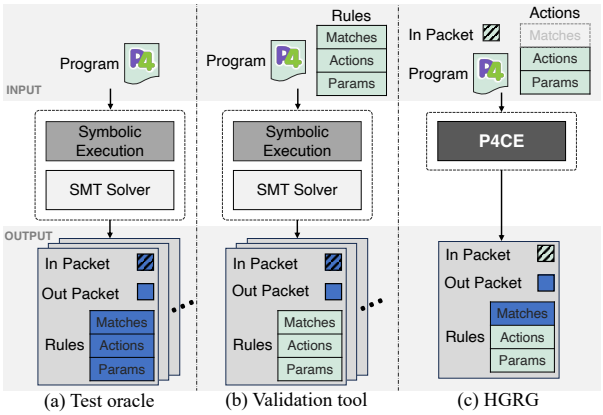
The packet evaluator's results become the bug oracle of data plane packet testing in Section 5.3. CHIMERA can detect an error if the packet testing result differs from the expected output of P4CE. For example, suppose an installed rule sends an input packet with the broadcast IPv4 address to the controller (*i.e.,* 255.255.255.255/32 for `ipv4_dst` and `send_to_cpu` action). P4CE can expect the output as an egress packet identical to the input packet and the egress switchport as the controller port (*e.g.,* 255 for BMv2). The packet testing will succeed if CHIMERA notices that the controller receives the expected output packet.

5.2. New Fuzzing Input Mutators

Compared to existing mutators for packets or rules, CHIMERA's two new mutation policies use inter-dependencies among multi-plane inputs to efficiently discover multi-plane bugs **(G2)**.

Parser-Aware Packet Mutation (PAPM). Random mutations in packets, such as bit flips, can be challenging to obtain even a single UDP packet from a TCP packet (*e.g.,* from `6` to `17` for the 8-bit IPv4 protocol field). Even though a fuzzer may employ standard packet libraries (*e.g.,* `libpcap` [10]), discrepancies arise among libraries and P4 program parsers. A fuzzer may create unnecessary packets that are not used in the P4 program or cannot generate interesting packets with custom headers (*e.g.,* In-band network telemetry (INT) [41]). Since a P4 program typically ignores packets that are not parsed, a fuzzer without efficient packet mutations will face difficulty in finding bugs.

CHIMERA directly leverages the P4 program's *parser*. In the preparation phase, the *PAPM translator* converts the parser code of the P4 program into a packet mutator of CHIMERA (see Figure 11 in Section A.1 for an example). The translator represents each parser state block as a parser method of the mutator class (*e.g.,* `parse_ipv4`). The mutator first receives a seed test input containing an input



(a) Test oracle (b) Validation tool (c) HGRG

Figure 4: Comparison of P4 analysis tools: (a) test oracles [66], [72], (b) validation tools [21], [37], and (c) header-guided rule generation (HGRG). While (a) and (b) are based on symbolic execution and generate multiple test cases, HGRG (c) uses concolic execution (P4CE) with the concrete input packet to get a single test case. Input and output values are marked in light green and blue color.

packet and its execution path of parser states. Then, the mutator mutates the executed parser-state path by removing a random number of items from the end. While following the mutant path of parser states (*e.g.,* `stateItr`), the mutator puts the specific value (*e.g.,* 17) into the corresponding header field (*e.g.,* `ipv4.protocol`), which is the condition of its successor state (*e.g.,* `parse_udp`). When the path ends at a specific parser state, the mutator randomly chooses one of its successor states. At the end of the parser, the mutator decodes all parsed headers into input packet bytes. Since the packet mutator is directly translated from the P4 parser code, CHIMERA can always generate valid packets used by the underlying P4 program.

Header-Guided Rule Generation (HGRG). A control plane rule consists of a table name, a set of match field(s), an action name, and a list of action parameter(s) based on the underlying P4 program. A fuzzer with random mutators may be unable to create a single valid rule that conforms to the P4 program's grammar. While the fuzzer can generate grammatically valid rules by referring to the P4 program (*e.g.,* SwitchV's p4-fuzzer [21]), it is unlikely that random packets (*e.g.,* 10.0.0.1 for destination) will match random values in rules (*e.g.,* 12.34.56.78/32 for `ipv4_dst`). Complicating matters, a P4 program can allow rules to have a match field with metadata updated in pipeline processing.

To generate an input packet matching the given rules, one could potentially use static analysis tools [21], [37], [66], [72] that run symbolic execution on the P4 program and an SMT solver to find the test case that satisfies conditions, as depicted in Figure 4. However, this approach is not suitable for a fuzzing mutator since such tools explore *all* possible execution paths of the P4 program to generate multiple test cases in a single run.

Instead, we opt to generate rules from the input packet. For example, if the packet's destination IPv4 address is 10.0.0.1, we can set the `ipv4_dst` match field to 10.0.0.1 instead of a random value. This provides two opportunities: P4CE can fill match fields related to metadata since it recognizes the corresponding metadata values (*e.g.,* `flow_id`); and P4CE can function as a fuzzing mutator, as it executes a single code path to obtain a single test case.

To support HGRG, P4CE employs the *table entry generator*, which creates a new rule when P4CE reaches a table by setting match fields with corresponding values of either packet headers or metadata fields. If an action is not given for the table, the table entry generator selects a default action for the table and continues the execution. P4CE returns the expected output with a list of generated rules.

5.3. P4 Network Fuzzing Framework

CHIMERA executes the P4 network fuzzing process in both planes **(G1)**. While requesting multi-plane inputs to the target P4 network, CHIMERA also manages results and tracks coverages in both planes.

In the **fuzzing step**, the *seed corpus* collects interesting seed test cases. For multi-plane interactions, a seed includes an input packet, an output packet, and a list of control plane rules. Users can configure the coverage metrics that the seed corpus refers to **(G4)**, such as P4 program coverage metrics (*e.g.,* actions and statements). *Mutation operators* mutate the selected seed. CHIMERA supports diverse mutations for an input packet, a single rule, and a set of rules. For packets, CHIMERA supports PAPM and random mutations such as bit flips. Given a set of rules, CHIMERA can add, delete, or modify a rule, merge rules in another seed, or generate valid rules by using HGRG. When mutating a single rule, CHIMERA can execute random and P4-aware mutations. CHIMERA mutates the string names (*e.g.,* match field) or values with the random rule mutation. With the P4-aware rule mutation, CHIMERA selects a type among valid types of the underlying P4 program (*e.g.,* longest prefix match) with corresponding values (*e.g.,* value with a prefix length) by requesting to P4CE. Users can configure existing mutation policies or write their own methods **(G4)**.

In the **emulation step**, the P4CE *client* communicates with P4CE interactively to request mutant test cases and receive expected outputs, valid table entries, and coverage metrics. CHIMERA's decoupling of the P4CE server from the framework allows analyzing mutants across various P4 programs in the network **(G4)**.

In the **verification step**, the *P4 rule validator* installs the given control plane rules by sending them to the *application agent*. This SDN application translates these rules into the SDN controller's northbound APIs. The P4 rule validator checks the state of each rule in the controller. If valid entries are installed correctly or invalid entries show the error state, the verification of rules succeeds. The *network manager* manages *host agents* in the target network to conduct data plane packet verification. After the rule verification passes, CHIMERA sends the given input packet to the host agent connected to the specified ingress switchport of the target switch. If the ingress switchport is the controller port,

TABLE 3: Mutation rules in CHIMERA.

Rule	Description
M1: *Coverage metrics*	Set of coverage metrics for feedback
M2: *Max num of seed corpus*	Max number of seeds in corpus
M3: *Packet-to-rule ratio*	Ratio of packet mutations to rule mutations
M4: *Max num of rules*	Max number of rules in each test case
M5: *Timeout for rule testing*	Max wait time for rule status update
M6: *Timeout for packet testing*	Max wait time for packet delivery

CHIMERA requests the packet to the application agent to send it as a `PACKET_OUT` message (*i.e.,* control plane packet) **(G1)**. The expected egress switchport of the test case can be either a switchport, controller port, or drop. CHIMERA monitors all outputs, including the controller, and compares the result with the expected output. (Detailed packet verification steps are provided in Section A.2.)

In addition to P4CE, users can configure bug detectors, such as sanitizers for both the switch and the controller **(G4)**. Following existing control plane fuzzers [34], [46], [51], [57], [58], CHIMERA can also monitor the topology information in the controller to see if the test case impacts core applications (*e.g.,* host provider and link provider) and causes inconsistency between both planes.

Feedback mechanism. By default, CHIMERA leverages the controller and software switch code coverage using the *coverage agents*. CHIMERA can enable P4 coverage as feedback by reading the test case metadata written by P4CE. P4 coverage has a bitmap and the maximum number of coverage entities (*e.g.,* total number of actions). CHIMERA will store this case in the corpus if the test case finds an interesting path in any coverage metric that users register.

Mutation policy. CHIMERA offers four mutation rules (Table 3). **M1** is the set of coverage metrics for feedback to store interesting seeds. Users can set the maximum number of seeds in the corpus by specifying **M2**. If the number of seeds exceeds **M2**, CHIMERA executes a seed minimization process based on registered coverage metrics defined in **M1**. Users can regulate the number of packet mutations compared to the number of rule mutations by specifying **M3**. CHIMERA might allow too many rules in a single test case (*e.g.,* merging rules from another seed); to avoid this, users can set the maximum number of rules in each test case by controlling **M4**. Users can control **M5** and **M6** to set the maximum time to wait for rule test results and packet test results, respectively.

6. Implementation

We implemented CHIMERA in Java in 36.4K lines of code and used ONOS v2.7.1 as the P4 control plane implementation. Our implementation is available at https://github.com/purseclab/Chimera. We used JaCoCo v0.8.6 [43] to measure ONOS code coverage and Soot v4.2.1 [85] to extract P4-relevant methods and entry points. We employed OptiMin [42] to reduce the number of stored seeds. We wrote the application agent in Java to translate flow rules and request packet I/O using ONOS APIs. We captured default rules installed by ONOS core applications via the P4Runtime client [14] connected to a read-only channel of the switch under test. We executed a RabbitMQ server to listen to network topology events from the ONOS controller. CHIMERA can be used with other controllers (*e.g.,* OpenDaylight) with minimal modifications at the API layer.

We ran the Stratum-BMv2 software switch (Stratum 2022-06-30 branch, BMv2 v1.14.0 with commit `6cb0907`) with the `afl-clang-lto` included for edge coverage. We included PI and BMv2 shared libraries and the Stratum binary to cover all code paths related to data plane processing (468.5K edges in total). We also applied AddressSanitizer [75] and UndefinedBehaviorSanitizer [33] on Stratum-BMv2. We reset all coverages before running each test case to measure code paths executed during the test. We built our test agent in Python with Mininet v2.3.0 to support our packet verification and tailor a network topology. Each emulated host executes the host agent written in Python with Scapy to send and sniff raw packets. CHIMERA can test other targets supporting P4Runtime (*e.g.,* Tofino [32], P4rt-OVS [12], [67], and PINS [4]).

We implemented P4CE by extending P4Testgen v1.2.4.1 [72] supported in P4C [13] in 23.1K lines of code. We facilitated interactive connections between P4CE and CHIMERA via gRPC. We also implemented the PAPM translator by referring to the P4-to-C translator code in ASSERT-P4 [37]. CHIMERA can be used in other P4 programs by extending P4Testgen, which supports eBPF [7], PNA [40], and Tofino's TNA [32].

We could not locate public repositories of existing P4 fuzzers, so we simulated their behaviors on our architecture, except for the user-specified feedback mechanism (*e.g.,* P6 [76]). We simulated FP4 [90] by setting the corpus to separate seeds by a set of table entries and using the P4 action coverage metric. For a control plane fuzzing tool, we used Intender [52] for interactions with P4 switches.

7. Evaluation

We evaluated CHIMERA with ONOS v2.7.1, and our network used Mininet v2.3.0 with a Stratum-BMv2 switch. We evaluated three P4 programs officially supported by Open Network Foundation for ONOS and fabric-tna: `basic.p4`, `int.p4`, and `fabric_v1model.p4`. Also, we selectively enable two P4 preprocessor macros (`WITH_UPF` and `WITH_INT`) in `fabric_v1model.p4` to test six transformed P4 programs in total, as shown in Table 4. ONOS requires a specific translator/interpreter for P4 programs. To the best of our knowledge, we cover *all* possible P4 programs supported by Open Network Foundation, such as ONOS [25] and fabric-tna [8]. We ran all components in a single virtual machine in Google Cloud Platform: 4 vCPUs, 32 GB memory, and 120 GB disk. We executed all experiments ten times to obtain reliable average values. Error bands represent a 95% confidence interval.

5. The path is a sequence of conditional statements, including P4 program branches and all actions of tables. We measured the number of paths by multiplying the ones of each programmable block computed with Ball-Larus encoding [24], [55].

TABLE 4: Tested P4 Programs. `basic`, `int`, and `fabric` stand for `basic.p4`, `int.p4`, `fabric_v1model.p4`, respectively. `fabric_int`, `fabric_upf`, and `fabric_upf_int` are `fabric_v1model.p4` with `WITH_INT`, `WITH_UPF`, and both `WITH_UPF` and `WITH_INT`, respectively.

P4 Program	LoC	Tables			Actions			Statements	Paths[5]	Rules in Seed
		Config	Hidden	Total	Config	Hidden	Total			
`basic`	741	3	0	3	7	0	7	37	3136	7
`int`	1612	5	2	7	10	34	44	131	2.18×10^{7}	7
`fabric`	2567	19	1	20	39	5	44	265	1.03×10^{16}	25
`fabric_int`	3440	23	4	27	50	10	60	445	3.2×10^{53}	30
`fabric_upf`	3215	29	1	30	65	5	70	412	7.09×10^{23}	44
`fabric_upf_int`	4130	33	4	37	76	10	86	606	2.36×10^{70}	48

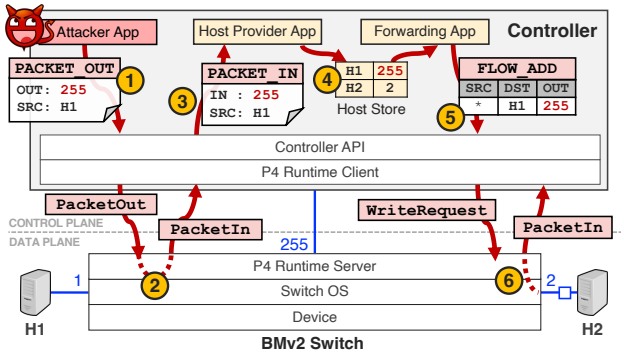


Figure 5: CVE-2023-41591 attack. The attacker application executes a man-in-the-middle attack on the victim host H_1 (steps 1–6). Links and their respective switchport numbers are marked in blue.

Mutation rules. We configured the default values for all mutation rules in Table 3. For **M1**, we used control plane code coverage (CC) and data plane code coverage (DC), except for FP4 using P4 action coverage. We set 1000, 16, and 100 for **M2**, **M3**, and **M4**, respectively. For timeouts, we specified 2 seconds for rule tests (**M5**) and 100 ms for packet tests (**M6**).

Responsible disclosure. For all of the vulnerabilities we discovered, we notified the ONOS, SD-Fabric, and P4.org teams of our findings at least 120 days prior to disclosure.

7.1. Discovered Vulnerabilities and Bugs

We analyze two previously unknown bugs as vulnerability case studies. Existing tools were not able to discover them because such tools only focused on one plane each.

Case study 1: Host tampering from a third-party application (BUG 2; CVE-2023-41591). Figure 5 shows the workflow of a control plane attacker who can execute a man-in-the-middle attack on a victim host. ① The attacker sends a crafted `PACKET_OUT` message with the egress switchport set to 255, the reserved port for the controller in BMv2. The attacker can fill the source IP/MAC addresses of the packet with a victim host, H_1. The controller translates the given `PACKET_OUT` message into P4 Packet I/O APIs (*i.e.,* `PacketOut`) and sends it to a target P4 switch. According to `PacketOut`, the P4 program of the switch transmits the packet to the egress switchport. ② However, since the egress switchport is the controller port number (*i.e.,* 255), the switch sends back this packet to the controller as the `PacketIn` message with the ingress switchport as 255.

③ The controller receives the `PACKET_IN` message with the same packet sent by the attacker. The host provider app registers a new host or updates the existing host by referring to received `PACKET_IN` messages. If the ingress switchport of `PACKET_IN` is one of the reserved values, such as the controller port used in OpenFlow, the app ignores the `PACKET_IN` message. ④ However, since the application does not know that 255 is the controller port used in the BMv2 switch, it updates H_1 information as connected to the switchport 255. ⑤ To support data plane connectivity to the updated H_1, network control applications such as the forwarding app will install a new rule that forwards packets destined for H_1 to the switchport 255. ⑥ After the new rule is installed, the attacker can receive any packet sent to H_1 and relay it to H_1 to deceive the victim into believing the connection is established.

Takeaways: CHIMERA can uncover this new vulnerability with the control plane input (*i.e.,* `PACKET_OUT`). This vulnerability occurs due to the semantic gap between P4 switches and the controller. Unless a fuzzing tool tests both planes at the same time, this bug will not be discovered.

Case study 2: Side-channel attack by installing a rule with identical match fields and priority (BUG 4). An attacker in a third-party application can infer current flow rules installed by other applications. The controller checks the state of flow rules in every configured flow polling cycle. However, due to BUG 4, the controller does not actively monitor a new rule if an existing flow rule has identical match fields and priority. Instead, the new rule will be updated later when the switch sends a subsequent message. The new rule's duration appears as 0 since the controller does not monitor the new rule. By leveraging this bug, the attacker can probe random rules and learn existing rules.

Takeaways: CHIMERA can discover this vulnerability by mutating control plane rules based on P4 program grammar.

Other newly found vulnerabilities and bugs. Table 5 shows 7 unknown bugs found by CHIMERA: 1 bug in BMv2, 6 bugs in ONOS. Among bug detectors used by CHIMERA, P4CE discovered 5 of 7 new bugs, while UBSAN and

topology checking found BUG 1 and BUG 2, respectively.

The BMv2 bug allows memory error (BUG 1), and all new ONOS bugs relate to errors introduced in Section 3.1: mismanagement in core (BUG 2–4), specification mismatch (BUG 5), and translation error (BUG 6, BUG 7). Among them, 2 bugs can only be triggered when providing both control plane rules and data plane packets (BUG 1, BUG 6). 2 bugs are cross-plane bugs caused by semantic gaps between the control and data plane (BUG 2, BUG 6). 3 bugs are security vulnerabilities and cause violations such as *host tampering* (BUG 2; CVE-2023-41591), *side channels* (BUG 4), and *denial of service* (BUG 6; CVE-2024-53423).

Takeaways: P4CE can discover the majority (71.4%) of newly found bugs in the P4 infrastructure.

7.2. Performance Evaluation

We compare CHIMERA's fuzzing efficiency to random mutations (*i.e.,* **Random**), **FP4** [90], and **Intender** [52] on six P4 programs, as shown in Table 4. Random uses CHIMERA's same coverage feedback and bug detectors but randomly mutates control plane rules and data plane packets.

FP4 is a line-rate fuzzing framework for P4 data plane switches. We implemented FP4 in x86 since the programmable switch code was unavailable. FP4 does not mutate rules during fuzzing. To create a fair performance comparison against FP4, which relies on the seed rule quality, we manually modified a seed generated from P4Testgen [72] for each P4 program to include as many P4 actions as possible. We used this seed for FP4, Random, and CHIMERA.

Intender is a fuzzing framework for intent-based networking (IBN), which abstracts P4-specific rules into network intents. Intender mutates intents and generates representative packets for a set of existing intents. Intender receives two reachability intents instead of P4-specific rules as a seed input. We tested Intender only for `basic` and `int` since ONOS's IBN does not support translating intents into rules of multi-table P4 programs.

Coverage analysis. Figure 6 shows the controller's covered branches with six different P4 programs for 24 hours. The branch coverage ratio varies across P4 programs due to differences in their Pipeconf packages. Among all fuzzers, CHIMERA has the highest branch coverage of the controller with any P4 program except for `int`. For `basic`, CHIMERA shows 1.08× and 1.07× higher branch coverage than FP4 and Intender. CHIMERA shows 1.1×, 1.09×, 1.07×, and 1.08× higher branch coverage than FP4 for `fabric`, `fabric_int`, `fabric_upf`, and `fabric_upf_int`, respectively. For `int`, CHIMERA shows 1.1× higher branch coverage than FP4 and similar coverage to Intender with a difference of less than 1%. However, as shown in Figure 7, the methods covered by CHIMERA differ from those of Intender. While Intender covers 37 methods related to path calculation for intents and intent APIs, CHIMERA calls 25 methods related to P4-specific inputs (*e.g.,* flow, group, and packet APIs). Compared to FP4, CHIMERA can cover all methods found by FP4 for all P4 programs, except for a few methods related to event handling or background processes in the controller core, not from fuzzing inputs. All methods only found by CHIMERA are involved in P4 APIs, packet libraries, and protocol implementations. Compared to Random, CHIMERA shows up to 1.03× higher coverage, which shows that understanding the P4 program can improve in covering more paths in the controller.

Figure 8 shows data plane edge coverage, representing possible transitions among control flow graph blocks. CHIMERA shows the highest coverage. For `basic`, CHIMERA can execute 1.08× and 1.16× more edges than FP4 and Intender. For `int`, CHIMERA shows 1.04× higher edge coverage than both FP4 and Intender. CHIMERA also has 1.1×, 1.12×, 1.08×, and 1.1× higher coverage than FP4 on `fabric`, `fabric_int`, `fabric_upf`, and `fabric_upf_int`, respectively. Compared to Random, CHIMERA shows up to 1.08× higher coverage.

We also measured the P4 action coverage using P4CE, with results shown in Figure 12 in Section B.1. Although CHIMERA does not leverage covered P4 actions as feedback, CHIMERA can achieve up to 2.64× higher P4 action coverage than FP4. While FP4 relies on a set of rules given by a seed input, CHIMERA can create new actions and generate packets matched to mutant rules. A detailed explanation of improvement in P4 action coverage is provided in Section B.1.

Takeaways: CHIMERA's mutation of both the control and data plane inputs shows the highest coverage on both planes. CHIMERA can cover more control plane branches related to fine-grained P4 APIs. CHIMERA can cover not only more data plane edges but also more P4 actions by mutating control plane rules that determine packet behavior. In addition, a complex P4 program can increase the coverage of both planes since both the controller and the switch need to support more diverse types of rules for the P4 program.

Mutation policies without seed rule. To highlight the impact of multi-plane input mutators (*i.e.,* PAPM and HGRG), we evaluate CHIMERA against different policies on `basic`, `int`, and `fabric_upf_int` (which subsumes `fabric`'s features) for 3 hours. To minimize the impact on the seed quality, we run experiments without any control plane rule in a seed. Except for the seed, we use the same configurations for CHIMERA, Random, and FP4. "P4-aware" guarantees that mutant rules are grammatically valid based on the underlying P4 program. We exclude Intender, which provides intent-level mutations, not rule-level mutations.

Figure 9 shows the coverage metrics measured by CHIMERA, P4-aware, Random, and FP4 without any seed rule. CHIMERA shows the highest coverage in both ONOS and Stratum-BMv2. For the control plane, CHIMERA shows 1.32×, 1.29×, and 1.27× higher coverage than FP4 and 1.14×, 1.11×, and 1.13× higher coverage than Random on `basic`, `int`, and `fabric_upf_int`, respectively. For the data plane, CHIMERA achieves 1.25×, 1.18×, and 1.24× higher coverage than FP4 and 1.23×, 1.16×, and 1.22× higher coverage than Random on three P4 programs. Unlike FP4 and Random, which depend highly on predefined seed rules, CHIMERA shows a stable fuzzing performance regardless of the seed quality.

P4-aware can improve both plane coverages compared

TABLE 5: List of previously unknown bugs and vulnerabilities discovered by CHIMERA. **XP** stands for cross-plane bugs.

Target	#	CVE	Trigger	XP	Description	Status
BMv2	1	-	CP rule + DP Packet	✗	Empty action-parameter vector is always accessed, causing *out-of-bound read*	Fixed
ONOS	2	CVE-2023-41591	CP packet	✓	App can create fake host by sending `PACKET_OUT` to the controller port, causing *host tampering*	Reported
	3	-	CP rule	✗	Group cannot be removed after P4 error, causing *resource exhaustion*	Reported
	4	-	CP rule	✗	Rule with the same match field values and priority takes longer to update its state and does not update its duration, causing *side channel*	Reported
	5	-	CP rule	✗	P4 pipeline translator tries to install direct action for indirect table and vice versa	Reported
	6	CVE-2024-53423	CP rule + DP packet	✓	`PACKET_IN` with malformed header cannot be deserialized, causing *denial of service*	Reported
	7	-	CP rule	✗	Default action is always translated into "`NoAction`"	Reported

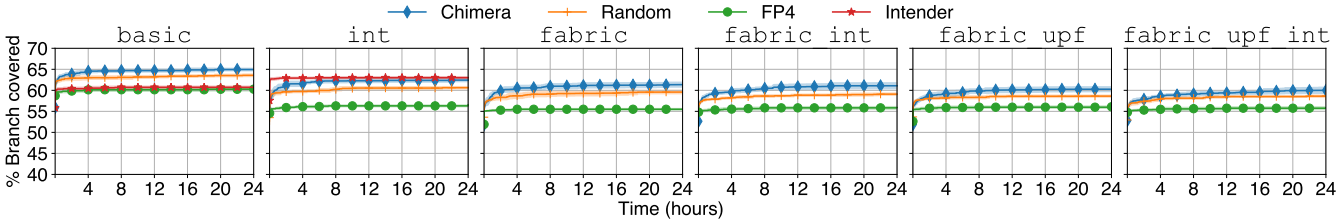


Figure 6: Covered branches of ONOS measured by different fuzzers with six P4 programs for 24 hours.

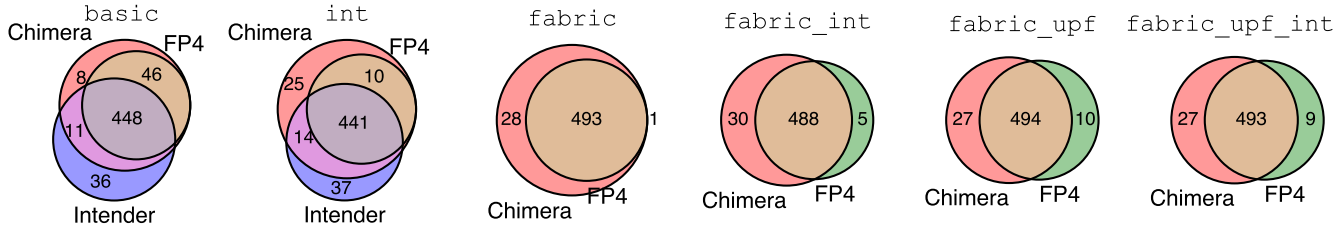


Figure 7: Venn diagram of covered methods of ONOS with six P4 programs for 24 hours.

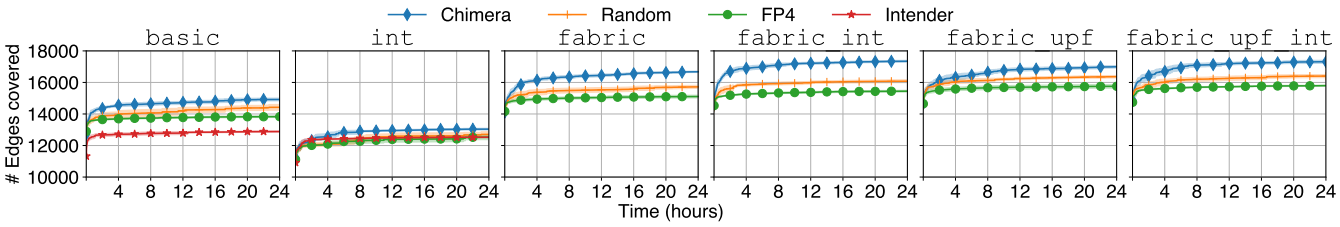


Figure 8: Covered edges of Stratum-BMv2 measured by different fuzzers with six P4 programs for 24 hours.

to Random and FP4. However, CHIMERA with PAPM and HGRG can achieve up to 1.03× higher ONOS coverage and up to 1.07× better data plane coverage than P4-aware. For complex P4 programs, data plane coverage differences become more significant. We measure P4 program statement coverage, shown in Figure 9c. Compared to P4-aware, CHIMERA achieves up to 2.19× higher statement coverage.

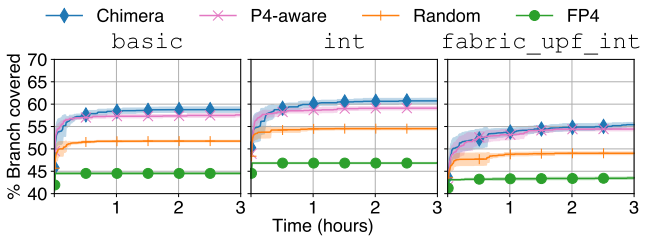
Takeaways: CHIMERA achieves the highest coverage in both planes and reduces the user burden of writing seeds. Since CHIMERA calls more P4 program statements, it can execute more functions of the data plane implementation.

Bug-finding efficiency. We demonstrate CHIMERA's efficiency in finding bugs by showing the *time-to-find* for each new bug, as shown in Table 5. The time-to-find is the average time to discover each bug, and the *reliability* is the bug discovery ratio among 10 runs for 3 hours. We exclude Intender, which tests intent APIs since it cannot find any bug related to fine-grained P4 rules and packets.

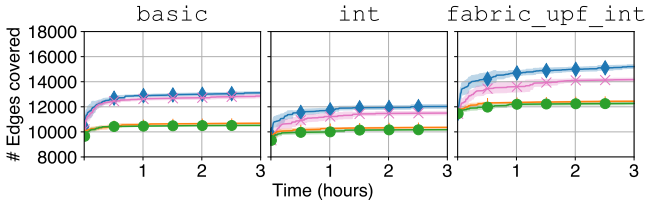
FP4 can find BUG 1 and BUG 6 for `basic` and `int`. BUG 1 can be discovered without any action parameter since ONOS installs default rules to receive ARP and LLDP packets. FP4 can also find BUG 6 since the initial seed contains a rule that sends a packet to the controller, but when we run fuzzers without rules in the seed on `basic` and `int`, FP4 cannot find this bug anymore, while CHIMERA can still find it. By mutating control plane inputs, Random can discover BUG 2, BUG 3, and BUG 6.

Takeaways: CHIMERA discovers all bugs using the P4 program and multi-plane input interdependencies.

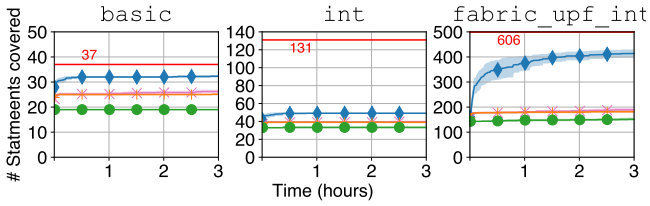
Latency in generating test cases. To mutate and verify



(a) Covered branches of ONOS.



(b) Covered edges of Stratum-BMv2.



(c) Covered P4 statements.

Figure 9: Measured coverages without seed rule on three P4 programs for 3 hours.

multi-plane inputs, CHIMERA leverages P4CE built on top of the P4 test oracle, P4Testgen [72]. We measured the number of test cases generated by CHIMERA and P4Testgen for 3 hours to show our better performance. A test case generated by both tools consists of an input packet, control plane rules, and an expected output packet, which can be used to validate the behavior of the P4 network. As a result, CHIMERA can achieve 2.3×[6], 2.64×, 2.97×, 3.43×, 96.93×, and 17.5× higher throughput (exec/s) than P4Testgen for all six programs, respectively. Compared to P4Testgen, CHIMERA can eliminate the overhead of the SMT solver. CHIMERA can speed up executions with concrete values, while P4Testgen cannot generate test cases with uninitialized variables.

Takeaways: CHIMERA can accomplish higher multi-plane coverage and better performance than existing fuzzing and static analysis tools, respectively.

False positive analysis. We measured the false positive rate (FPR) among found errors during two verification methods: rule and packet verification. We measured FPR from our 24-hour experiments on the simulated FP4, Random, and CHIMERA, which share our bug detectors and framework. Since we set timeout values for the rule (*i.e.,* 2 seconds) and packet verification (*e.g.,* 100 ms), the timeout can also occur in correct test cases. Among all detected errors, we manually classified false positives by replaying errors with extended timeout configurations.

The control plane rule verification has no false positives. The data plane verification shows 1.2% FPR. These false positives occur mainly due to unimplemented features in P4CE as described in Section B.2. Supporting such features of P4CE could potentially reduce FPR to 0.07%, caused by non-deterministic behavior.

Takeaways: P4CE serves as a reliable bug detector with a low false positive rate.

8. Discussion

Cross-plane bugs and vulnerabilities. We found two cross-plane bugs stemming from the semantic gap between the controller's control plane logic and the switch's data plane logic. Prior work on cross-plane attacks [81], [83], [88] has focused on OpenFlow-based SDN, but P4-enabled network infrastructure provides more significant opportunities for cross-plane bugs due to additional programmability.

P4 program complexity. Given the scalable improvements in P4 statement coverage with more complex P4 programs (Figure 9c), we expect CHIMERA's time-to-find to increase gradually with the complexity as bugs located in deep program paths become more challenging to uncover. Rather than testing the full-stack infrastructure with a fixed set of P4 programs, CHIMERA can evaluate more P4 programs by testing only data planes, enabling analysis of the relationship between bug-finding efficiency and P4 complexity. We leave the P4 complexity analysis as future work.

Support for other P4 infrastructure. CHIMERA's modular design supports fuzzing on diverse P4 infrastructure. Control plane APIs can be supported with minimal modification of CHIMERA. Users can also test the controller with other interfaces (*e.g.,* ONOS IBN [3], 5G UPF [62]) by allowing CHIMERA to collect rules from underlying P4 switches. CHIMERA can be extended to other P4 software and hardware implementations since P4CE's use of P4Testgen [72] supports BMv2 v1model [5], eBPF [7], PNA [40], and TNA [32]. To apply fuzzing on hardware switches, black-box fuzzing could incorporate P4 program coverage metrics to find new bugs (*e.g.,* P4 statement coverage [72]). While PAPM and HGRG show improved coverages in both the data plane and P4 programs, we leave the analysis of P4 coverage metrics as future work.

Differential testing. CHIMERA relies on P4CE completeness, which derives from the P4Testgen [72] tool maintained by the P4C community [13]. A limitation of differential testing is the indistinguishability between errors in two test suites (*e.g.,* P4CE and a target P4 network) with identical results, though we did not encounter this.

Non-deterministic behavior. CHIMERA does not support non-deterministic behaviors of P4 switches. To tackle such non-determinism, we can apply state-tracking techniques [23], [48], or dynamic analysis [61] on the target switch by collecting actual outputs for non-deterministic variables. We leave non-determinism as future work.

6. For `basic.p4`, P4Testgen stopped after 288 tests around 48.9 s (5.89 exec/s). CHIMERA generated 145998 tests in 3 hours (13.52 exec/s).

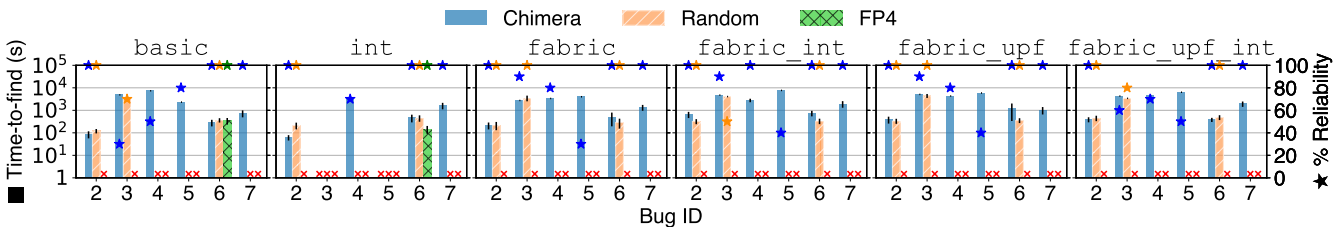


Figure 10: Mean time-to-find each bug by different fuzzers with six P4 programs for 3 hours. The bars are the time-to-find, and the stars are the reliability of each bug in Table 5. ✗ means a bug is not found. We exclude BUG 1, which all fuzzers can find in the beginning.

9. Related Work

For **P4 testing**, Gauntlet [73] and P4Fuzz [20] adopt fuzzing in P4 program generation to find unknown bugs in P4C, an open-source reference compiler for the P4 language. P4Pktgen [66] utilizes a symbolic execution to automatically generate test cases for a target P4 program. P4Testgen [72] leverages symbolic execution, concolic execution, and taint analysis to create a P4 program test oracle. FP4 [90] and P6 [76] fuzz packets to discover unknown P4 switch bugs, but both tools require user efforts to recognize abnormal behaviors through assertions promptly. Meissa [95] uses symbolic execution to discover bugs in P4-related components: P4C, a P4 switch, and its proprietary controller. SwitchV [21] leverages fuzzing to mutate forwarding rules and symbolic execution to generate random packets, but it randomly mutates forwarding rules without any guidance.

Network verification checks against invariants to assert formal properties about the network's configuration. Hydra [71] uses per-packet runtime verification. VeriDP [94] checks for control and data plane inconsistencies. Unlike CHIMERA, both approaches require user-specified assertions. Earlier static verification approaches [49], [50] do not account for bugs outside of the forwarding configuration.

Control plane fuzzing tools have been proposed to identify the cause of bugs [74], generate test cases [46], [84], detect logical [58] and performance [59] vulnerabilities, identify operational inconsistencies [34], and bridge intent semantic gaps [52]. These tools complement CHIMERA since they can detect control plane bugs and vulnerabilities; CHIMERA focuses on multi-plane bugs and vulnerabilities.

Fuzzing is used for automated software testing. For efficient input generation, code coverage has been used to guide generation [1], [22], [26], [27], [36], [68], [70], [93], or more paths from the static analysis and concolic execution [30], [38], [54], [79], [92]. Fuzzing has been used for network protocol bugs [69], [78], [96].

10. Conclusion

We presented CHIMERA, a multi-plane P4 network fuzzing framework. We motivated the need to understand multi-plane inputs to discover bugs and vulnerabilities based on existing known bugs collected from bug reports in the P4 ecosystem. We designed and implemented CHIMERA to provide both the control and data plane inputs to fuzz P4 network infrastructure using concolic execution to generate a bug oracle, verify rules, and track P4 coverage. We proposed two new approaches in input mutations, leveraging inter-dependencies among multi-plane inputs and P4 programs. We evaluated CHIMERA using a representative P4 stack of several P4 components to uncover 7 new bugs, 3 of which were security vulnerabilities.

Acknowledgments

We thank the anonymous reviewers at IEEE S&P 2025 for their helpful comments and suggestions. This material is based upon work supported by the National Science Foundation through grants CNS-2145744 and CNS-2346499. The authors declare no competing interests.

References

[1] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[2] Onos security: Security-mode onos. https://wiki.onosproject.org/display/ONOS10/ONOS+Security%3A+Security-mode+ONOS, Feb 2015.

[3] ONOS Intent Framework. https://wiki.onosproject.org/display/ONOS/Intent+Framework, May 2016.

[4] P4 Integrated Network Stack (PINS). https://opennetworking.org/pins/, 2021.

[5] Behavioral model (bmv2). https://github.com/p4lang/behavioral-model, 2024.

[6] Behavioral model (bmv2) github issues. https://github.com/p4lang/behavioral-model/issues, 2024.

[7] eBPF. https://ebpf.io, 2024.

[8] Fabric-TNA. https://github.com/stratum/fabric-tna, 2024.

[9] Fabric-TNA github issues. https://github.com/stratum/fabric-tna/issues, 2024.

[10] Libpcap. https://www.tcpdump.org/, 2024.

[11] Onos issues. https://jira.onosproject.org/browse/ONOS, 2024.

[12] Orange – p4rt-ovs. https://p4.org/onf-product/orange-p4rt-ovs/, 2024.

[13] p4c: P4_16 reference compiler. https://github.com/p4lang/p4c, 2024.

[14] p4runtime-go-client. https://github.com/antoninbas/p4runtime-go-client/, 2024.

[15] Pi library repository. https://github.com/p4lang/PI, 2024.

[16] Pi library repository github issues. https://github.com/p4lang/PI/issues, 2024.

[17] SD-Fabric DHCP Relay Application. https://docs.sd-fabric.org/master/advanced/connectivity/dhcp-relay.html, 2024.

[18] Stratum: enabling the era of next generation SDN. https://opennetworking.org/stratum/, 2024.

[19] Stratum github issues. https://github.com/stratum/stratum/issues, 2024.

[20] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. P4fuzz: Compiler fuzzer fordependable programmable dataplanes. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, pages 16–25, 2021.

[21] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. Switchv: automated sdn switch validation with p4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 365–379, 2022.

[22] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[23] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[24] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 46–57. IEEE, 1996.

[25] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[28] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[29] Mingming Chen, Thomas La Porta, Teryl Taylor, Frederico Araujo, and Trent Jaeger. Manipulating openflow link discovery packet forwarding for topology poisoning. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3704–3718, 2024.

[30] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[31] Lori A Clarke and David S Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[32] Intel Corporation. Intel® intelligent fabric processors. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, 2024.

[33] LLVM Developers. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2024.

[34] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. AIM-SDN: Attacking information mismanagement in SDN-datastores. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 664–676, 2018.

[35] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free p4 programs. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 571–585, 2020.

[36] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[37] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[38] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[39] The P4.org API Working Group. P4Runtime Specification version 1.3.0. https://staging.p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html, Dec 2020.

[40] The P4.org API Working Group. P4 Portable NIC Architecture (PNA) version 0.7. https://staging.p4.org/p4-spec/docs/PNA-v0.7.html, Dec 2022.

[41] The P4.org Applications Working Group. In-band Network Telemetry (INT) Dataplane Specification version 2.1. https://p4.org/p4-spec/docs/INT_v2_1.pdf, Nov 2020.

[42] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pages 230–243, 2021.

[43] Marc R Hoffmann, B Janiczak, and E Mandrikov. Eclemma-jacoco java code coverage library. https://www.jacoco.org/jacoco/, 2011.

[44] IEEE Communications Society. Comcast: ONF Trellis software is in production together with L2/L3 white box switches. https://techblog.comsoc.org/2019/09/14/comcast-puts-onf-trellis-software-into-production/, September 2019.

[45] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[46] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. BEADS: Automated attack discovery in OpenFlow-based SDN systems. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 311–333. Springer, 2017.

[47] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.

[48] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 286–301, 2021.

[49] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[50] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[51] Jinwoo Kim, Minjae Seo, Eduard Marin, Seungsoo Lee, Jaehyun Nam, and Seungwon Shin. Ambusher: Exploring the security of distributed sdn controllers through protocol state fuzzing. *IEEE Transactions on Information Forensics and Security*, 2024.

[52] Jiwon Kim, Benjamin E. Ujcich, and Dave (Jing) Tian. Intender: Fuzzing intent-based networking with intent-state transition guidance. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[53] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[54] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. {CAB-Fuzz}: Practical concolic testing techniques for {COTS} operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, 2017.

[55] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. Tracking p4 program execution in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 117–122, 2020.

[56] K Shiv Kumar, Ranjitha K, PS Prashanth, Mina Tahmasbi Arashloo, Venkanna U, and Praveen Tammana. Dbval: Validating p4 data plane runtime behavior. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 122–134, 2021.

[57] Seungsoo Lee, Seungwon Woo, Jinwoo Kim, Vinod Yegneswaran, Phillip Porras, and Seungwon Shin. AudiSDN: Automated detection of network policy inconsistencies in software-defined networks. In *IEEE INFOCOM 2020*, pages 1788–1797, 2020.

[58] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip A Porras. DELTA: A security assessment framework for software-defined networks. In *Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[59] Ao Li, Rohan Padhye, and Vyas Sekar. Spider: A practical fuzzing framework to uncover stateful performance issues in sdn controllers. *arXiv preprint arXiv:2209.04026*, 2022.

[60] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[61] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498, 2023.

[62] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay Thakur, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A p4-based 5g user plane function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 162–168, 2021.

[63] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[64] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar 2008.

[65] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, 2014.

[66] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[67] Tomasz Osiński, Halina Tarasiuk, Paul Chaignon, and Mateusz Kossakowski. P4rt-ovs: Programming protocol-independent, runtime extensions for open vswitch with p4. In *2020 IFIP Networking Conference (Networking)*, pages 413–421. IEEE, 2020.

[68] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, pages 329–340, 2019.

[69] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[70] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[71] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. Hydra: Effective runtime network verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 182–194, 2023.

[72] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. P4testgen: An extensible test oracle for p4-16. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 136–151, 2023.

[73] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 683–699, 2020.

[74] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, et al. Troubleshooting blackbox sdn control software with minimal causal sequences. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 395–406, 2014.

[75] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[76] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. Fix with p6: Verifying programmable switches at runtime. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[77] Apoorv Shukla, S Jawad Saidi, Stefan Schmid, Marco Canini, Thomas Zinner, and Anja Feldmann. Toward consistent sdns: A case for network state fuzzing. *IEEE Transactions on Network and Service Management*, 17(2):668–681, 2019.

[78] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[79] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[80] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 17–32, 2021.

[81] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 648–663, 2018.

[82] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Adam Bates, William H Sanders, and Hamed Okhravi. Causal analysis for {Software-Defined} networking attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3183–3200, 2021.

[83] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Steven R Gomez, Adam Bates, William H Sanders, and Hamed Okhravi. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

[84] Benjamin E Ujcich, Uttam Thakore, and William H Sanders. Attain: An attack injection framework for software-defined networking. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 567–578. IEEE, 2017.

[85] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[86] Open vSwitch. Open vSwitch Extensions. https://docs.openvswitch.org/en/latest/topics/ovs-extensions.

[87] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, pages 122–135, 2017.

[88] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. Unexpected data dependency creation and chaining: A new attack to sdn. In *IEEE Symposium on Security and Privacy*, pages 264–278, 2020.

[89] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, et al. Unleashing smartnic packet processing performance in p4. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1028–1042, 2023.

[90] Nofel Yaseen, Liangcheng Yu, Caleb Stanford, Ryan Beckett, and Vincent Liu. Fp4: Line-rate greybox fuzz testing for p4 switches. *arXiv preprint arXiv:2207.13147*, 2022.

[91] Changhoon Yoon, Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Heedo Kang, Martin Fong, Brian O'Connor, and Thomas Vachuska. A security-mode for carrier-grade sdn controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 461–473, 2017.

[92] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[93] Michal Zalewski. American fuzzy lop (afl). http://lcamtuf.coredump.cx/afl, 2017.

[94] Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 19–33, 2016.

[95] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. Meissa: scalable network testing for programmable data planes. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 350–364, 2022.

[96] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

```
state parse_ipv4 {
    packet.extract(hdr.ingress.ipv4);
    fabric_md.ingress.routing_ipv4_dst = hdr.ingress.
        ipv4.dst_addr;
    fabric_md.ingress.bridged.base.ip_eth_type =
        ETHERTYPE_IPV4;                    // 2048
    transition select(hdr.ingress.ipv4.protocol) {
        PROTO_TCP: parse_tcp;              // 6
        PROTO_UDP: parse_udp;              // 17
        PROTO_ICMP: parse_icmp;            // 1
        default: accept;
    }
}
```

(a) Ethernet type parser state code in `fabric.p4`

```
private void parse_ipv4() {
    pos += hdr._ingress_ipv4.getBitLength();  // Extract
        hdr._ingress_ipv4
    hdr._ingress_ipv4.isValid = true;
    fabric_md._ingress_routing_ipv4_dst18 = hdr.
        _ingress_ipv4.dst_addr;
    fabric_md._ingress_bridged4._base_ip_eth_type10.
        putValue(2048);
    List<String> sucStates = List.of(
            "parse_tcp",
            "parse_udp",
            "parse_icmp",
            "accept");
    int caseNum = -1;
    if (stateItr != null && stateItr.hasNext()) {
        caseNum = sucStates.indexOf(stateItr.next());
    }
    if (caseNum < 0) {
        caseNum = rand.nextInt(4);
    }
    if (caseNum == 0) {          // parser_tcp
        hdr._ingress_ipv4.protocol.putValue(6);
        parse_tcp();
    } else if (caseNum == 1) {  // parser_udp
        hdr._ingress_ipv4.protocol.putValue(17);
        parse_udp();
    } else if (caseNum == 2) {  // parser_icmp
        hdr._ingress_ipv4.protocol.putValue(1);
        parse_icmp();
    } else {                     // accept
        hdr._ingress_ipv4.protocol.putRandom();
        accept();
    }
}
```

(b) Translated code in `FabricPktMutator.java`

Figure 11: Example of the result of PAPM translator.

Appendix A. Supplementary Design Details

A.1. Parser-Aware Packet Mutation (PAPM)

Figure 11 shows an example of the P4 parser state code (`fabric_v1model.p4` in Table 4) and translated code.

A.2. Multi-Plane Packet Verification

CHIMERA verifies the expected output on both planes. In the control plane, the *network manager* communicates with the application agent running on the controller to send `PACKET_OUT` messages or receive `PACKET_IN` messages. In the data plane, CHIMERA executes the *test agent* that

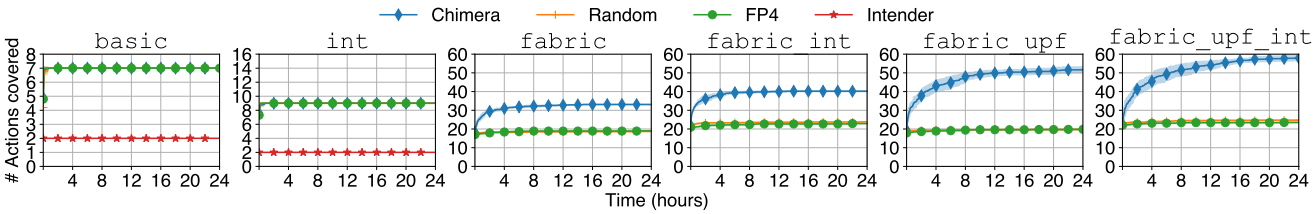


Figure 12: Covered P4 actions measured by different fuzzers with six P4 programs for 24 hours.

manages the emulated network, including the *host agent* in each host. To check the expected output received from P4CE, the test agent sends `SNIFF` to a destination and `SEND` to a source. On receiving the `SNIFF` message, the host agent listens to packets on the expected egress switchport to receive the expected packet. The source host agent then sends the packet to the ingress switchport. To detect a packet drop, the *timeout checker* starts the timer after confirming all `SEND` and `SNIFF` requests for each packet test. The timeout checker waits for a response from the host agent, sniffing a packet until a configurable timeout value (100ms by default). To prevent timeout, the sniffing host agent executes multiple threads: one for storing packets in a queue and the other for comparing received packets to the expected output packet. When the host agent or application agent at the destination receives the correct packet, the agent returns the success message to the test agent; otherwise, a timeout is sent.

Appendix B. Extended Evaluation

B.1. P4 Action Coverage

Figure 12 shows the P4 action coverage while running CHIMERA, Random, and FP4. CHIMERA does not leverage the P4 action coverage as feedback compared to FP4. Intender installs P4 rules translated from reachability intents and covers only two actions for both `basic` and `int`: `set_egress_port()` and `drop()`.

The number of P4 actions covered by CHIMERA is *greater than or equal to* the ones covered by FP4. For simple P4 programs such as `basic` and `int`, CHIMERA and FP4 show similar P4 action coverage. CHIMERA shows 1.74×, 1.76×, 2.62×, and 2.64× higher P4 action coverage than FP4 for `fabric`, `fabric_int`, `fabric_upf`, and `fabric_upf_int`, respectively. While FP4 relies on a set of flow rules given by a seed input, CHIMERA can create new actions and generate packets matched to mutant rules. Even though FP4 is implemented on a programmable hardware switch with high performance, covered P4 actions can be smaller without mutating control plane rules. Random mutations for both control plane rules and data plane packets do not help increase P4 action coverage. Since CHIMERA can execute more diverse P4 actions, CHIMERA can achieve the highest coverage in both control and data planes.

B.2. False Positive Analysis

The data plane verification shows a 1.2% false positive rate (FPR), mainly caused by unimplemented features of P4CE. First, P4CE miscalculates overwritten rules, causing 0.48% FPR. P4CE can check whether a new rule has the same priority and identical match fields but different actions to calculate the expected behavior with such rules. Second, P4CE does not support multiple outputs and represents such cases as drops, causing 0.65% FPR. `fabric_int` can generate an output packet and an INT report of the ingress packet. To support packet cloning, P4CE can have multiple packet contexts. When cloning the packet, P4CE can copy the current execution state, track the P4 program path, and keep metadata. P4CE can separately execute each copy from the following line after the clone. Once all copies are finished, P4CE can collect the output packets. The remaining false positives (0.07% FPR) are caused by non-deterministic behavior, such as timestamp-based match fields.

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper presents a novel multi-plane fuzzing framework that detects bugs and vulnerabilities in P4 network infrastructures by simultaneously testing both the control and data planes.

C.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Identifies an Impactful Vulnerability
- Creates a New Tool to Enable Future Science

C.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field. Most P4 security research focuses on the data plane, but many vulnerabilities arise from its interaction with the control plane. As a result, the authors propose a fuzzing framework that tests both planes simultaneously for more comprehensive bug detection.
2) This paper identifies impactful vulnerabilities in P4 network infrastructures. By applying the proposed system to three common P4-based network infrastructures, the authors have found 7 new bugs, including 3 security-critical vulnerabilities, 2 bugs triggered by multi-plane inputs, and 2 cross-plane bugs.
3) This paper creates a new tool to enable future science. The authors plan to open source their system, enabling other researchers to leverage it for further research.

C.4. Noteworthy Concerns

1) The authors apply their techniques only to software P4 targets, leaving it unclear how these methods would perform on hardware P4 devices or what bugs and vulnerabilities could be uncovered on hardware.
2) The authors use the concolic execution of the P4 program as a bug oracle. Some reviewers have concerns about its validity and effectiveness in accurately detecting bugs and vulnerabilities.