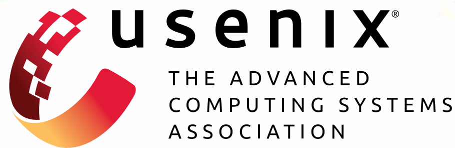


INTENDER: Fuzzing Intent-Based Networking with Intent-State Transition Guidance

Jiwon Kim, *Purdue University;* Benjamin E. Ujcich, *Georgetown University;*
Dave (Jing) Tian, *Purdue University*

https://www.usenix.org/conference/usenixsecurity23/presentation/kim-jiwon

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

INTENDER: Fuzzing Intent-Based Networking with Intent-State Transition Guidance

Jiwon Kim
Purdue University

Benjamin E. Ujcich
Georgetown University

Dave (Jing) Tian
Purdue University

Abstract

Intent-based networking (IBN) abstracts network configuration complexity from network operators by focusing on *what* operators want the network to do rather than *how* such configuration should be implemented. While such abstraction eases network management challenges, little attention to date has focused on IBN's new security concerns that adversely impact an entire network's correct operation. To motivate the prevalence of such security concerns, we systematize IBN's security challenges by studying existing bug reports from a representative IBN implementation within the ONOS network operating system. We find that 61% of IBN-related bugs are *semantic bugs* that are challenging, if not impossible, to detect efficiently by state-of-the-art vulnerability discovery tools.

To tackle existing limitations, we present INTENDER, the first semantically-aware fuzzing framework for IBN. INTENDER leverages network topology information and intent-operation dependencies (IOD) to efficiently generate testing inputs. INTENDER introduces a new feedback mechanism, intent-state transition guidance (ISTG), which traces the history of transitions in intent states. We evaluate INTENDER using ONOS and find 12 bugs, 11 of which were CVE-assigned security-critical vulnerabilities affecting network-wide control plane integrity and availability. Compared to state-of-the-art fuzzing tools AFL, Jazzer, Zest, and PAZZ, INTENDER generates up to 78.7× more valid fuzzing input, achieves up to 2.2× better coverage, and detects up to 82.6× more unique errors. INTENDER with IOD reduces 73.02% of redundant operations and spends 10.74% more time on valid operations. INTENDER with ISTG leads to 1.8× more intent-state transitions compared to code-coverage guidance.

1 Introduction

Modern network paradigms, such as software-defined networking (SDN), purport to solve the complexity of network management by opening up a network's control and data planes through programmable interfaces. While this flexibility has revolutionized tailored capabilities in domains ranging from telecommunication to cloud providers (among others), the same flexibility still requires network operators to understand their networks' low-level protocols and policy rules—but now with increasing complexity costs. Moreover, such complexity introduces additional barriers towards mapping higher-level objectives (*e.g.,* business requirements, organizational security policies) to *what* the network is doing and *how* such configuration is implemented.

Intent-based networking (IBN) reduces the semantic gap between high-level objectives and network implementation, while taming complexity costs through additional abstraction [19, 41]. IBN allows network operators to manage the network without needing to understand the implementation details of the network's device interfaces, protocols, or topology. Unlike traditional network management that requires static configuration, IBN operators declaratively specify their intent about *what* they want the network to do within a set of constraints (*e.g.,* reachability between nodes). The network's intent system compiles such intents into a low-level configuration of *how* to achieve the intents based on the underlying network's implementation.

Although the burgeoning IBN paradigm has moved towards standardization [9,19,41], open source projects [3,5,7], industry [1,2,23], and academic works [20,58], very little attention has been paid towards the *security* of IBN itself [19].

In this paper, we start with investigating the security perspective of current IBN implementations. In particular, we perform a case study analysis on the IBN implementation of the enterprise-grade ONOS network operating system [13], which is the basis for proprietary network operating systems used in production environments [29]. Our findings are twofold. First, among 2,233 bugs reported about ONOS, 186 of all bugs (8%) are related to IBN. Second, among all 186 IBN-related bugs, 114 of such bugs (61%) are *semantic bugs*. Unlike syntactic bugs that usually lead to memory corruption, semantic bugs require domain knowledge and often do not cause program crashes. Thus, semantic bugs cannot be easily detected by off-the-shelf vulnerability discovery tools. Our experiment with AFL shows that only 0.14% of error cases are related to

semantic bugs.

To tackle this challenge, we present INTENDER, the first semantically-aware black-box fuzzing framework for IBN. Unlike the state-of-the-art fuzzing tools that leverage code coverage for feedback, we introduce intent-state transition guidance (ISTG) as a new feedback mechanism. ISTG does not require source-level or binary-level instrumentation (as code coverage does), and it captures the unique characteristics of IBN implementations through historical tracing of transitions among intent states. To generate fuzzing input efficiently, INTENDER applies a topology-aware input generation that understands the constraints imposed by the current network topology to increase the validity of each fuzzing input. Furthermore, INTENDER utilizes intent-operation dependency analysis to reduce redundant operations during fuzzing.

We evaluate the efficacy of INTENDER on ONOS, which provides a complex and representative IBN implementation that has seen real-world adoption by telecommunications providers such as Comcast [29]. We discovered 12 bugs, 11 of which were security-critical vulnerabilities assigned with 11 CVEs affecting network-wide control plane integrity and availability. Compared with state-of-the-art fuzzing tools AFL, Jazzer, Zest, and PAZZ, INTENDER generates up to 121.3× more valid fuzzing input, achieves up to 2.2× better coverage, and detects up to 82.6× more unique errors. Compared to baselines, INTENDER with IOD reduces 73.02% of redundant operations and spends 10.74% more time on valid operations. INTENDER with ISTG leads to 1.8× more intent-state transitions compared to code-coverage guidance (CCG).

In summary, our contributions are as follows:

- We conduct a systematization of existing IBN bugs within ONOS to understand the representative scope of IBN vulnerabilities, and we find that 61% of IBN-related bugs are *semantic bugs* that existing state-of-the-art vulnerability tools fall short in discovering.
- We present a black-box fuzzing technique that detects IBN semantic bugs, which combines topology-aware input generation, intent-operation dependency analysis, and an intent-state transition guidance mechanism.
- We design and implement the INTENDER framework, which includes a black-box IBN fuzzer architecture. We further adopt 4 state-of-the-art fuzzing techniques within the INTENDER framework: AFL, Jazzer, Zest, and PAZZ.
- We apply INTENDER to the ONOS IBN implementation and find 12 previously-unknown bugs with 11 CVEs assigned. Compared to the state-of-the-art fuzzing tools, INTENDER generates up to 121.3× more valid fuzzing input, achieves up to 2.2× better coverage, and detects up to 82.6× more unique errors.

2 Background

As network management becomes more complex, IBN offers human-centric and workflow-centric abstractions to ease the

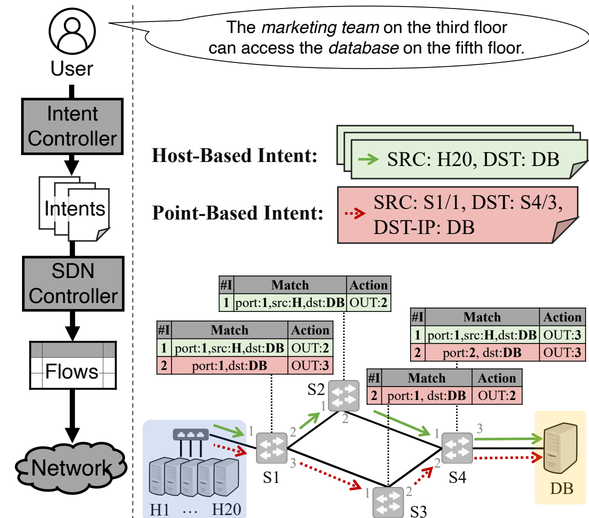


Figure 1: Overview of IBN. The marketing team has hosts $H_1 - H_{20}$. The user's intent can be represented as either host-based intents with the green solid path or a point-based intent with the red dotted path.

implementation burden. IBN enables practitioners to manage the network by defining *what* goals should be achieved, instead of unnecessarily focusing on *how* to implement such goals. Figure 1 shows the overall view of IBN. Practitioners declaratively specify their goals through *intents*, which detail any path reachability that must be met (*e.g.,* host-based or point-based). Intents are compiled into implementation-specific forwarding rules, device interfaces, and network topologies. IBN stands at the forefront of next-generation networks that incorporate machine learning (ML) and natural language processing (NLP) techniques into closed-loop autonomous networks [11, 14, 31].

IBN has been realized through standardization activities by the Open Networking Foundation (ONF) [41], the Internet Research Task Force (IRTF) [19], and the 3rd Generation Partnership Project (3GPP) [9], among others [44]. Among open-source network operating systems and network orchestration tools, OpenDayLight (ODL) [39], Open Networking Operating System (ONOS) [13], and Open Network Automation Platform (ONAP) [25] all implement IBN [3, 5, 7]. Although differences occur among projects (*e.g.,* natural language or declarative-based input), all of the projects have a common reference model of IBN that translates and activates high-level intents into the network.

Each intent in IBN includes an *intent lifecycle* that is managed by a centralized *intent controller*.[1] A practitioner submits their intent to the controller, which is then validated, translated, compiled, and installed into the network. The controller continuously observes any events that change the network configuration (*e.g.,* topology change) and re-verifies that ex-

[1] In programmable networks, the intent controller may be realized as a subsystem of a network operating system (NOS) or a software-defined networking (SDN) controller.

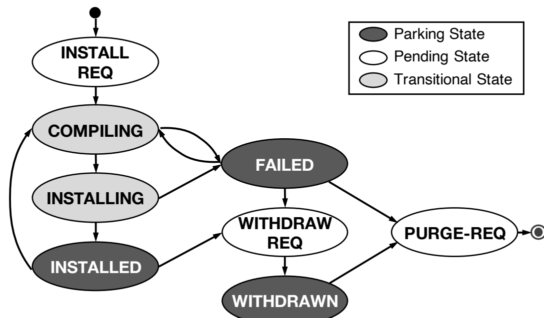


Figure 2: ONOS intent state machine. The state machine includes all intent states and transitions.

isting intents' specifications are still satisfied. As a result, an intent exists in a specific lifecycle stage at any given time. We refer to each stage as an *intent state* and the set of all possible states and transitions as the *intent state machine*.

Figure 2 shows a representative intent state machine implemented by ONOS [13]. We use ONOS as a representative example because it is based on a well-developed technical recommendation from ONF [41], but we note that the conceptual model generalizes to any IBN implementation, such as ODL [3].

Figure 3 shows how the operations map to a series of state transitions within the state machine: A user can add a new intent or modify an existing intent by requesting `submit` (`INSTALL_REQ`). The controller attempts to translate and compile the intent into an installable intent (`COMPILING`). After compilation, the controller attempts to install the intents into network devices (`INSTALLING`). Upon success, the intent is considered installed (`INSTALLED`). If failures occur at any point, the intent is considered failed (`FAILED`). A user can withdraw an *active intent* that has either been installed or has failed (`WITHDRAW_REQ`). Once completed, the intent is considered withdrawn (`WITHDRAWN`). A user can also remove an intent completely from the controller (`PURGE_REQ`). If the network topology changes, any active intents will return to recompilation (`INSTALL_REQ`, `INSTALLING`, and `INSTALLED`).

3 Security Vulnerabilities in IBN

IBN's abstraction features enable practitioners to efficiently reason about network activities without having to worry about the underlying protocol, interface, or forwarding rule implementation. At the same time, though, this abstraction creates new security challenges that affect the overall security posture of IBN.

We explore the space of security vulnerabilities of IBN through an extensive case study analysis of nearly 400 existing bugs that have been reported about the ONOS IBN implementation. We systematize those vulnerabilities according to impact, and we find that most are logic (or semantic) bugs specific to the IBN domain, which complicate their discovery (§ 3.1). We demonstrate the severity of such vulnerabilities

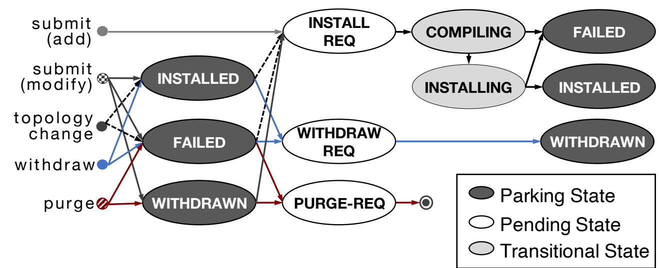


Figure 3: Operations (left side) and their respective intent-state transitions in the ONOS intent state machine. Such transitions are possible paths in the intent state machine.

through a representative example (CVE-2022-24109), along with the limitations of existing fuzzing tools used for discovery (§ 3.2). We outline the limitations and challenges of automated, efficient discovery, which underlie our motivation for creating INTENDER (§ 3.3).

3.1 Systematization of IBN Bugs

As a starting point for uncovering new IBN-related bugs, we first explore previously reported IBN bugs. For our data set, we looked at the 2,233 publicly-available bug reports that have been submitted regarding ONOS [8]. We searched for all reports that mentioned "intent" and found 394 bug reports (17% of total ONOS bug reports). We excluded 119 bug reports that involved issues caused by other system parts (*e.g.,* GUI interface), 70 bug reports that involved auxiliary functionalities (*e.g.,* cluster synchronization, debugging, and testing), and 19 bug reports that were duplicates or not bugs. After filtering, we studied the remaining 186 bug reports (47% of "intent" bug reports; 8% of total ONOS bug reports).

Approach. We classify the 186 bugs into 12 categories according to their impact on the controller, as shown in Table 1. We define *syntactic bugs* as bugs that are caused by syntactically invalid inputs or bugs that cause obvious undesired behavior (*e.g.,* controller shutdown). We define *semantic bugs* as those that are syntactically valid but whose behaviors depend on the logic and semantics of the IBN domain.

For syntactic bugs, the controller does not handle inputs correctly when bugs occur. The controller can respond with an internal server error, or deny correct input due to unimplemented features (**SYN1**). An intent can disappear from the controller due to the internal error (**SYN2**), can have data different from what a user requests (**SYN3**), or can be corrupted leading to a `CORRUPT` state (**SYN4**). IBN bugs can result in other failures that can happen in any program. The controller can crash (**SYN5**), or CPU or memory resources can be exhausted (**SYN6**). Within the network, the topology can be disbanded (**SYN7**) or the throughput can decrease (**SYN8**).

[2]Among 186 bugs, 8 bugs have multiple outcomes at the same time. For example in ONOS-1409, when hosts are reconnected, the relevant intent remains in the `INSTALLED` state. Also, the connectivity of two hosts fails with PathNotFoundException in the controller [4].

Table 1: Bug analysis of the ONOS intent subsystem.

Type	Code	Impact	Detection Mechanism	# Bugs Based on Intent Operation of Root Cause				
				submit	withdraw	purge	topo-change	Total
Intent syntactic bug	**SYN1**	Denied intent request	Input validation	2	-	-	-	2
	SYN2	Not found	Input validation	9	-	-	-	9
	SYN3	Wrong intent data	Input validation	3	-	-	-	3
	SYN4	Corrupt intent	Input validation	1	-	-	-	1
Other syntactic bug	**SYN5**	Controller shutdown	Application agent	1	-	-	-	1
	SYN6	Resource exhaustion	Resource agent	1	2	-	-	3
	SYN7	Topology disband	Application agent	2	1	-	-	3
	SYN8	Throughput drop	Performance test	15	-	-	1	16
	SYN9	Exceptions	Log detection	29	6	2	5	42
Intent semantic bug	**SEM1**	Inconsistent intent state	Control-Plane (CP) test	17	10	7	16	50
	SEM2	Failure in connectivity	Data-Plane (DP) test	29	2	-	14	45
	SEM3	Impact on existing intent	CP/DP tests	6	2	-	-	8
	SEM4	Garbage flow rules	Flow-Intent mapping	6	4	-	1	11
Total				121	27	9	37	194 (186[2])

Program exceptions can also occur (**SYN9**). These syntactic bugs are easily found by existing tools or verifying other resources (*e.g.,* CPU and memory utilization) because incorrect behavior is not dependent on program logic.

For semantic bugs, the intent violates the underlying network's integrity, although there may be no immediate error in the controller. For example, when these bugs occur, the intent state can be inconsistent with the network topology, processing time,[3] and operations[4] (**SEM1**). Even if the intent state is correct relative to the topology, the connectivity for the intent can show as failed (**SEM2**). Installing or deleting an intent can affect the existing intents by disturbing the connectivity (**SEM3**). Bugs can generate unnecessary flow rules or leave behind flow rules after removing intents (**SEM4**). We categorize these bugs as semantic bugs, as these bugs' impacts are harder to recognize without considering IBN semantics.

Results. Overall, we found that a majority (61%) of bugs are semantic bugs. Given the difficulty of efficiently and automatically finding semantic bugs with off-the-shelf software bug and vulnerability discovery tools, we argue in favor of a tool that is tailored to the IBN domain's semantics and we motivate the need for designing such a tool.

We also note that the detection mechanisms in Table 1 show that semantic bugs require domain-specific verification: control plane and data plane verification. The lack of both mechanisms in off-the-shelf software bug-finding tools motivates our interest in a comprehensive vulnerability discovery framework that incorporates the network.

3.2 Motivating Attack Example

Our systematization from Section 3.1 demonstrates the need for a semantically-aware vulnerability discovery tool for IBN.

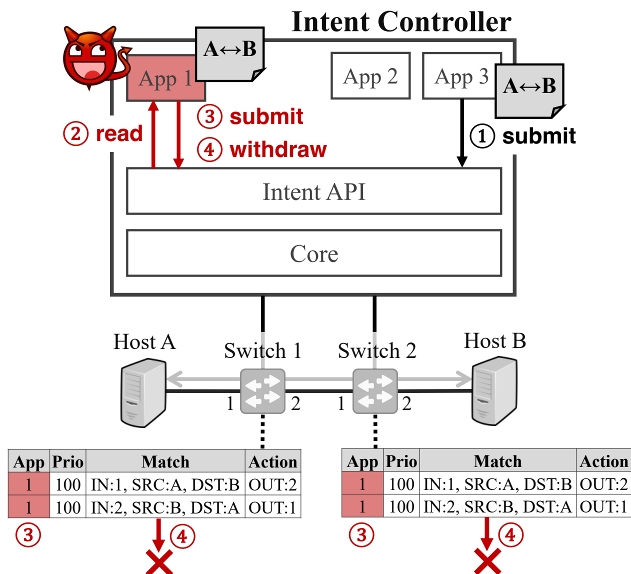


Figure 4: Overview of CVE-2022-24109 attack scenario described in Section 3.2. The attacker executes steps 2–4 to eradicate flow rules of the normal intent.

To motivate the broad-reaching security impacts of IBN on network functionality and to show why the understanding of IBN's security posture is crucial, we now demonstrate the severe consequences that can be exploited from intent semantic bugs by attackers. We use a representative semantic vulnerability that we discovered (CVE-2022-24109).[5] We also outline the limitations of existing state-of-the-art approaches that would have made discovery difficult or impossible.

Attack scenario. Figure 4 shows the exploit workflow. An attacker in the control plane (*e.g.,* an application or a REST client) accesses the internal intent framework. When a normal application submits an intent to provide connectivity between Host A and Host B ①, the attacker can read the installed intent ② and submit a new equivalent intent with the same

[3] The intent can persist in any transitional state, such as `INSTALLING` and `WITHDRAWING`.

[4] For example in ONOS-381, when removing an installed intent, the intent is stuck in the `INSTALLED` state [6].

[5] We discuss the responsible disclosure process in Section 7.1.

priority but a different key ③. In this step, while the intent framework has two equivalent intents, there is only one set of flow rules, since flow rules of the normal intent are overwritten by those of the attacker's intent. Thus, the attacker can eliminate these flow rules by deleting the second intent ④. Even though the intent submitted by the normal application remains in the system, all related flow rules are removed.

As shown in this example, IBN introduces new security challenges compared to traditional networking. When new intents are translated into flow rules in the network, existing flow rules can be overwritten or disturbed by others. Adversaries can exploit the vulnerabilities with intents to affect or even manipulate the network traffic within the system. In addition, new components for IBN, such as the intent store and the intent state machine, inevitably introduce a new attack surface for adversaries.

In order to discover unknown bugs in IBN, we could use existing tools, such as off-the-shelf fuzzers. However, there are many limitations in existing techniques that preclude discovery, which we illuminate through the motivating example.

Limitations in existing tools. Existing fuzzing tools input a random sequence into a target program and check whether the program behaves incorrectly. Due to the lack of semantic knowledge about IBN, however, such tools might only find basic bugs related to input verification. For instance, to reproduce the motivating example, the vulnerability requires two constraints: `INSTALLED` intents and a `withdraw` operation. For any intent to be `INSTALLED`, an input should have included the existing source and existing destination in the network. However, successfully finding any existing host based on random values is unlikely and requires a substantial input space search; such inefficiency wastes valuable testing time. Without prior knowledge of varying network information such as device ID, host ID, and location of hosts, it is hard to generate even one `INSTALLED` intent. Moreover, we cannot apply the topology information as a fixed grammar in fuzzing tools, as the topology differs in every environment.

The intent state that triggers the vulnerability is `WITHDRAWN`, which requires that the fuzzer executes the `withdraw` operation. While existing fuzzing tools can simply request `withdraw` with the random intent ID, it might be easily rejected by the controller since a random ID is very unlikely to match one of the existing intents.

Limitation in existing algorithms. To support IBN fuzzing, existing fuzzing tools can be configured with a post-processor that uses the underlying network to generate valid intents by communicating with a controller. Even with a post-processor, though, such existing tools may have difficulties in discovering unknown bugs because of their limited guidance policies. Well-known fuzzers, such as AFL [62] and IJON [12], attempt to cover more program code (or states) based on the state machine of a target program during the execution. If the code coverage cannot distinguish bug cases from normal cases, such tools do not aid in the detection of such bugs.

Table 2: Code coverage of the motivating example with no distinction between normal and error cases.

Package	Classes	Branch Coverage	
		Normal	Error
Intent	`org/onosproject/net/intent/impl/phase`	28%	28%
	`org/onosproject/store/intent/impl`	20%	20%
	`org/onosproject/net/intent/impl`	19%	19%
	`org/onosproject/net/intent/impl/installer`	10%	10%
	`org/onosproject/net/intent/impl/compiler`	8%	8%
Network	`org/onosproject/net/driver/impl`	25%	25%
	`org/onosproject/net/flow/impl`	11%	11%
	`org/onosproject/net/domain/impl`	4%	4%
Controller	`org/onosproject/openflow/controller/impl`	5%	5%
	`org/onosproject/openflow/controller/driver`	5%	5%
	`org/onosproject/openflow/controller`	3%	3%

Table 2 shows the code coverage of the motivating example compared to one of a normal case, where the user removes the existing intent before adding its copy. Given the large number of classes within ONOS, we select packages related to intent operations only. From Table 2, it is clear that code coverage cannot differentiate the error case from the normal case. Except for the flow rules in the network devices that are overwritten by the bug case, both the normal and error cases have the same number of intent operations (two additions and one deletion) and flow rule operations. Because the controller does not have any capability to resolve duplicated intents and flow rules, the covered code will be the same in both cases.

3.3 Challenges of IBN Fuzzing

Based on our bug report analysis and motivating example, we find that intent semantic bugs are difficult for existing fuzzers and fuzzing approaches to uncover. We now generalize several challenges in finding intent semantic bugs, which we use as the basis for designing a new fuzzer for IBN.

Challenge (C1): Topology awareness. Generation of any installed intent is complicated without knowing the underlying network topology information. Existing fuzzers focus on a program's code to generate inputs that are likely to cause a bug. To increase efficiency, such fuzzers utilize the code coverage metric as feedback or apply a user-defined grammar to make valid inputs. In IBN, an intent that is specified based on reachability requires both a source ID and a destination ID of existing network objects (*e.g.,* host addresses). Given valid inputs in the grammar of the intent request, an existing fuzzer is unlikely to generate an installed intent, as any random identifier is unlikely to match one of the existing objects (*e.g.,* host addresses). Thus, an IBN fuzzer needs topology awareness to efficiently guide the input process.

Challenge (C2): Support for diverse operations. Intent bugs can occur in any operation, as shown in the right-sided columns of Table 1. In requesting random intents, an IBN fuzzer needs to execute other operations on existing intents or mutate a topology to discover more bugs. Specifically, such an IBN fuzzer needs to manage a network controlled by the target controller in order to mutate a topology at a specific

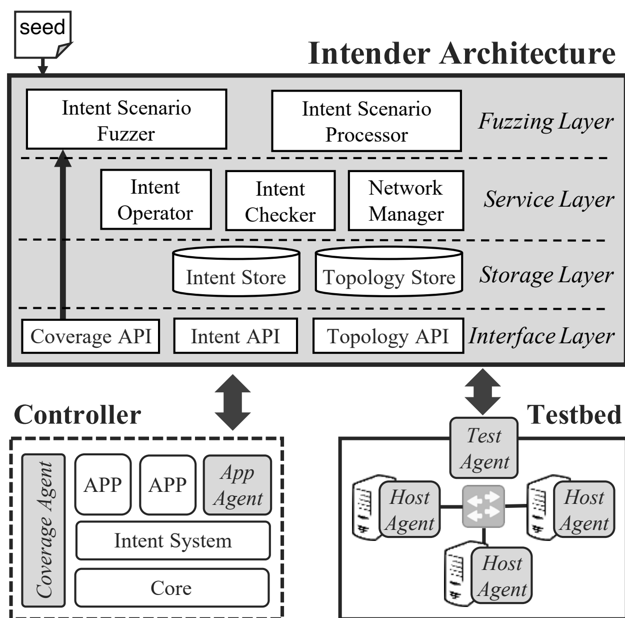


Figure 5: Overview of INTENDER framework. New components are highlighted in gray.

time.

Challenge (C3): Need for new guidance metric. As shown in the motivating example (and in particular, Table 2), the traditional code coverage metric used in existing fuzzers cannot distinguish errors from normal cases. Installing several intents that share the underlying path has the same coverage in the program code regardless of the number of intents. However, the number of intents is semantically important, since more intents can be changed while mutating the network topology. Thus, an IBN fuzzer needs a new guidance metric to generate diverse and meaningful inputs for IBN.

Challenge (C4): Need for black-box approach. Although open-source IBN implementations can be studied more easily with existing code-coverage-based approaches, the utility of analysis with code coverage is significantly curtailed when testing proprietary, closed-source IBN implementations. Thus, an IBN fuzzer should enable a black-box approach whose metrics are agnostic to the underlying program code.

Challenge (C5): Lack of detection mechanisms. A violation of intent semantics, which we discovered to be the majority cause of ONOS intent bugs, is hard to automatically recognize without additional domain knowledge. An intent's state is determined by two factors: network topology and previous operations. For instance, if an intent is active and successful, its state will be `INSTALLED`. When the topology changes because of a link failure or a link recovery, existing intents' states will change accordingly. Although the intent's state is correctly represented, the intent's *actual* connectivity—implemented by forwarding rules in the data plane—may not be satisfied because of unknown errors. To detect an intent state's correctness, additional detection mechanisms, such as the control plane and data plane verification, are necessary.

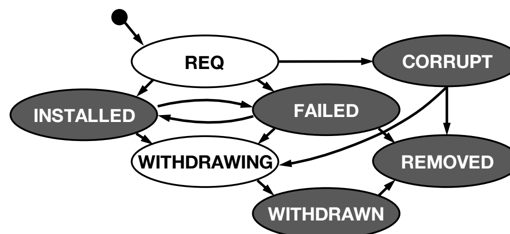


Figure 6: INTENDER state machine, which includes all intent states and transitions.

Table 3: INTENDER operations.

Operation	Input State	Output State
`add-intent`	-	`INSTALLED`, `FAILED`, `CORRUPT`
`mod-intent`	any state except `REMOVED`	`INSTALLED`, `FAILED`, `CORRUPT`
`withdraw-intent`	`INSTALLED`, `FAILED`, `CORRUPT`	`WITHDRAWN`
`purge-intent`	`FAILED`, `CORRUPT`, `WITHDRAWN`	`REMOVED`
`topology-change`	`INSTALLED`, `FAILED`	`INSTALLED`, `FAILED`

4 INTENDER Overview

Based on the aforementioned challenges in Section 3.3, we present INTENDER, a fuzzing framework for IBN that discovers unknown security vulnerabilities in IBN implementations. Figure 5 shows the framework overview.

We devise new guidance and input mutation policies for IBN fuzzing (§ 5.2). INTENDER receives seed scenarios that consist of a list of diverse operations written by domain experts (**C2**). During execution, INTENDER mutates each scenario in the seed corpus by appending or truncating operations. INTENDER generates a random intent based on the underlying topology (**C1**), and it uses the concept of *intent-operation dependency* to reduce the number of unnecessary operations in the mutant scenario and to increase efficiency. INTENDER also collects the history of transitions in intent states from the controller; if the scenario generates a unique history, it stores this mutant scenario into the corpus and keeps running until it is terminated (**C3**).

We construct the INTENDER architecture to apply our fuzzing techniques (§ 5.3). INTENDER communicates with the controller using the controller's API (**C4**) to request and receive topology and intent information. INTENDER uses a network testbed based on network emulation to mutate the network topology at specific times (**C2**), and it provides both control plane and data plane verification mechanisms (**C5**).

5 Design

We now present the intent model used by INTENDER (§ 5.1), the new fuzzing techniques that INTENDER incorporates (§ 5.2), and the INTENDER architecture (§ 5.3).

5.1 Intent Model

Given that an IBN fuzzer should work across different IBN implementations (**C4**), we define the *intent model* that allows

for generalization and abstraction across implementations. Figure 6 shows the state machine of intent used in INTENDER. The INTENDER intent state machine contains 7 states: `REQ` and `WITHDRAWING` for waiting states; and `INSTALLED`, `FAILED`, `CORRUPT`, `WITHDRAWN`, and `REMOVED` for final states. Table 3 shows five operations supported by INTENDER and state transitions in accordance with each operation.

5.2 New Fuzzing Techniques

In order to discover unknown semantic vulnerabilities in IBN, we design new fuzzing techniques. Compared to existing approaches, these generation and guidance policies increase the efficiency of finding semantic bugs.

Topology-Aware Intent Generation (TAIG). An IBN controller receives an intent as an input, and a fuzzer can generate random strings or even grammatically correct inputs to find unknown bugs. Without the consideration of topology, it is unlikely that the intent can ever reach the `INSTALLED` state; that makes it hard to find semantic bugs **(C1)**.

INTENDER checks topology information to create valid intents. It gathers topology information, builds a topology graph, and randomly chooses sources and destinations based on this graph. In addition, INTENDER can generate a scenario with multiple operations at the same time to test diverse cases. While generating operations, such as intent or topology operations, INTENDER reads or updates the configurational topology graph. During the scenario execution, it updates the operational store by receiving messages from the controller and verifies each operation.

Multiple-operation scenario mutation. As we have shown from our case study of existing bugs in Table 1, intent bugs can happen in any operation **(C2)**. By supporting all intent operations, fuzzers can find more IBN-related bugs. Although bugs that occur during installation can be discovered by requesting a random string or a topology-aware input, other operations require existing intents.[6] In order to cover these cases, the random order of operations as well as the random input is required.

For this purpose, INTENDER considers a set of operations as an input scenario. Users input well-written scenarios into INTENDER as *seeds*. When INTENDER mutates a seed scenario, it appends or truncates random operations at the end of each scenario; changing an operation in the middle will affect all other subsequent operations. In addition, since the intent's identifier is assigned by the controller, each `add-intent` operation has an internal intent identifier that is indicated by subsequent operations.

Intent-Operation Dependency (IOD). According to the state transitions from INTENDER operations in Table 3, dependencies exist among states. A naive fuzzer that generates

[6] For instance, `withdraw` and `purge` require an existing intent's identifier in order to not be simply rejected by the controller. In addition, changing topology without any existing intent will not invoke any IBN function.

Algorithm 1 The Intent-State Transition Guidance

Input: set of initial scenarios C, topology T
Output: a set of test inputs and failing inputs

1: $S \leftarrow C$
2: $F \leftarrow \emptyset$
3: $totalCoverage \leftarrow \emptyset$
4: **while** time period expires **do**
5:     **for all** $scenario \in S$ **do**
6:         $mutant \leftarrow \text{MUTATE}(scenario, T)$
7:         $coverage \leftarrow \emptyset$
8:         $I \leftarrow \emptyset$
9:         **for all** $operation \in mutant$ **do**
10:             $history, result \leftarrow \text{RUN}(operation, I, T)$
11:             $coverage \leftarrow coverage \cup \{history\}$
12:             **if** result = FALSE **then**
13:                 $F \leftarrow F \cup \{mutant\}$
14:                 **break**
15:         **if** $coverage \notin totalCoverage$ **then**
16:             $S \leftarrow S \cup \{mutant\}$
17:             $totalCoverage \leftarrow totalCoverage \cup coverage$
18:         $\text{ROLLBACK}(mutant, I, T)$
19: **return** S, F

redundant operations will waste time finding irrelevant scenarios. INTENDER considers the *intent-operation dependency* (*IOD*) between intent operations to generate more relevant test scenarios. During the scenario mutation, it calculates the expected states of each intent and generates the next operation among appropriate operations. For practitioners that want to test exceptional cases, INTENDER provides a knob to adjust the frequency of exceptional operations.

In addition, topology operations can be dependent on intents. While deleting existing links or hosts highly affects `INSTALLED` intents, adding links or hosts randomly will not likely change `FAILED` intents, since an intent will shift to the `INSTALLED` state when all ends of the intent exist in the topology. Therefore, INTENDER can prioritize adding one end of any `FAILED` intent as a new link or host to increase efficiency.

Intent-State Transition Guidance (ISTG). Code-coverage guidance (CCG), which is found in most traditional fuzzers [15,16,24,43,47,49,62], is significantly limited when guiding a fuzzer to find unknown IBN semantic bugs **(C3)**. Although CCG helps find some installation bugs, other bugs (including the motivating example) may not be distinguished by the code coverage metric.

For a new metric, we focus on intent-state transitions. When any operation affects the intent, the controller executes the operation; the intent's state will be changed. If the intent state is not changed, it is likely that the controller does nothing and the underlying network remains as it is. Therefore, by guiding INTENDER to find more diverse transitions in intent states, it will invoke functions that have not been executed before.

With this new metric, we devise a new guidance policy called *intent-state transition guidance* (*ISTG*) as shown in Algorithm 1. ISTG receives a set of initial scenarios C from

Table 4: INTENDER mutation rules.

Rule	Description
R1: *Max appending*	Max number of operations appended to the end
R2: *Max truncating*	Max number of operations truncated at the end
R3: *Topology-operation overhead*	Overhead on `topology-change` operations
R4: *Exceptional operation overhead*	Overhead on exceptional operations
R5: *Velocity*	Velocity of adding operations in the scenario

a user that is stored in a set S of seed scenarios (line 1). In each step, ISTG chooses a scenario in S for the mutation (line 5). ISTG mutates the chosen scenario with the topology information T (line 6), runs each operation within the mutant scenario, and receives its result and the history of intent-state transitions during the operation (line 10). When a new intent is installed, ISTG records its state. When an intent is modified, withdrawn, or purged, it records the index and the final state of the intent. If the topology operation is executed, ISTG records the indices and the final states of all affected intents. If an operation fails during the execution, the algorithm stops running the mutant scenario and stores it into F (line 13). After executing all operations of the scenario, ISTG checks whether this history coverage is new or not (line 15). If it introduces new coverage, ISTG stores the mutant into S (line 16) and updates $totalCoverage$ (line 17). Finally, ISTG rolls back the topology into the initial state and removes the remaining intents (line 18). The process continues until a given time period expires, and ISTG returns the seed corpus S and the failed inputs F as results (line 19).

ISTG attempts to find additional unique intent-state transitions.[7] Changes in the intent state will bring more diverse situations, and we show in Section 7.2 that the likelihood of finding bugs will increase accordingly.

Mutation policy. To allow users more control over INTENDER in their environments, we provide five mutation rules as shown in Table 4. Users can regulate the number of new operations by specifying **R1** and **R2**, which are the maximum number of operations appended or truncated at the end of a previous scenario. **R3** defines the overhead of `topology-change` operations compared to other intent operations. As the latency of `topology-change` operations may differ among network environments, users can set a small number for the **R3** value. In addition to overhead, the probability of executing topology-change operations will increase when the number of existing intents increases. **R4** specifies the overhead of exceptional operations based on the dependency described in Section 5.2. Finally, **R5** determines the velocity of appending operations.

5.3 INTENDER Architecture

Given the aforementioned intent model and fuzzing techniques, we now present INTENDER's architecture. In order to solve the challenges outlined in Section 3.3, we identify a set of design goals as follows:

[7]If the guidance mutates a scenario by only appending `add-intent` operations, the length of transitions will increase and every scenario will be considered unique. To avoid adding operations infinitely, practitioners can regulate the velocity of appending operations (**R5**).

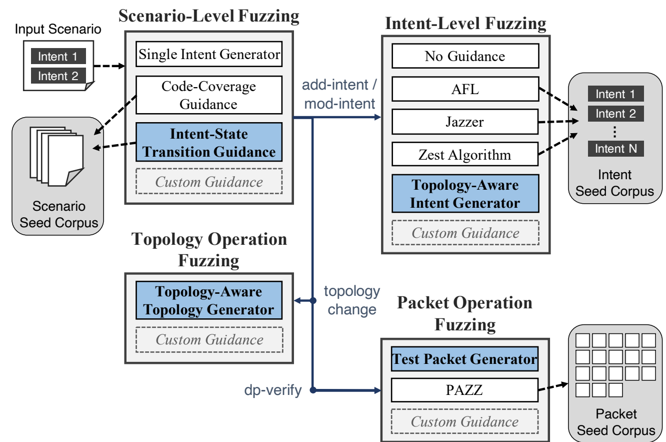


Figure 7: Fuzzing abstract layers in *Intent Scenario Fuzzer*. INTENDER employs blue-colored guidances by default.

Goal (G1): Operation handling. While a random input of an intent request might cause a bug in IBN, other intent operations or changes in topology can trigger bugs in an intent controller. The architecture should support the random ordering of operations, as well as random values in each operation.

Goal (G2): Detection mechanism. To discover both syntactic and semantic intent bugs, the architecture should have a new detection mechanism for semantic bugs. The architecture should recognize intent semantics violations in the control plane by checking reachability, as well as in the data plane by sending test packets.

Goal (G3): Abstract design. The new architecture should be designed with abstraction layers to allow developers to add their own features as plugins. Such interfaces should support multiple IBN implementations and new fuzzing techniques.

Figure 5 shows the various components of INTENDER's architecture. Developers can supplement components without affecting other components (*e.g.,* support of new IBN implementations or new fuzzing techniques).

In the **fuzzing layer**, INTENDER receives input seeds from the user. The *intent scenario fuzzer* selects a seed scenario from the *scenario seed corpus* and mutates it with its own policy, as shown in Figure 7. Scenario-level fuzzing mutates the order of operations in the seed scenario. Intent-level fuzzing uses a *topology-aware intent generation* policy that randomly chooses two topology endpoints to generate a random intent. Users can replace our generator with other fuzzing techniques that mutate an intent in the *intent seed corpus*. Topology-operation fuzzing mutates topology operations by randomly choosing a valid device, host, or link. For the data plane verification, packet-operation fuzzing will generate test packets that represent installed intents or utilize other tools such as PAZZ [51]. After mutating the seed, the *intent scenario processor* uses new input to return the history of intent-state

transitions and the results of the input scenario to the intent scenario fuzzer as feedback guidance.

In the **service layer**, the *intent operator* adds, modifies, or deletes a given intent (**G1**). Since the intent or the request message could be either syntactically valid or not, the *intent operator* directly sends the message as a string through intent APIs. If the request is handled successfully, the *intent operator* stores an intent object with an expected state in the intent store. The *intent checker* checks the correctness of the intent state from the control plane's perspective and verifies the target intent when it receives the `cp-verify-intent` operation from the processor (**G2**). The *network manager* acts as a proxy of the *test agent* to manage all data-plane operations by handling topology operations (**G1**) and the `dp-verify-intent` operation (**G2**). When receiving a `topology-change` operation, the *network manager* requests the operation to the *test agent*, receives the corresponding topology update, and confirms that the update is correct.

In the **storage layer**, INTENDER stores intents and the current topology to verify an intent's correctness. In addition, the fuzzing stage uses a configurational store when mutating scenarios, as the generator cannot actually run operations and change the topology during this execution.

In the **interface layer**, INTENDER's APIs allow developers to extend and support new IBN implementations and fuzzing techniques (**G3**). To use the application API, the *application agent* runs alongside the IBN controller. The topology communication actively updates the topology graph with nodes for devices and hosts and edges for links, whenever it receives any message from the controller. INTENDER supports a code coverage API to measure code coverage; that requires a *code coverage agent* within the controller.

INTENDER provides a **network testbed** (**G1/G2**) to support data plane verification and topology operations. The *test agent* operates and manages an emulated network, though this does not preclude INTENDER's use in a real network environment (**G3**). The agent can start the network emulation with a given network topology at the beginning and mutate the topology during the execution such as adding or deleting hosts, links, and devices without physical constraints. A *host agent* can send or receive test packets to support data plane verification.

The **application agent** requests intent operations by receiving commands from the main architecture. The application agent can also check for controller or topology availability.

5.3.1 Control plane verification

Given the `cp-verify-intent` operation, the *intent checker* conducts the control plane verification of given intents. Algorithm 2 describes the process of verification in the control plane. Before getting the result from the controller, it calculates the expected state of intent based on the underlying topology (line 14). The function EXPECT checks whether any path between a source and a destination exists or not (lines 1–8). For a given waiting time, it repeatedly gets the result from the controller (line 16). If the result of the state and data is the same as the expected state and the original data, the control plane verification of the intent succeeds (line 18). Otherwise, the intent fails in the verification (line 20).

The *intent checker* can detect any inconsistency in the reachability intent. We can further extend the *intent checker* with the intent composition tools that synthesize diverse types of intents [10, 46, 54].

5.3.2 Data plane verification

Given the `dp-verify-intent` operation, the *network manager* delivers this operation to the *test agent* and receives the

Algorithm 2 The Intent Checker Algorithm

```
1:  function EXPECT(I, T)
2:      s ← T.GETNODE(I.src)
3:      d ← T.GETNODE(I.dst)
4:      if T.PATHEXISTS(s,d) == FALSE then
5:          return FAILED
6:      else if I.isBothWay == TRUE &
             T.PATHEXISTS(d,s) == FALSE then
7:          return FAILED
8:      return INSTALLED
9:
10: procedure MAIN
11:     I ← the intent
12:     T ← the current topology (N: nodes, E: edges)
13:     C ← the target controller
14:     expcState ← EXPECT(I,T)
15:     while waiting time expires do
16:         J ← GETINTENT(C, I.key)
17:         if J.data = I.data & J.state = expcState then
18:             return TRUE
19:         WAIT()
20:     return FALSE
```

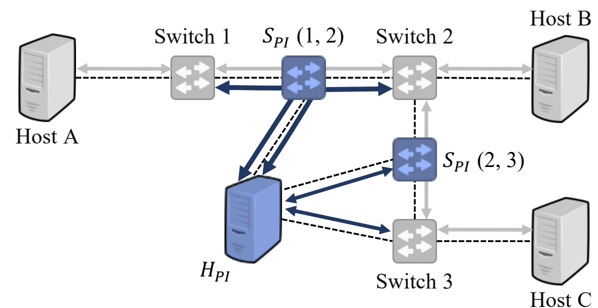


Figure 8: Point interleaving switch (S_{PI}) and host (H_{PI}). An intent controller can only detect grey nodes and edges, and it recognizes as two ends communicate through grey arrows. Blue arrows show mirroring paths from and to H_{PI} via S_{PI}.

result of verification of the target intent from the *test agent*. We provide two ways for the *test agent* to execute the data plane verification: test packets and PAZZ [51].

Intent-based test packet. The *test agent* requests `SNIFF` to the destination and `SEND` to the source. When the *host agent* at the destination receives the correct packets, the *host agent* returns the test success message to the *test agent*, which forwards the success response to the *network manager*. Otherwise, the *test agent* sends the timeout response to the *network manager*. In the case of intents with bidirectional connectivity, the *host agent* in a source can request a `PING`.

However, the source or destination of the intent can be (1) empty, (2) one of the ends of the link, or (3) a port connected to the host. To send a packet from any arbitrary port, we leverage *point-interleaving switches* (S_{PI}) and a *point-interleaving host* (H_{PI}) in the network testbed, as shown in Figure 8. First, if the intent has an empty switch port, the *test agent* adds a temporary link between the port to H_{PI}. Second, if the intent has a port within the existing link, test packets should be injected within the link. For this purpose, there is a point-interleaving switch (S_{PI}) for each link. Since S_{PI} cannot send packets, H_{PI} connects all S_{PI}s with multiple data interfaces. If the source of the intent is one of the ends of the link, H_{PI} sends test packets to the interface that connects to S_{PI} of the link. If the destination of the intent is one of the ends of the link, S_{PI} mirrors test packets forwarded to H_{PI} for `SNIFF`. Finally, if the intent has a port connected to the host, the *host agent* at the host will send or sniff test packets. After the test, the *test agent* removes all temporal links and interfaces.

PAZZ detects any inconsistency between the control plane and the data plane by comparing information in the controller with packet metadata that contains the path and flow rule history. While PAZZ does not need any special node (S_{PI} and H_{PI}) in the topology, PAZZ requires modified switches to add a custom L2 `VERIFY` header as the metadata.

Initially, the *network manager* requests a PAZZ test to the *test agent* that requests a sender to send *production traffic* with `tcpreplay` for performance (*e.g.,* 10^6 *pps*). If the initialization succeeds, the *network manager* learns the sender's management IP address to directly request *fuzz traffic* to the *host agent* in the sender. The *network manager* mutates packets among the header space uncovered by the given intent. The receiver samples packets from both *production traffic* and *fuzz traffic* and sends them to the *consistency tester*. The *consistency tester* requests the path and rule history of the sample packet to the *network manager* since the *network manager* can calculate this history with the topology and flow rules stored in the storage layer. Finally, the *consistency tester* will recognize any data-plane fault by comparing the history with the `VERIFY` header of the sample. When the *network manager* tests all possible headers or the time expires, it stops all processes and clears all resources made during the test.

6 Implementation

We implemented INTENDER in Java and chose ONOS v2.5.1 as our intent controller. Our implementation is available at https://github.com/purseclab/intender. We wrote the application agent in Java to request intents to ONOS. We built the network testbed with network emulation using Mininet v2.2.2, running Open vSwitch v2.14.0. Within the testbed, we implemented the test agent and host agents in Python with the Scapy library to send and sniff packets. To learn the topology view of the ONOS controller, we execute a RabbitMQ server that transmits the topology change messages from the controller to INTENDER.

Our framework easily supports other IBN implementations by replacing interfaces and relevant components, which we discuss in Section 8.

Furthermore, we implemented 4 state-of-the-art fuzzers within INTENDER framework to demonstrate the generalization of our framework and compare against them with our new fuzzing mechanisms. Since existing fuzzers require either target binaries or unit tests, we ported these fuzzers into INTENDER to execute fuzzing tests on a running IBN controller. We describe these implementations in detail.

AFL. We leverage JQF [42] framework to support AFL [62] in Java. As JQF utilizes a proxy program for Java programs to communicate with AFL, we implemented a proxy for INTENDER. The proxy transfers a new random input from AFL to INTENDER via inter-process communication (IPC). After running a test with the input, INTENDER records coverage and sends it back as feedback to the proxy. The proxy finally stores the coverage into a shared memory accessible by AFL.

Zest. We also ported Zest [43] algorithm from JQF. Zest leverages a *parametric generator* from a stream of pseudorandom numbers and *semantic guidance* by storing valid inputs into the seed corpus. Since Zest requires a custom generator to follow the grammar of the target program (*e.g.,* XmlDocumentGenerator for WebXml), we implemented our custom generator to map random numbers to inputs that are syntactically valid in the intent grammar.

Jazzer [30] is another coverage-guided fuzzer for the JVM platform, integrated into Google's OSS-Fuzz [50]. Jazzer receives either test programs or whole binaries as a target, then fuzzes them using the libFuzzer engine [49]. Instead of porting all codebases of Jazzer to INTENDER, we used another proxy program to communicate with Jazzer. INTENDER communicates random inputs and coverage data with the proxy via IPC. The proxy leverages `TRACE_PC_INDIR` to set a bit in the libFuzzer's value-profile bitmap. The overhead of IPC used in both AFL and Jazzer is negligible compared to the overhead of other heavy operations such as `dp-verify-intent` or `topology-change`.

PAZZ [51] is designed to verify the inconsistency between the control plane and the data plane in SDN. PAZZ receives a set of policies from an SDN controller, then combines pro-

duction traffic with active probes using abnormal packets for periodical testing. PAZZ requires modified switches to create a custom L2 shim header that stores a hash chain of rules and ports as its path history. By comparing this header of sample packets with an expected hash chain from the controller, PAZZ can detect data-plane faults in three components: flow rules, topology, and paths. Unlike AFL, Jazzer, and Zest, PAZZ mutates a random packet to test the data plane.

7 Evaluation

We evaluated INTENDER with ONOS v2.5.1. We ran all components within a single virtual machine, *e2-standard-4* in Google Cloud Platform: 4 vCPUs, 16GB memory, and 60GB SSD. We ran tests for 24 hours for bug discovery and performance comparison. To mitigate the non-deterministic behavior of fuzzing, we repeated experiments 20 times. We show the 95% confidence interval around the average as the error bands.

We compare the fuzzing efficiency of INTENDER to AFL [62], Jazzer [30], Zest [43], and PAZZ [51], in terms of code coverage and bug detection. We used the Eclemma-JaCoCo library (v0.8.6) [28] to measure IBN-related code coverage within ONOS. We extracted relevant methods from ONOS using Soot (v4.2.1) [57]. We started with 4 intent-related entry points to track all code paths to find methods that these entry points pass through. This is because ONOS contains a huge number of modules such as applications and drivers. By running the pre-runtime analyzer, we can eliminate those classes irrelevant to the intent processing.

Mutation rules. For the rest of our evaluation, we set the following default values for all mutation rules in Table 4. We set **R1** and **R2** as 3 and 1 to mutate scenarios to have more random actions. In our test environment, we found that the average time spent on `topology-change` is 33.3× longer than the time spent on `add-intent`. To generate the appropriate number of `topology-change` operations, we derive **R3** as 16 to spend approximately 2× more time on `topology-change` than `add-intent`. We set **R4** as 4 to generate the exceptional case every four normal operations. **R5** (v) decides whether to add operations or not (f) by calculating the following equation with the number of scenario seeds (S) and the number of actions within the previous scenarios (P):

$$f(S,P) = \begin{cases} 1, & \text{if } (|S| \gg \max\{|P|-v, 0\}) > 0. \\ 0, & \text{otherwise.} \end{cases}$$

We set **R5** as 10 to limit the velocity of adding operations after 10 operations. Users can configure all knobs as needed.

7.1 Discovered Vulnerabilities and Bugs

We notified the ONOS Security Response Team of all of the vulnerabilities found through INTENDER by following a responsible disclosure process. The Security Response Team acknowledged our discoveries.

7.1.1 Security case studies

We introduce two previously unknown IBN vulnerabilities found by INTENDER to demonstrate their adverse effects in the correct network operation.

Flow manipulation by crafted intents. The intent can modify or delete flow rules of existing intents that share the path (BUG 8; CVE-2021-38364). In the case of a `PointToPoint` intent that connects two different points, flow rules installed by the intent have only an `IN_PORT` match and an `OUTPUT` action. If an attacker installs a new intent with the same as the existing intent except for the destination, flow rules will be overwritten to be redirected to the destination. It happens due to the absence of comparison of flow rules installed by intents. While INTENDER can find this bug by generating multiple `add-intent` operations, other fuzzing tools which request a single intent for each step cannot detect errors in multiple intents.

Exceptional operation. Although the `purge-intent` operation does not have any impact on installed intents, a `PointToPoint` intent ignores any relevant topology-change events after a purge request (BUG 12; CVE-2022-24035). This bug happens due to the early withdrawal of the management of devices. When a user requests `purge-intent` to an intent with the `INSTALLED` state, `IntentManager` in ONOS adds this intent into the pending queue and withdraws the management of devices specified in the intent immediately. `IntentBatchDelegate` receives the pending intent and rejects purging this intent. Since it already removed the device information for this intent, the intent will not be recompiled, even though one of the devices or links on the path is disconnected. If an attacker requests `purge-intent` to any installed intent, the intent might not be affected at that time, but its connectivity will be disrupted after changes in topology. INTENDER can find this bug by generating a scenario with an exceptional case and topology mutation. However, other fuzzing tools cannot find it, since they do not support these operations.

7.1.2 New vulnerabilities and bugs

INTENDER found 12 previously unknown bugs as shown in Table 5, 11 of which are security-critical vulnerabilities assigned with 11 CVEs. The unique characteristics of IBN bring a new attack surface, such as intent-store exhaustion (BUG 1), mishandling intent-state transitions (BUG 2–4, BUG 11, BUG 12), violating network invariant (BUG 5), absence of intent-flow mapping (BUG 6, BUG 9), and flow-rule conflicts (BUG 7, BUG 8, BUG 10).

Among newly found bugs, AFL, Jazzer, and Zest found 3 syntactic bugs related to the incorrect grammar or invalid

Table 5: List of unknown intent bugs in ONOS discovered by INTENDER.

#	CVE ID	Type	Operation	Description
1	CVE-2021-38363	**SYN2**	`add-intent`	`PointToPoint` intent with invalid point field causes NullPointerException
2	CVE-2022-29604	**SYN4**	`add-intent`	`PointToPoint` intent which has an upper-case letter in a device ID shows `CORRUPT`
3	CVE-2022-29606	**SYN4**	`add-intent`	`PointToPoint` intent which has a large switch port number shows `CORRUPT`
4	CVE-2022-29609	**SEM1**	`add-intent`	`HostToHost` intent with the same source and destination shows `INSTALLING`
5	CVE-2022-29608	**SEM2**	`add-intent`	`PointToPoint` intent installs an invalid flow rule causing network loop
6	CVE-2022-29605	**SEM2**	`add-intent`	Intent tries to install IPv6 flow rules into OF10 switches
7	CVE-2022-29944	**SEM3**	`add-intent`	Intent cannot bypass intents with higher priority
8	CVE-2021-38364	**SEM3**	`add-intent`	Intent can delete or modify flow rules of previous intents which share the path
9	-	**SEM4**	`add-intent`	`PointToPoint` intent with switch port 0 installs useless flow rules
10	CVE-2022-24109	**SEM3**	`withdraw-intent`	Deletion of one of the duplicate intents removes all flow rules
11	CVE-2022-29607	**SEM1**	`mod-intent`	`HostToHost` intent modified to have same source and destination shows `INSTALLED` without any flow rules
12	CVE-2022-24035	**SEM1**	`purge-intent` & `topology-change`	After requesting purge on installed `PointToPoint` intent, the state of intent does not change to `FAILED` with link failure

value of intent requests. These bugs can be easily discovered by checking internal logs. The remaining 9 bugs were found by our new fuzzing techniques in INTENDER. While 6 of those bugs occurred when adding an intent, 3 were introduced during other operations. BUG 12 occurs with multiple operations on the intent. Attackers can exploit these bugs to cause integrity violations of intents. While AFL, Jazzer, and Zest may find a few bugs with syntactically-invalid inputs, our techniques employed in INTENDER focus more on finding semantic bugs with multiple operations. These methods can be complementary to each other.

7.2 Performance Comparison

Overall efficiency. We evaluate INTENDER in terms of the efficiency of bug finding and compare it with AFL, Jazzer, Zest, and PAZZ. Figure 9 shows the experimental results of different fuzzing tools. "V/AFL" represents "Vanilla AFL" without any verification in the control plane (CP) and the data plane (DP). It treats a random string as an intent and checks for a program failure with a response code. "AFL", "Jazzer", and "Zest" operate their own mutation and guidance policy within the INTENDER architecture. "PAZZ" continuously verifies the consistency of pre-installed intents. For PAZZ experiments, we replicate the 4-ary fat-tree experimental setting in its paper by using three installed intents with destination-based routing. "INTENDER" includes all features proposed in the paper.

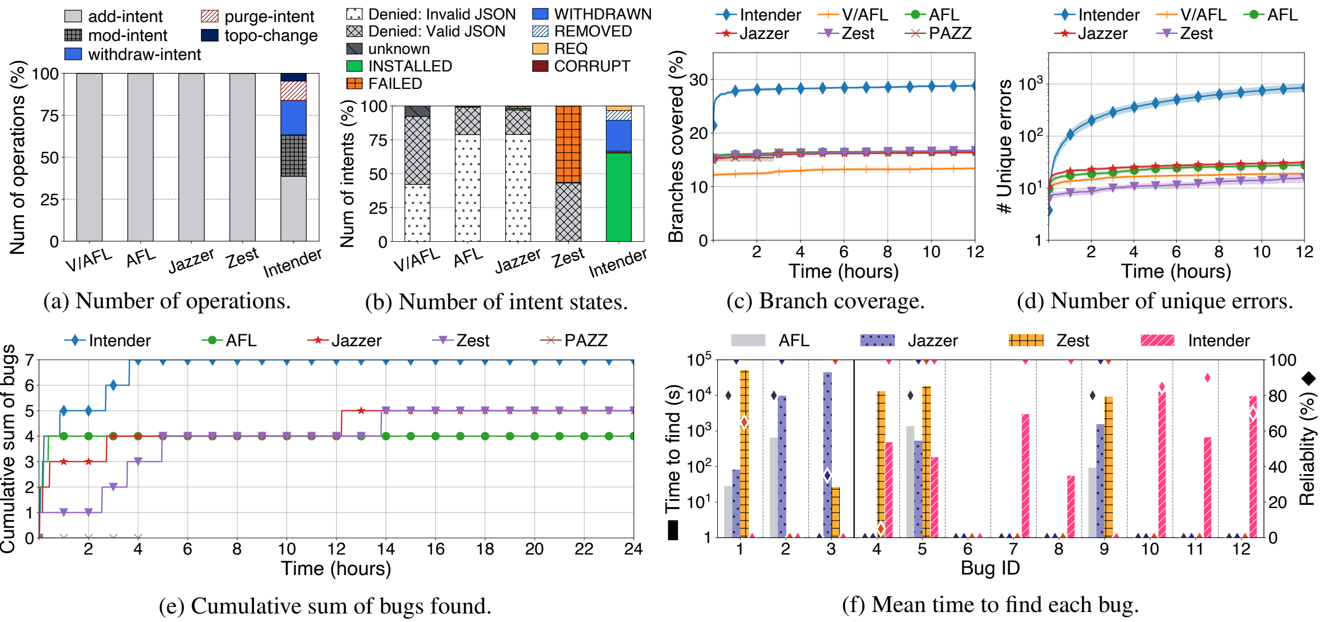
Figure 9a and Figure 9b show the proportional number of operations and intents, respectively. Since PAZZ checks the inconsistency of IBN with the three pre-installed intents, it does not generate any additional intent during the test and stops fuzzing in 4 hours after covering all possible IP addresses of destinations. We omit PAZZ in the figures accordingly. Vanilla AFL, AFL, Jazzer, and Zest only request `add-intent` with random inputs. Due to their randomness, 91.28%, 99.11%, and 97.07% of intents generated by vanilla AFL, AFL, and Jazzer are denied by the controller. Zest reduces the denial rate of intents to 43.14% with grammatically-valid inputs. Even though the random intents are accepted, `INSTALLED` intents generated by AFL, Jazzer, and Zest account for only 0.19% 1.16%, and 0.56% (vanilla AFL cannot distinguish the intent state). On the contrary, INTENDER always generates valid intents in terms of grammar and topology. Moreover, INTENDER can generate other intent states such as `WITHDRAWN` and `REMOVED`, since INTENDER executes all operations, unlike AFL, Jazzer, and Zest. As a result, INTENDER can test the semantics of IBN efficiently with different operations and states.

Moving forward, we compare INTENDER to other fuzzers in terms of code coverage and bug finding. For the code coverage as shown in Figure 9c, we measure the branch coverage of relevant classes implementing methods that intent operations rely on. Vanilla AFL without any verification can only test the intent API parser resulting in 13.6% coverage. AFL, Jazzer, and Zest show similar coverage, such as 16.66%, 16.56%, and 17.44%, respectively. Despite only with three installed intents, PAZZ shows 16.41% coverage, similar to others. Instead, INTENDER outperforms all existing fuzzers with 29.97% branch coverage. Since INTENDER tests the semantics of IBN with multiple operations that consist of valid inputs, it shows 2.2×, 1.8×, 1.8×, 1.7×, and 1.8× better coverage than vanilla AFL, AFL, Jazzer, Zest, and PAZZ, respectively.

Regarding the bug-finding performance, Figure 9d describes the number of unique errors[8] found during fuzzing. Since bugs discovered by INTENDER mostly do not incur any crash in the controller, we employ the number of unique errors to calibrate bug-finding capability, instead of checking stack backtraces. As a result, INTENDER generates more unique errors compared to other fuzzers: 82.6×, 56.4×, 52.5×, and 64.3× more than vanilla AFL, AFL, Jazzer, and Zest, respectively. PAZZ cannot find any error or bug, since it focuses on finding an inconsistency between intents and network by mutating packets, not intents.

We also measured the number of bugs found by different

[8] We define an IBN error as an input that has one of the adverse impacts described in Table 1.



(a) Number of operations.
(b) Number of intent states.
(c) Branch coverage.
(d) Number of unique errors.
(e) Cumulative sum of bugs found.
(f) Mean time to find each bug.

Figure 9: Performance comparison tests. V/AFL stands for vanilla AFL. In Figure 9f, the rectangular bars are the *time-to-find* and the diamond dots are the reliability of each bug in Table 5. The 24-hour tendency for branch coverage (9c) and unique errors (9d) remains consistent across the codebase criteria, such as relevant packages, classes, and methods.

fuzzers over time, as shown in Figure 9e. Vanilla AFL cannot detect any bug without any verification. Although the IBN controller responded with an internal server error for random inputs generated by vanilla AFL, there was no crash in the controller. With the help of verification modules, AFL, Jazzer, and Zest found up to 5 bugs in total. PAZZ, which tests only one set of intents, did not discover any bug. Instead, INTENDER found 7 bugs within 4 hours. In terms of bug efficiency, AFL, Jazzer, Zest, and INTENDER can discover 4, 4, 3, and 7 bugs within the first four hours, respectively.

Further, we examine the *time-to-find* for each bug in Table 5 in Figure 9f, except for BUG 6 which occurs when the version of the switch mismatches with the intent specification. Since we do not mutate versions of switches in this experiment, BUG 6 is not found by any fuzzing tool. The time-to-find is the average time until a bug is first discovered by a fuzzer. The *reliability* shows the success rate of bug finding among 20 trials.

INTENDER focuses on finding semantic bugs, while AFL, Jazzer, and Zest typically find syntactic bugs, and PAZZ cannot find any bugs. A pure random string generated by AFL and Jazzer can be easily denied by the REST parser and leads to internal server error responses. Although some of the random strings might follow the JSON format, they could contain a new JSON key that is not allowed in the intent system, such as "de" instead of "device", which causes an exception within the intent processor. This kind of input is not found after the request (BUG 1). In contrast, a syntactically-valid input from Zest rarely finds BUG 1 with longer *time-to-find* and lower reliability, since inputs always follow the intent grammar. AFL, Jazzer, and Zest can generate corrupt intents, which occur due to a wrong upper-case letter in a device ID (BUG 2) or a large port number (BUG 3).

While AFL, Jazzer, and Zest also find a few semantic bugs, these are shallow bugs compared to others. BUG 5 found by these fuzzers and INTENDER is related to a network loop problem of a single intent that can be easily reproduced by mutating a port of a normal intent. Also, AFL, Jazzer, and Zest can detect BUG 9 that creates garbage flow rules by setting a switch port number as 0, whereas INTENDER cannot generate a zero-numbered port for any intent, since it depends on the underlying topology consisting of natural number ports.

Meanwhile, INTENDER finds significantly more semantic bugs than AFL, Jazzer, and Zest. First, INTENDER finds BUG 4 that is not found in AFL and Jazzer, and is unlikely to be found in Zest (5% reliability). Since INTENDER refers to the topology, it can easily detect BUG 4, which requires the intent with the same source and destination. Similarly, INTENDER can detect BUG 11, which occurs when modifying an existing intent into an invalid intent with the same source and destination. Moreover, unlike AFL, Jazzer, and Zest, inputs from INTENDER are likely to be accepted by the controller.

As a result, INTENDER can find 7 semantic bugs, 5 of which are not discovered by AFL, Jazzer, and Zest. PAZZ attempts to discover an inconsistency in the data plane without mutating intents, thus cannot find any semantic bug. Since INTENDER generates a list of diverse operations with several states, it can find more semantic bugs.

Intent validity. To show the efficiency of *topology-aware intent generation* (TAIG), we measure the validity of intents generated by AFL, Jazzer, Zest, and TAIG for 3 hours. We exclude PAZZ since it does not mutate intents. For this ex-

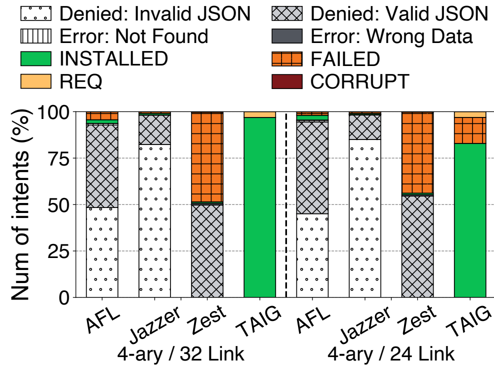


Figure 10: Results of intent validity.

periment, TAIG generates only one `add-intent` operation in each scenario, as in AFL, Jazzer, and Zest. Figure 10 shows the experimental result on two different 4-ary fat-tree topologies: without any link failure (4-ary / 32 Link) and with a few link failures (4-ary / 24 Link).

The ratio of accepted intents of Zest (50.22%) is much higher than one of AFL (7.19%) and Jazzer (1.95%) since the generator in Zest always creates inputs valid in the intent grammar. `INSTALLED` intents generated by AFL, Jazzer, and Zest account for only 1.96%, 0.82%, and 0.77% for the 4-ary fat-tree topology and 2.55%, 0.63%, and 0.82% for the topology with link failures, respectively. While AFL could generate more `INSTALLED` intents than Jazzer and Zest, the ratio of `INSTALLED` intents generated by AFL decreases to 0.19% after running 24 hours as shown in Figure 9b. Moreover, AFL, Jazzer, and Zest generate a small number of `INSTALLED` intents within a margin of error, regardless of link failures in the topology.

Contrary to AFL, Jazzer, and Zest, `INSTALLED` intents generated by TAIG account for 96.9% and 82.9% for the topology without and with link failures, respectively. Since TAIG refers to the topology, it can generate 9.4×, 10×, and 78.7× more `INSTALLED` intents than AFL, Jazzer, and Zest, respectively. TAIG also shows that the number of `INSTALLED` and `FAILED` intents are dependent on the link failures in the topology. Since TAIG can generate more topology-relevant intents, it helps focus on finding semantic bugs instead of shallow ones.

Redundant operations. We measure the number of redundant operations and elapsed time executed by INTENDER with and without the intent-operation dependency (IOD) for 3 hours. We classify `withdraw-intent` on `WITHDRAWN` intents and `purge-intent` on `INSTALLED` intents as redundant operations. As a result, INTENDER with IOD reduces 73.02% of redundant operations, contrary to INTENDER without IOD. Since INTENDER without IOD generates more redundant operations, it spends 3.46%, 19.7%, 8.69%, and 17.2% more time on withdrawal, purge, verification, and cleanup for redundant intents than one with IOD, respectively. INTENDER with IOD reduces these operations and spends 10.74% more time on valid operations, especially 15.34% more time on `topology-change` operations.

Coverage metric. We evaluate INTENDER with the code-coverage guidance (CCG) and the intent-state transition guidance (ISTG) for 24 hours. Figure 11 shows the result of the performance of CCG and ISTG during the first 12 hours. After 12 hours, 85% of machines stop running INTENDER due to a topology inconsistency between the IBN controller and the network, caused by errors in `topology-change` operations. Even though the number of tested machines decreases after 12 hours, causing the error band to become wider, the overall trend remains unchanged.

Figure 11a shows the branch coverage. We measure the branch coverage of relevant classes implementing methods that intent operations rely on. In terms of branch coverage, the two guidance policies have similar coverage around 30%. Since we devise ISTG to find more transitions in intent states, not to cover more code paths, ISTG does not improve the branch coverage. Both guidance policies comparatively show a small coverage, since we test IBN with only reachability intents without diverse match fields and actions, which are one of the vast parts in the code of IBN. In addition, measuring the code coverage during the execution of operations excludes the code used to initialize the controller and synchronize the cluster.

There are also two reasons why these guidances have the same coverage. As described in Section 3.2, two different scenarios with the same number of flow rules and intent operations show the same coverage in relevant methods, even though there is a difference in the order of operations. Also, we find that multiple intents that share the path show the same code coverage regardless of the number of installed intents, which affects changes in IBN for subsequent topology operations. Recent work in the literature [17] indicates that a fuzzer with better coverage may not be best at finding bugs, so the same coverage in both CCG and ISTG does not imply the same probability of finding bugs.

Figure 11b shows the number of unique intent-state transitions, which shows that ISTG can guide INTENDER to generate a larger number of unique intent-state transitions than CCG. As noted in Section 5.2, ISTG might guide INTENDER to generate a scenario with numerous `add-intent` operations. Since we restrict the number of appended operations to prevent this shortcoming, the maximum number of intent-state transitions in each scenario is not proportional to the number of unique transitions, as shown in Figure 11c. Also, even with the regulation, ISTG shows 1.8× higher in the maximum number of intent-state transitions than CCG.

Figure 11d shows the ratio of unique errors that are found. ISTG can find a larger number of unique errors with a higher ratio than CCG, as ISTG guides INTENDER to find more intent-state transitions.

As a result, while CCG and ISTG have similar branch coverage, ISTG has more coverage in intent-state transitions than CCG. Frequent transitions among states invoke more changes in flow rules and network topology. ISTG is better

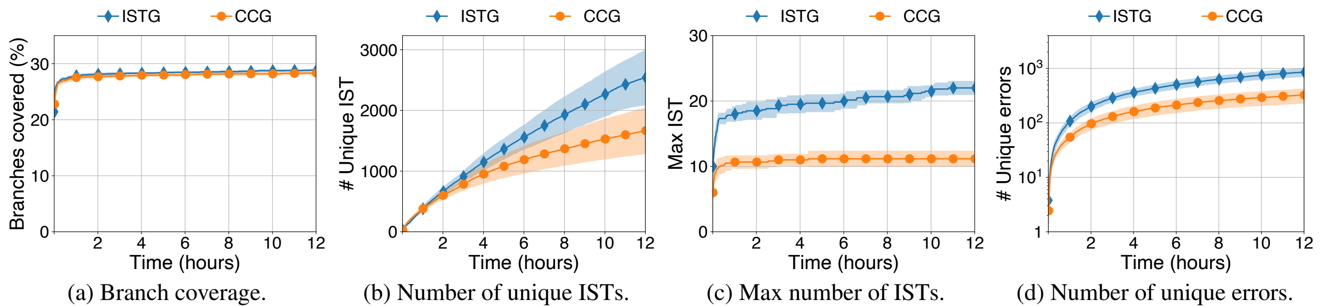


Figure 11: Performance comparison tests for CCG and ISTG. IST stands for intent-state transitions. The overall trend remains unchanged for 24 hours.

guidance for IBN because it guides an IBN fuzzer to generate more diverse situations within the IBN.

8 Discussion

Support for other controllers. While we implement and evaluate INTENDER on ONOS, INTENDER is not tied to one IBN controller or implementation. IBN developers and users can use our framework to test their solutions by modifying only a few components. Users can write interfaces such as intent API and topology API to execute fuzzing tests on IBN controllers with INTENDER. Users can write their own mutation or guidance policies. In addition, ISTG can be applied to other controllers that use intent-state transitions, such as ODL [40]. Since ISTG does not require access to the controller code, INTENDER can test closed-source controllers so long as the topology and intent states can be queried from the controller's API.

Physical testbed. We can extend our emulation-based testbed to a physical testbed with actual devices by deploying host agents to each machine. INTENDER can also execute remote commands to physical switches to mutate the topology (*e.g.*, link deletion). While mininet has a limit in resources on a single server, we do not encounter any issues in our test environment. In addition, we leave bandwidth-constraint testing that could depend on computing resources as future work.

Other types of intents. In this paper, we propose our new fuzzing tool to test reachability intents. INTENDER verifies intent connectivity by calculating the path and sending a test packet. In addition to reachability, there are other types of intents, such as waypoint, QoS, stateful, and time-based intents. We leave supporting these kinds of intents as future work.

9 Related Work

Programmable networking fuzzing. Although our fuzzer is the first fuzzer designed specifically for IBN, a closely related area is fuzzing for software-defined networking (SDN). STS [48] introduces a troubleshooting system for SDN with a new technique to automatically identify a minimal sequence of inputs causing a bug. DELTA [37] employs a black-box fuzzing technique to detect unknown vulnerabilities in the controller logic. BEADS [32] automatically generates test cases with respect to the OpenFlow protocol's semantics. PAZZ [51] automatically detects data plane faults to recognize inconsistencies in SDN by fuzzing with the coverage of packet header space. In addition, AIM-SDN [21] and Aud-iSDN [36] attempt to detect inconsistency in configurational and operational stores inside SDN controllers.

Programmable networking security analysis. INTENDER is the first work that systematizes IBN security issues. Most work in studying the security of programmable networks has been in the related SDN context. D2C2 [59] introduces the new data-plane attacks in SDN using data dependency chaining and defense mechanism for these vulnerabilities. EventScope [55] investigates semantic vulnerabilities in SDN through clustering applications. ConGuard [60] discovers time-of-check-to-time-of-use (TOCTTOU) race conditions in SDN controllers. ATTAIN [56] proposes the attack injection framework for SDN with attack analysis.

Network verification. As an orthogonal approach to fuzzing, several works use static verification to enforce network policy. These works verify the composition of different kinds of intents, such as reachability and waypoint [46], bandwidth, stateful, and temporal policies [10], as well as multi-tenancy [54]. Although these tools can verify multiple intents theoretically, the IBN implementations can have software bugs, such as the ones discussed in this paper, that cannot be detected through verification. To test network policies on the data plane, other tools can be used instead of our network test framework [22, 27, 63]. While these approaches can uncover violations in new policy sets, they cannot find errors during topology and intent operation changes. Network invariant verification [33–35, 38] is complementary to INTENDER and can be used to verify network invariants of given intents.

Fuzzing. Fuzzing is used for automated software testing. To move further from generating random inputs, many fuzzers have attempted to cover more code coverage [12, 15, 16, 24, 30, 43, 47, 49, 62], or more paths from the static analysis and concolic execution [18, 26, 53, 61]. Among them, Jazzer [30] and JQF [42] support fuzzing tests on Java programs. We show that existing fuzzers have difficulty discovering semantic bugs in IBN. In addition, fuzzing has been also applied

in the network area to discover unknown bugs [45, 52, 64]. While these works try to cover more protocol states such as the TCP stack, INTENDER traces the transitions of states of each intent, not the state of the intent controller.

10 Conclusion

We presented INTENDER, the first IBN fuzzing framework. We analyzed existing bugs in a popular IBN implementation to derive a set of challenges faced in discovering semantic vulnerabilities. We demonstrated how our intent-state transition guidance approach can be used to efficiently discover semantic bugs and vulnerabilities. We designed a fuzzer architecture and demonstrated that INTENDER can find more semantic bugs as compared to prior methods. INTENDER found 12 unknown bugs and 11 security vulnerabilities in ONOS, which demonstrates its practical efficacy.

Acknowledgments

We thank our anonymous reviewers and shepherd for their helpful and perceptive comments that enhanced this paper.

References

[1] Cisco Intent-Based Networking (IBN). https://www.cisco.com/c/en/us/solutions/intent-based-networking.html.

[2] Juniper Apstra. https://www.juniper.net/us/en/products/network-automation/apstra/apstra-system.html.

[3] ODL Network Intent Composition. https://wiki.opendaylight.org/display/ODL/Network+Intent+Composition, Jan 2015.

[4] ONOS-1409. https://jira.onosproject.org/browse/ONOS-1409, Apr 2015.

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

[6] ONOS-381. https://jira.onosproject.org/browse/ONOS-381, May 2018.

[7] ONAP Intent-Based Network. https://wiki.onap.org/display/DW/Intent-Based+Network, Jan 2021.

[8] ONOS Jira Bug Tracker. https://jira.onosproject.org/, Jan 2022.

[9] 3GPP. Management and orchestration; Intent driven management services for mobile networks. Technical Specification (TS) 28.312, Jan 2023. Version 17.2.0.

[10] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. Supporting diverse dynamic intent-based policies using janus. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[11] Azzam Alsudais and Eric Keller. Hey network, can you understand me? In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017.

[12] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[13] 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 *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. ACM, 2014.

[14] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2Text: Query-guided summarization of network forwarding behaviors. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[16] 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.

[17] Marcel Böhme, Laszlo Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.

[18] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[19] Alexander Clemm, Laurent Ciavaglia, Lisandro Granville, and Jeff Tantsura. Intent-based networking - concepts and definitions. RFC 9315, Oct 2022.

[20] Douglas Comer and Adib Rastegatnia. OSDF: An intent-based software defined network programming framework. In *IEEE Conference on Local Computer Networks (LCN)*, 2018.

[21] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. AIM-SDN: Attacking information mismanagement in SDN-datastores. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[22] Seyed K Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing context-dependent policies in stateful networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[23] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's software-defined networking control plane. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[25] Linux Foundation. Open Network Automation Platform. https://www.onap.org/, Feb 2019.

[26] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[27] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, 2015.

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

[29] 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/, Sep 2019.

[30] Code Intelligence. Jazzer. https://github.com/CodeIntelligenceTesting/jazzer.

[31] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, Walter Willinger, and Sanjay G Rao. Hey, Lumi! using natural language for intent-based network management. In *USENIX Annual Technical Conference (ATC)*, 2021.

[32] 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*. Springer, 2017.

[33] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[34] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[35] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[36] 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.

[37] 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)*, 2017.

[38] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[39] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven SDN controller architecture. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2014.

[40] Anu Mercian, Felipe Yrineu, Joon-Myung Kang, Raphael Amorim, Saket M Mahajani, Mario Sanchez, and Sujata Banerjee. Network Intent Composition (NIC) Be Feature Update and Demo: Intent Compilation, Lifecycle Management and Automated Mapping. http://sched.co/7RBY, 2016. OpenDaylight Summit.

[41] ONF. Intent NBI – Definition and Principles. https://opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf, Oct 2016.

[42] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: coverage-guided property-based testing in Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[43] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with Zest. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[44] Lei Pang, Chungang Yang, Danyang Chen, Yanbo Song, and Mohsen Guizani. A survey on intent-driven networks. *IEEE Access*, 8:22862–22873, 2020.

[45] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: a greybox fuzzer for network protocols. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[46] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[47] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[48] 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 *ACM SIGCOMM*, 2014.

[49] Kostya Serebryany. libFuzzer–a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[50] Kostya Serebryany. OSS-Fuzz-Google's continuous fuzzing service for open source software. 2017.

[51] 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.

[52] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[53] 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 *Network and Distributed System Security Symposium (NDSS)*, 2016.

[54] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[55] 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)*, 2020.

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

[57] 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*. 2010.

[58] Yiming Wei, Mugen Peng, and Yaqiong Liu. Intent-based networks for 6g: Insights and challenges. *Digital Communications and Networks*, 6(3):270–280, 2020.

[59] 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 (S&P)*, 2020.

[60] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the brain: Races in the SDN control plane. In *USENIX Security Symposium*, 2017.

[61] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, 2018.

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

[63] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[64] 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 *USENIX Annual Technical Conference (ATC)*, 2021.