FSF: Code Coverage-Driven Fuzzing for Software-Defined Networking

Hyuntae Kim, Seongil Wi, Hyunjoo Lee, and Sooel Son(✉)

KAIST, Daejeon, Korea
{kimht_,seongil.wi,sn220865,sl.son}@kaist.ac.kr

Abstract. A Software-Defined Networking (SDN) controller plays a key role for assuring the security and robustness of its underlying network system. Previous studies focus on eliciting bugs in such SDN controller via penetration testing or fuzzing without considering code coverage feedback from a target controller under testing. We propose FSF, a code coverage-driven SDN fuzzer. We designed and implemented a fuzzing algorithm to take into account coverage differences incurred by mutated OpenFlow (OF) messages. FSF demonstrated its superiority in increasing the code coverage of a target controller and generated unique 146 tests that trigger bugs in the latest version of Floodlight, a well-known open-source SDN controller.

1 Introduction

Recent years have seen a surging interest in *software-defined networking (SDN)*. SDN is an innovative methodology to build a networking system wherein network controlling attributes are abstracted by software referred to as an SDN controller. SDN has been applied in diverse fields, such as cellular networks [4,31], IoT [11,23], and broadband access [6,29] infrastructures, offering its own benefits for large enterprises and telecommunication networks.

Meanwhile, the growing popularity of adopting SDN calls into question the security of SDN systems. Yoon *et al.* pointed out that emerging SDN stacks have introduced new attack vectors due to their design decisions on facilitating dynamic network flows and topology managements [32]. Previous studies also introduced SDN security challenges [18,26] and manifested concrete attack scenarios [5].

Security researchers have conducted fuzzing and penetration testing to automatically gauge the security of off-the-shelf and open-source SDN systems [9,15,20,30]. Notably, DELTA [20] and BEADS [15] conducted fuzz testing by randomly mutating seed traffic, which are generated by executing the `pingall` and `iperf` commands from hosts. However, these approaches did not leverage any feedback information from a controller under testing, thereby solely depending on the input and output behaviors of the controller. They generated tens of thousands of testing strategies, i.e., test cases, which were chosen at the discretion of the testing analyst without any runtime feedback from a controller.

© Springer Nature Switzerland AG 2020
I. You (Ed.): WISA 2019, LNCS 11897, pp. 41–54, 2020.
https://doi.org/10.1007/978-3-030-39303-8_4

Consequently, the generated test cases do not trigger diverse inherent behaviors of the controller.

In this paper, we design and implement FSF, the first *code coverage-driven SDN fuzzer*. FSF is designed to find bugs in a target SDN system by feeding unexpected test inputs to the south bound communication channels between an SDN controller and SDN switches. The crux of our approach is to leverage the code coverage information obtained from an SDN controller to guide the test case generation.

We evaluated FSF using Floodlight, a popular open-source SDN controller, to vet its capabilities of increasing testing code coverage and discovering SDN controller bugs. FSF outperformed DELTA, a previous state-of-the-art SDN fuzzing tool, in covering code coverage and produced discovered 146 of unique test inputs that trigger bugs residing in the controller.

Our main contributions are as follows.

1. We present a novel code coverage-driven fuzz testing algorithm tailored for testing an SDN system. The proposed technique leverages the coverage information from a controller to evolve test cases during a fuzzing campaign.
2. We implement the proposed algorithm in our prototype and evaluate it on the latest version of Floodlight. FSF produced unique 146 tests that trigger critical bugs, which affect the daily operations of an entire SDN system. To the best of our knowledge, our tool is the first feedback-driven SDN fuzzer.

2 Background and Motivation

Software-Defined Networking. A network system consists of two main planes: a data plane that forwards network packets between routers, and a control plane that computes network paths that forward packets. In traditional networks, the data plane and the control plane tightly coupled within a single device, which makes it hard to insert new functionalities or updates forwarding rules into the device. SDN has emerged to overcome this problem. It conceptually separates the control plane from the data plane, and they communicate with a protocol called OpenFlow [3] to exchange the routing information. Changing the flow table with a logically centralized controller is straightforward, thus easing the management of a network system.

SDN Fuzz Testing. *Fuzzing* or *fuzz testing* is a software testing technique that detects software security vulnerabilities and was first used by Miller in the early 1990s [22]. Fuzzing feeds adversarial inputs to a program under test and monitors resulting crashes [7,13,27,28,33].

Many studies have applied SDN fuzzing to enhance the security of SDN systems [9,15,20,30]. SDN fuzzing differs from general fuzz testing schemes. Due to the intrinsic nature of a SDN system that consists of diverse architectural components and their complicated interconnections, the following questions should be addressed when designing a fuzzing algorithm: (1) which components provide an input (*testing source*)? (2) which components take this input (*testing target*)? (3) which bugs or threats the algorithm find (*detection criteria*)?

DELTA [20], for example, implements a black box fuzzing technique that tests an entire SDN system (testing target) consisting of SDN applications, control channels, and hosts (testing source). In particular, their testing source mutates SDN control flow sequences and input values of the control flow. It employs seven detection criteria, including controller crash and switch-performance degradation. Another SDN fuzzer is BEADS [15]. BEADS drops, duplicates, delays, and changes OpenFlow packets from malicious switches and injects ARP packets from malicious hosts to test the controller. They validate OpenFlow error messages, network state, pair-wise connectivity, controller resource usage, and switches based on the detection criteria.

Limitations of Previous SDN Fuzzers. Previous fuzzers expose some of the erroneous behaviors of the SDN system, but their approaches have the limitation of only observing the input and output behaviors of the controller as a black box. They limit the mutations of the input data without runtime feedback or controller internal information (e.g., code coverage) that can evolve test inputs to trigger unexpected behaviors. In fact, none of the previous SDN testing methods has applied this feedback information to the mutations. Thus, the generated test input often fails to cover the diverse operations of an SDN controller. For example, to modify an ongoing OpenFlow message, BEADS uses the following strategy; modifies a *specific field* of a *specific type* of a OpenFlow message to a *specific value*. Since there is no automatic guidance or feedback to systematically select these values (field, type, modification values), they blindly select values that are likely to trigger inherent bugs. As a result, their approaches are not general to elicit diverse unexpected SDN controller behaviors.

3 Threat Model

In this paper, we assume a *southbound interface (SBI) attacker*. An SBI is an interface between an SDN controller and its connected switches. An SBI attacker is capable of compromising a switch or performing a man-in-the-middle attack that feeds malicious OpenFlow messages to the controller on the SBI. Several previous studies have shown the feasibility of compromising practical SDN switches by exploiting network operating systems using outdated software [24,25]. Once an attacker compromises a switch, she is able to generate an arbitrary OpenFlow message as controller input. It is also feasible for an SBI attacker to perform an MITM attack by exploiting the communication channel between the control plane and the data plane. The OpenFlow specification [3] recommends the use of SSL/TLS protection to protect OpenFlow messages. However, many existing controllers are packaged with the default setting that disables the SSL/TLS support to ease the initial deployment. Furthermore, the protection is frequently disabled due to its noticeable performance degradation [5,10].

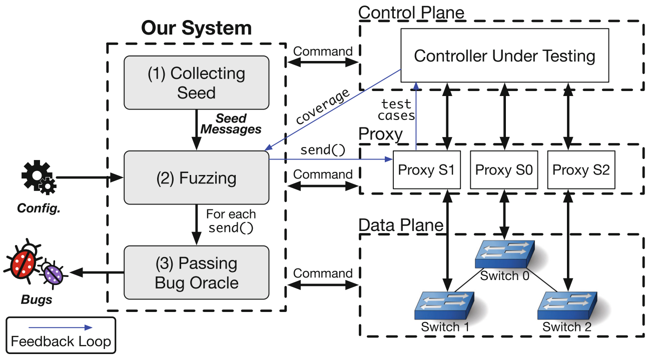


Fig. 1. FSF architecture.

4 Design

This section provides an overview of FSF and then describes each procedure for leveraging the coverage feedback from an SDN controller under testing, which makes FSF distinctive from other fuzzing tools, including DELTA and BEADS.

4.1 Overview

Figure 1 depicts the overall architecture of FSF. At a high level, it takes a set of user-configurable parameters. FSF then initiates a fuzz testing campaign. Once the campaign is completed, FSF reports a set of discovered bugs. FSF conducts a fuzz testing campaign in tandem with an SDN testing environment. The testing environment consists of a controller under testing, a set of SDN switches (data plane), and proxies that connect between the switches and their controller. FSF orchestrates these testing components to conduct a coverage-driven fuzzing testing campaign.

Testing Infrastructure. We implemented a *proxy* for each channel between the controller and its connected switches. Thus, each of these proxies is able to model the capability of an SBI attacker (Sect. 3).

A proxy has two roles: (1) forwarding benign OpenFlow messages, and (2) sending manipulated OpenFlow messages to a controller. The first role is required to maintain the continuous connections between the controller and its switches, which is an intrinsic characteristic of an SDN system. The proxy simply forwards incoming OpenFlow messages to avoid tampering with any ongoing transactions originating from benign switches. The second role models the capability of an SBI attacker. In the proxy, FSF mutates OpenFlow messages according to a given mutation policy and then sends these messages to elicit erroneous behaviors in the controller. The user-provided configuration parameters govern the distinction between benign and malicious switches. For instance, when Switch 1 in Fig. 1 is configured as a malicious switch, its corresponding Proxy 1 feeds diverse tests to elicit malicious behaviors in the controller.

Testing Procedure. Given a configuration file, FSF conducts three phases. Phase I collects seed messages by monitoring ongoing messages or by generating arbitrary OpenFlow messages (Sect. 4.2). Based on the collected seed messages, Phase II conducts a fuzzing campaign (Sect. 4.3). Specifically, it splits seed messages into several message sets, where the length of each is governed by a given configuration parameter. For each message set, FSF randomly mutates those messages in the set and feeds the mutated set to the controller. It then obtains the instruction coverage of the controller and leverages the feedback for subsequent fuzzing iterations. Phase III determines whether each mutated message set triggers bugs residing in the controller, thus serving as a bug oracle (Sect. 4.4).

4.2 Collecting Seed Messages

FSF begins a fuzzing campaign by collecting seed messages, which are observed from configured proxies. Because our testing target is an SDN controller, we only consider OpenFlow messages from a malicious switch to the controller. To diversify seed messages, we first identified what types of OpenFlow messages belong to switch-to-controller flows. According to the OpenFlow v1.3 specification [3], of 30 message types, 15 types are switch-to-controller OpenFlow messages. Note that DELTA [20] and BEADS [15] only covered six message types out of these 15 types (40%), which the `pingall` and `iperf` commands are able to trigger.

FSF considers all the 15 types of switch-to-controller messages. This means that FSF covers more diverse code spots in a target controller, thereby increasing the possibility of eliciting unexpected behaviors. We collect seed messages through three methods: (1) capturing packets in the stand-by state; (2) capturing packets after executing pre-defined commands to the control plane or data plane; and (3) generating packets according to the OpenFlow grammar specification.

Stand-By State. A proxy gathers seed messages by capturing network packets when a controller is in the stand-by state, awaiting incoming OpenFlow messages. In this state, the controller is involved only in (1) handshaking procedures that establish connections between the controller and its switches and in (2) checking the stability of established connections.

Commands Sent to the Control or Data Plane. To collect diverse seed messages, FSF lets the controller and its switches execute pre-defined commands. It then captures the packets caused by the exercised commands. FSF sends commands to the controller via its REST API. For instance, FSF inserts a flow rule to a switch and then removes it to generate `FLOW_REMOVED` messages. It also performs commands to the data plane using switch command-line interface (CLI) or host CLI. For instance, FSF asks a switch to disconnect a connection and reconnect it on a specific switch port to capture a `PORT_STATUS` message.

Generating Packets. The aforementioned methods are unable to cover the remaining four types. For these messages, FSF generates random messages according to the OpenFlow v1.3 specification [3]. When generating these messages, FSF identifies data fields to fill as well as their constraints and then assigns random values to generate seed messages.

Algorithm 1. Feedback-driven SDN Fuzzing Algorithm.

```
1  function Fuzzing($conf$, $seed\_msgs$, $controller$, $proxy$, $switch$)
2      $cov\_base \leftarrow$ ResetComponent($controller$, $proxy$, $switch$)
3      for $i \leftarrow 0$ to $conf.size_q$ do
4          $subset \leftarrow$ RandomSample($seed\_msgs$, $conf.size_s$)
5          $Q$.enqueue($subset$, 0)
6      while $Q \neq \emptyset$ do
7          $subset$, $counter \leftarrow Q$.dequeue()
8          $test\_cases \leftarrow$ MutateSubset($subset$)
9          $proxy$.send($test\_cases$)
10         $unseen\_msgs \leftarrow$ GetUnseenMsgs()
11         $found\_bug$, $cov \leftarrow$ Evaluate($controller$, $proxy$, $switch$)
12         $bugs$.append($found\_bug$)
13         if $cov\_base < cov$ then
14             $mutated\_m$, $non\_mutated\_m \leftarrow$ SplitMsgs($test\_cases$)
15             $Q$.enqueue($mutated\_m$.append($unseen\_msgs$, $conf.size_s$),0)
16             $Q$.enqueue($non\_mutated\_m$.append($unseen\_msgs$, $conf.size_s$),0)
17             $cov\_base \leftarrow cov$
18         else if $counter < conf.threshold_c$ then
19             $Q$.enqueue($subset$, $counter + 1$)
20     return $bugs$
```

4.3 Coverage-Driven SDN Fuzzing

Given a set of seed messages, FSF performs code coverage-driven fuzz testing by leveraging the coverage feedback from a target SDN controller. Algorithm 1 describes the overall fuzzing procedure. The underlying idea is to discard messages that caused no increase of code coverage and to give more chances to messages that already increased code coverage. Our assumption is that a message that helped increase code coverage is likely to be a good seed for further mutations, increasing code coverage.

The algorithm starts with a configuration file (*conf*), seed messages (*seed_msgs*), and instances of a controller, proxies, and switches.

FSF begins by resetting all the components in a testing environment and computing the baseline code coverage of the *controller* in Ln 2. Lns 4–5 initialize a test queue Q by assigning multiple input subsets, each of which contains randomly sampled messages from *seed_msgs*. Ln 5 enqueues each subset with its counter value, which is later used for discarding subsets tested multiple times. The size of Q and *subset* is configurable by setting $conf.size_s$ and $conf.size_q$.

For each iteration, FSF mutates a message subset dequeued from Q, sends the mutated messages in this subset, and evolves the message set by leveraging the feedback of a code coverage difference from the target controller, as Lns 6–19 show. The `MutateSubset` function in Ln 8 mutates randomly chosen messages in the subset. There are various ways to mutate messages such as flipping multiple bytes or inserting dummy bytes but, based on the results of our empirical study (Sect. 5.1), we selected the flipping multiple bits operation. Ln 9 sends messages in *test_cases* to the controller one by one. Note that there exist certain messages that require the precedence of a request from the controller (e.g., `MULTIPART_REPLY`). To address this, FSF invokes a REST API to incur the corresponding request (e.g., `MULTIPART_REQUEST`) from the SDN controller before sending a response message from the proxy as its replying message.

The `GetUnseenMsgs` function in Ln 10 collects previously unobserved messages. The unseen messages are switch-to-controller messages that occur after performing Ln 9. Their contents are unique so that FSF has not observed beforehand. Adding unseen messages to test cases improves the diversity of seed messages on which the mutations are performed. Therefore, we put these unseen messages in the queue later in Lns 15 and 16. Ln 11 performs an evaluation to determine whether bugs are triggered through the implemented bug oracle (Sect. 4.4) and to measure the cumulative code coverage in the controller.

If the mutated *test_cases* successfully hits new code space in the controller, FSF enqueues it to Q (Ln 13–17). As *test_cases* consists of mutated and non-mutated messages, FSF splits them by invoking the `SplitMsgs` function in Ln 14. Ln 15 enqueues the purely mutated messages (*mutated_m*) to Q. Because the size of the *mutated_m* is smaller than the $conf.size_s$, the number of ($conf.size_s$-*len(mutated_m)*) of *unseen_msgs* is randomly selected and appended to *mutate_m*. The same process is used to enqueue non-mutated messages (*non_mutated_m*) because there is a chance that they contribute to increasing the code coverage when they are mutated later (Ln 16). We designed the algorithm to refine messages by separating mutated messages from non-mutated messages since non-mutated messages were already tried in previous iterations. Thus, we create a new message set by adding several unseen messages to mutated messages, which help the odd of increasing code coverage. At the same time, it gives another chance to non-mutated messages by creating a new message set with additional unseen messages.

If the *test_cases* does not touch any new code spots, FSF does not discard it immediately. We give it more chances to be used in further testing by putting them to Q with an increased counter, as shown in Ln 19. The value of the counter threshold $conf.size_s$ is also determined by a given configuration file.

4.4 Bug Oracle

The bug oracle determines whether test inputs trigger bugs or not by monitoring the components of the testing environment. We describe four standards to implement a bug oracle.

Controller Process Termination. FSF checks whether the controller process has terminated or crashed. In SDN, because multiple switches are continuously connected to an SDN controller, the controller's abrupt termination causes a denial of service for the entire SDN network.

Control Plane Resource Exhaustion. FSF also checks whether the CPU usage of the control plane process suddenly surges after sending testing messages. The abrupt increased usage when compared to a benign baseline indicates an opportunity for a denial of service, which impairs the controller's ability to deal with OpenFlow messages. Therefore, a bug that exhausts the CPU usage of the control plane can drop the QoS of the entire network.

Benign Switch Disconnection. We consider whether a benign switch is disconnected from the controller. A benign switch, e.g., switch 0 in Fig. 1, is a switch that is not compromised by the SBI attacker. When a mutated switch-to-controller message from a compromised switch contributes to other unrelated switch-to-controller channels being disconnected, the bug oracle considers it a DoS for benign switches.

Inter-host Communication Disconnection. The final bug oracle standard is a pair-wise connectivity test to check whether the data plane network works well. In particular, our testing scheme uses `pingall` command from the hosts to check that all hosts are reachable from all other hosts. This is effective in detecting message spoofing attacks and connectivity attacks.

5 Evaluation

We evaluated FSF using a real-world SDN system. We preliminary analyzed the efficacy of deployed mutation operations (Sect. 5.1), and measured the performance of FSF for improving testing code coverage (Sect. 5.2) and finding bugs (Sect. 5.3).

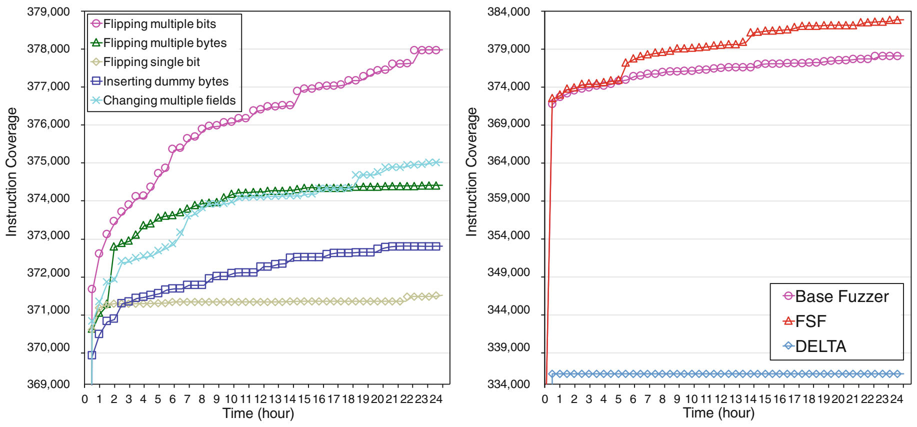
Experimental Setup. We evaluated FSF on the latest version (v1.2) of Floodlight [1]. We setup our system within a Virtual Box with an Intel core i7-9700K CPU and 9 GB of RAM. To measure the instruction coverage of the controller under testing, we used JaCoCo [2], a Java code coverage library. JaCoCo conducts online instrumentation in which instrumentation code is inserted in Java byte code when Java classes are loaded into main memory. We set the timeout to be 24 h for each fuzz testing campaign, and measured the cumulative instruction coverage of the controller during the testing time.

5.1 Operation Significance

To compare the efficacy of different mutation operations, we implemented a base fuzzer that only uses five mutation operations but leverages no feedback from the controller. This fuzzer takes following procedures: (1) setting the proxy to mutates observed switch-to-controller messages with a 10% probability; (2) it periodically generating multiple switch-to-controller messages according to the procedure described in Sect. 4.2.

Mutation Operations. We designed five mutation operations as follows. Note that each operation is designed to explore the diverse control flow of the controller under testing.

(a) *Flipping multiple bits*: it selects and flips multiple random bits. The number of bits to be mutated is randomly chosen from 1 to one tenth of all available bits.
(b) *Flipping multiple bytes*: it selects and flips the selected bytes. The number of mutated bytes is randomly selected from 1 to 10.



(a) Base fuzzer with five mutations. (b) FSF vs. Base Fuzzer vs. DELTA.

Fig. 2. The instruction coverage of different fuzzers running for 24 h.

(c) *Flipping single bit*: it flips one randomly chosen bit in the message.
(d) *Inserting dummy bytes*: it inserts random bytes at the random position within the message. The length of the dummy is randomly chosen from one byte to 71,680 bytes.
(e) *Changing multiple fields*: it selects multiple random message fields, and change the selected field values to random values while preserving the type constraints according to the OpenFlow specification [3]. The number of mutated fields is randomly set from 1 to 20% of all available fields.

We compared the instruction coverage of the controller when each single mutation operation was applied. The objective here is to gauge how each mutation affects on increasing the instruction coverage of the SDN controller under testing.

Figure 2a shows the instruction coverage of preliminary base fuzzer with five operations for 24 h. Note that the *flipping multiple bits* operation touched the most number of instructions, 377,970 instructions in total. Based on the above observation, we adopted the *flipping multiple bits* operation to FSF.

We further analyzed root causes of observed coverage differences among the different mutation strategies. Floodlight internally uses `OpenFlowJ`, which parses a given OpenFlow messages according to the OpenFlow protocol specification. When a received message does not meet this specification, `OpenFlowJ` raises an exception, hindering to reach a deeper code region. Furthermore, each message field in a OpenFlow message requires a different primitive type, such as `uint8`, `uint32`, and `uint64`. Non-compliance of such primitive type requirements will also cause low code coverage. The three mutations of *changing multiple fields*, *flipping multiple bytes*, and *inserting dummy bytes* are more likely to generate OpenFlow messages that do not satisfy the specification nor the primitive type constraints. For instance, *flipping multiple bytes* often breaks a required type constraint. For example, the `Capabilities` field value in a `FEATURES_REPLY` message should have one of the following values: `0x00000000`,`0x00000001`, `0x00000002`, `0x00000004`, `0x00000008`, `0x00000020`, `0x00000040`, and `0x00000100`. When flipping multiple bytes of this message, this mutation is able to generate a message with `0x00000011`, which is not a valid OpenFlow message.

Flipping multiple bits is a simple but effective mutation strategy in practice. It is widely used in many fuzzers [12,14,33]. As shown in Fig. 2a, *flipping multiple bits* also was the most effective mutation strategy than other ones in terms of improving code coverage. Bit flipping causes no significant change in its target field, thus resulting in a high chance of not violating the aforementioned constraints. Also, *flipping multiple bits* is better than *flipping single bit* in generating more diverse tests.

5.2 Coverage Improvement

We compared the coverage improvement of FSF with that of two other fuzzers: (1) a preliminary base fuzzer with the *flipping multiple bits* operation (Sect. 5.1), and (2) DELTA [20], a state-of-the-art SDN security assessment framework. Unfortunately, the DELTA project [20] did not contain a fuzz testing function at the time of writing. The project does, however, support penetration testing, a key task of DELTA with 40 known attack scenarios. To compare our tool with DELTA, we measured the number of covered instructions after conducting penetration testing.

Figure 2b shows the instruction coverage of different fuzzers. We observed that FSF and the base fuzzer significantly outperformed DELTA on instruction coverage. Recall from Sect. 4.2, both fuzzers leverage all of the switch-to-controller OpenFlow message types, while DELTA only relies on a set of limited known attack scenarios. Therefore, we concluded that it is important to have a diverse set of seed messages to conduct comprehensive testing of an SDN system.

We also observed that FSF touched 4,835 more instructions than base fuzzer. As stated in Sect. 5.1, the main difference between the preliminary base fuzzer and FSF is the existence of a coverage feedback loop. Therefore, we note that the difference in the instruction coverage comes from the coverage feedback iteration.

5.3 Bugs Found

We further analyzed FSF in terms of its bug finding ability. Recall from Sect. 4.4 that we consider four types of bugs as our detection criteria: (1) controller process termination, (2) control plane resource exhaustion, (3) benign switch disconnection, and (4) inter-host communication disconnection. Table 1 summarizes the number of test instances that trigger bugs residing in the target SDN controller. We counted the number of distinct tests that trigger the bugs based on two different metrics that each column represents.

The second column in Table 1 shows the number of mutated message sets that trigger the corresponding bug. In total, 1 controller process termination,

Table 1. The number of test instances that trigger bugs.

Bug oracle	Total test instances	Unique test instances
Controller process termination	1	0
Control plane resource exhaustion	0	0
Benign switch disconnection	198	132
Inter-host communication disconnection	18	14

198 benign switch disconnection, and 18 inter-host communication disconnection bugs with their test instances were reported during the 24 h of a fuzzing campaign. We observed that FSF successfully triggered three types of bugs. Benign switch disconnection imposes a DoS for other benign switches. Hosts can not communicate with each other when disconnection occurs. Furthermore, the controller process crashes cause a DoS for the entire SDN network.

As a postmortem analysis, we extracted mutated message sets (i.e., *test_cases*) that successfully triggered bugs, send each of it in initial testing infrastructure, and see whether same bugs were triggered or not. Additionally, we minimize the reproducible *test_cases* by leveraging delta debugging [30,34] technique to get a minimized message set that causes the same bug. Finally, we compare minimized subsequence with each other in terms of sequence length, message type, and message length to count the number of unique message sets. As the third column in Table 1 shows, FSF found unique 146 test instances.

We further analyzed the 14 minimized unique instances that triggered the inter-host communication disconnection and identified two unique bugs via conducting postmortem analyses on the controller source with the input instances. One bug caused the failure of `ping` operations between hosts under benign switches. When a malicious switch sends an identified attack payload, this packet contributes hosts under benign switches to disconnecting from the network, causing a remote denial of service. Another bug caused not only the failure of `ping` operations but also the flooding of `PACKET_OUT` messages, demonstrating a feasible denial of service. Both bugs got assigned CVE numbers and have remained in the reserved status at the time of writing. On the other hand, the instances triggered the controller termination was not reproducible in the postmortem analysis.

6 Discussion

FSF only supports the latest version of Floodlight. However, it is straightforward to apply it to other types of SDN controllers, e.g., POX, ONOS, ODL, because the core idea of FSF in leveraging code coverage of the controller is indeed platform agnostic.

FSF only adopted the *flipping multiple bits* mutation. However, we believe that consolidating multiple mutations will bring a better result in terms of finding bugs as well as improving code coverage. Also, deploying combinatorial testing [8] that mutates fields with only known interesting values helps prune unnecessary test cases, enabling an efficient fuzzing campaign.

Note that the mutation ratio for the *flipping multiple bits* mutation is an important parameter to effectively trigger bugs [7]. Thus, exploring optimal mutation ratios for a fuzzing campaign can be a promising future direction of research.

We only implemented the bug oracles that detect the availability of an underlying SDN network. However, the bug oracles can be extended to check the confidentiality [21] and integrity of a target SDN network.

7 Related Work

SDN Attacks and Defenses. Many previous studies have presented attacks and defenses that can occur in SDN [5,16,17,19,26]. Benton *et al.* [5] presented the feasibility of an MITM attack in control channels due to the lack of SSL/TLS adoptions by vendors. Kazemian *et al.* [16] proposed an SDN system hardening tool. They identify all state changes in the communication channel, and check network policies in real time based on the header space analysis.

There exist previous survey studies to summarize the various SDN security issues [17,19,30,32]. Scott-Hayward *et al.* [30] proposed possible DoS attacks due to the limitations due to the design decisions, including centralized controllers and network flow tables. Flow Wars [32] presented a survey for the possible attacks with its attack vectors. They found 14 attacks and 22 concrete attack scenarios. For each attack classification, they suggested defense mechanisms.

SDN Fuzzing. Based on these SDN security problems, prior studies have been actively conducted on implementing automated testing tools to identify vulnerabilities [9,15,20,30]. DELTA [20] deploys a blackbox fuzz testing technique. It models the fuzzing input source to be malicious applications, control channels, or hosts. The input sources mutate SDN control flow sequences or input values of such control flows. BEADS [15] assumes malicious switches and hosts. It supports various mutations, including dropping, duplicating, delaying, and changing OpenFlow messages as well as ARP injection operations to elicit erroneous behaviors in an SDN system. AIM-SDN [9] focuses on identifying problematic data inconsistencies between data stores, which may exist in an SDN system. It uses REST API and SBI to perform fuzz testing to find data inconsistency problems.

Acknowledgements. We thank anonymous reviewers for their helpful feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00254, SDN security technology development).

References

1. Floodlight. http://www.projectfloodlight.org/floodlight
2. JaCoCo: Java code coverage library. https://www.jacoco.org/jacoco/
3. Openflow switch specification: Version 1.3.1. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf
4. Basta, A., Kellerer, W., Hoffmann, M., Morper, H.J., Hoffmann, K.: Applying NFV and SDN to LTE mobile core gateways, the functions placement problem. In: Proceedings of the Workshop on All Things Cellular: Operations, Applications and Challenges, pp. 33–38 (2014)
5. Benton, K., Camp, L.J., Small, C.: Openflow vulnerability assessment. In: Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, pp. 151–152 (2013)
6. Bertaux, L., et al.: Software defined networking and virtualization for broadband satellite networks. IEEE Commun. Mag. **53**(3), 54–60 (2015)
7. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 725–741 (2015)
8. Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Softw. **13**(5), 83–88 (1996)
9. Dixit, V.H., Doupé, A., Shoshitaishvili, Y., Zhao, Z., Ahn, G.J.: AIM-SDN: attacking information mismanagement in SDN-datastores. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 664–676 (2018)
10. Durner, R., Kellerer, W.: The cost of security in the SDN control plane. In: Proceedings of the ACM CoNEXT Student Workshop (2015)
11. Flauzac, O., González, C., Hachani, A., Nolot, F.: SDN based architecture for IoT and improvement of the security. In: Proceedings of the IEEE International Conference on Advanced Information Networking and Applications Workshops, pp. 688–693 (2015)
12. Hocevar, S.: zzuf. https://github.com/samhocevar/zzuf
13. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proceedings of the USENIX Security Symposium, pp. 445–458 (2012)
14. Householder, A.D., Foote, J.M.: Probability-based parameter selection for black-box fuzz testing. Technical report, CMU/SEI-2012-TN-019, CERT (2012)
15. Jero, S., Bu, X., Nita-Rotaru, C., Okhravi, H., Skowyra, R., Fahmy, S.: BEADS: automated attack discovery in openflow-based SDN systems. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) RAID 2017. LNCS, vol. 10453, pp. 311–333. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66332-6_14
16. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, pp. 99–111 (2013)
17. Klöti, R., Kotronis, V., Smith, P.: Openflow: a security analysis. In: Proceedings of the IEEE International Conference on Network Protocols, pp. 1–6 (2016)
18. Kreutz, D., Ramos, F., Verissimo, P.: Towards secure and dependable software-defined networks. In: Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, pp. 55–60 (2013)
19. Kreutz, D., Ramos, F.M., Verissimo, P., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2015)

20. Lee, S., Yoon, C., Lee, C., Shin, S., Yegneswaran, V., Porras, P.A.: DELTA: a security assessment framework for software-defined networks. In: Proceedings of the Network and Distributed System Security Symposium (2017)
21. Lei, X., Huang, J., Hong, S., Zhang, J., Gu, G.: Attacking the brain: races in the SDN control plane. In: Proceedings of the USENIX Security Symposium, pp. 451–468 (2017)
22. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
23. Ojo, M., Adami, D., Giordano, S.: A SDN-IoT architecture with NFV implementation. In: Proceedings of the IEEE Globecom Workshops, pp. 1–6 (2016)
24. Pickett, G.: Abusing software defined networks. In: Proceedings of the Black Hat EU (2014)
25. Pickett, G.: Staying persistent in software defined networks. Black Hat Briefings (2015)
26. Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G.: A security enforcement kernel for openflow networks. In: Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, pp. 121–126 (2012)
27. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (2017)
28. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: Proceedings of the USENIX Security Symposium, pp. 861–875 (2014)
29. Rückert, J., Bifulco, R., Rizwan-Ul-Haq, M., Kolbe, H.J., Hausheer, D.: Flexible traffic management in broadband access networks using software defined networking. In: Proceedings of the IEEE Network Operations and Management Symposium, pp. 1–8 (2014)
30. Scott, C., et al.: Troubleshooting blackbox SDN control software with minimal causal sequences. ACM SIGCOMM Comput. Commun. Rev. **44**(4), 395–406 (2015)
31. Trivisonno, R., Guerzoni, R., Vaishnavi, I., Soldani, D.: SDN-based 5G mobile networks: architecture, functions, procedures and backward compatibility. Trans. Emerg. Telecommun. Technol. **26**(1), 82–92 (2015)
32. Yoon, C., et al.: Flow wars: systemizing the attack surface and defenses in software-defined networks. IEEE/ACM Trans. Netw. **25**(6), 3514–3530 (2017)
33. Zalewski, M.: American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/
34. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Proceedings of the ACM SIGSOFT Software Engineering Notes, pp. 253–267 (1999)