Learning Failure-Inducing Models for Testing Software-Defined Networks

RAPHAËL OLLANDO, University of Luxembourg, Luxembourg
SEUNG YEOB SHIN, University of Luxembourg, Luxembourg
LIONEL C. BRIAND*, Lero SFI Centre for Software Research, University of Limerick, Ireland and University of Ottawa, Canada

Software-defined networks (SDN) enable flexible and effective communication systems that are managed by centralized software controllers. However, such a controller can undermine the underlying communication network of an SDN-based system and thus must be carefully tested. When an SDN-based system fails, in order to address such a failure, engineers need to precisely understand the conditions under which it occurs. In this article, we introduce a machine learning-guided fuzzing method, named FuzzSDN, aiming at both (1) generating effective test data leading to failures in SDN-based systems and (2) learning accurate failure-inducing models that characterize conditions under which such system fails. To our knowledge, no existing work simultaneously addresses these two objectives for SDNs. We evaluate FuzzSDN by applying it to systems controlled by two open-source SDN controllers. Further, we compare FuzzSDN with two state-of-the-art methods for fuzzing SDNs and two baselines for learning failure-inducing models. Our results show that (1) compared to the state-of-the-art methods, FuzzSDN generates at least 12 times more failures, within the same time budget, with a controller that is fairly robust to fuzzing and (2) our failure-inducing models have, on average, a precision of 98% and a recall of 86%, significantly outperforming the baselines.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Networks → Programmable networks**.

Additional Key Words and Phrases: Software-Defined Networks, Software Testing, Fuzzing, Machine Learning

ACM Reference Format:
Raphaël Ollando, Seung Yeob Shin, and Lionel C. Briand. 2024. Learning Failure-Inducing Models for Testing Software-Defined Networks. 1, 1 (February 2024), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software-defined networks (SDN) [25] have emerged to enable programmable networks that allow system operators to manage their systems in a flexible and efficient way. SDNs have been widely deployed in many application domains, such as data centers [18, 60], the Internet of Things [51, 55], and satellite communications [20, 37]. The main idea behind SDNs is to transfer the control of networks from localized, fixed-behavior controllers distributed over a set of network switches

*Part of this work was done when he was affiliated with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) of the University of Luxembourg.

Authors' addresses: Raphaël Ollando, raphael.ollando@uni.lu, University of Luxembourg, 29 Avenue John F. Kennedy, Luxembourg, 1859, Luxembourg; Seung Yeob Shin, seungyeob.shin@uni.lu, University of Luxembourg, 29 Avenue John F. Kennedy, Luxembourg, 1859, Luxembourg; Lionel C. Briand, lionel.briand@lero.ie, Lero SFI Centre for Software Research, University of Limerick, Tierney building, Limerick, V94 NYD3, Ireland and University of Ottawa, 800 King Edward Avenue, Ottawa, ON K1N 6N5, Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Association for Computing Machinery.
XXXX-XXXX/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(in traditional networks) to a logically centralized and programmable software controller. With complex software being an integral part of SDNs, developing SDN-based systems (SDN-systems), e.g., data centers, entails interdisciplinary considerations, including software engineering.

In the context of developing SDN-systems, software testing becomes even more important and challenging when compared to what is required in traditional networks that provide static and predictable operations. In particular, even though the centralized controller in an SDN-system enables flexible and efficient services, it can undermine the entire communication network it manages. A software controller presents new attack surfaces that allow malicious users to manipulate the systems [16, 53]. For example, if malicious users intercept and poison communication in the system (using ARP spoofing [14]), such attacks broadly impact the system due to its centralized control. Furthermore, the centralized controller interacts with diverse kinds of components such as applications and network switches, which are typically developed by different vendors. Hence, the controller is prone to receiving unexpected inputs provided by applications, switches, or malicious users, which may cause system failures, e.g., communication breakdown.

To test an SDN controller, engineers need first to explore its possible input space, which is very large. A controller takes as input a stream of control messages which are encoded according to an SDN communication protocol (e.g., OpenFlow [45]). For example, if a control message is encoded with OpenFlow, it can have 2^{2040} distinct values [45]. Second, engineers need to understand the characteristics of test data, i.e., control messages, that cause system failures. However, manually inspecting test data that cause failures is time-consuming and error-prone. Furthermore, misunderstanding such causes typically leads to unreliable fixes.

There are a number of prior research strands that aim at testing SDN-systems [1, 2, 5, 11, 29, 35, 43, 62, 64]. Most of them come from the network research field and focus on security testing relying on domain knowledge, e.g., known attack scenarios [35]. The most pertinent research works applied fuzzing techniques to different components of SDN-systems. For example, RE-CHECKER [62] fuzzes RESTful services provided by SDN controllers. SwitchV [1] relies on fuzzing and symbolic execution to test SDN switches. BEADS [29] tests SDN controllers by being aware of the OpenFlow specification. However, none of these fuzzing techniques employ interpretable machine learning techniques to guide their fuzzing process and to provide models that characterize failure-inducing conditions. Even though the software engineering community has introduced numerous testing methods, testing SDN-systems has gained little attention. The most pertinent research lines have proposed techniques for learning-based fuzzing [10, 23, 65] and abstracting failure-inducing inputs [24, 30] to efficiently explore the input space and characterize effective test data that cause system failures. Learn@Fuzz [23] employs neural-network-based learning methods for building a model of PDF objects for grammar-based fuzzing. A prior learning-guided fuzzing technique, named smart fuzzing [10], relies on deep learning techniques to test systems controlled by programmable logic controllers (PLC). SeqFuzzer [65] uses deep learning techniques to infer communication protocols underlying PLC systems and generate fuzzed messages. However, deep learning techniques are not suitable to characterize failure-inducing test data as they do not provide interpretable models. Furthermore, these techniques do not account for the specificities of SDNs, such as SDN architecture and communication protocol. Existing work on abstracting failure-inducing inputs [24, 30] targets software programs that take as input strings such as command-line utilities (e.g., find and grep), which are significantly different from SDN-systems. In summary, no existing work simultaneously tackles the problem of efficiently exploring the input space and accurately characterizing failure-inducing test data while accounting for the specificities of SDNs.

Contributions. In this article, we propose FuzzSDN, a machine learning-guided Fuzzing method for testing SDN-systems. In particular, FuzzSDN targets software controllers deployed in SDN-systems. FuzzSDN relies on fuzzing guided by machine learning (ML) to both (1) efficiently explore

the test input space of an SDN-system's controller (generate test data leading to system failures) and (2) learn failure-inducing models that characterize input conditions under which the system fails. This is done in a synergistic manner where models guide test generation and the latter also aims at improving the models. A failure-inducing model is practically useful [24] for the following reasons: (1) It facilitates the diagnosis of system failures. FuzzSDN provides engineers with an interpretable model specifying how likely are failures to occur, e.g., the system fails when a control message is encoded using OpenFlow V1.0 and contains IP packets, thus providing concrete conditions under which a system will probably fail. Such conditions are much easier to analyze than a large set of individual failures. (2) A failure-inducing model enables engineers to validate their fixes. Engineers can fix and test their code against the generated test data set. A failure-inducing model can also be used as a test data generator to reproduce the system failures captured in the model. Hence, engineers can better validate their fixes using an extended test data set.

We evaluated FuzzSDN by applying it to several systems controlled by well-known open-source SDN controllers: ONOS [4] and RYU [54]. In addition, we compared FuzzSDN with two state-of-the-art methods (i.e., DELTA [35] and BEADS [29]) that generate test data for SDN contollers and two baselines that learn failure-inducing models. As baselines, we extended DELTA and BEADS to produce failure-inducing models, since they were not originally designed for that purpose but were nevertheless our best options. Our experiment results show that, compared to state-of-the-art methods, FuzzSDN generates at least 12 times more failing control messages, within the same time budget, with a controller that is fairly robust to fuzzing. FuzzSDN also produces accurate failure-inducing models with, on average, a precision of 98% and a recall of 86%, which significantly outperform models inferred by the two baselines. Furthermore, FuzzSDN produces failure-inducing conditions that are consistent with those reported in the literature [29], indicating that FuzzSDN is a promising solution for automatically characterizing failure-inducing conditions and thus reducing the effort needed for manually analyzing test results. Last, FuzzSDN is applicable to systems with large networks as its performance does not depend on network size. Our detailed evaluation results and the FuzzSDN tool are available online [44].

Organization. The rest of this article is organized as follows: Section 2 introduces the background and defines the specific problem of learning failure-inducing models for testing SDN-systems. Section 3 describes FuzzSDN. Section 4 evaluates FuzzSDN in a large empirical study. Section 5 compares FuzzSDN with related work. Section 6 concludes this article.

2 BACKGROUND AND PROBLEM DESCRIPTION

In this section, we describe the fundamental concepts of an SDN-system. We then discuss the problem of identifying input conditions under which the system under test fails.

Controller. In general, an SDN-system is composed of two layers: infrastructure and control layers. The infrastructure layer contains physical components, such as switches and links, that build a physical network transferring data flows such as audio and video data streams. The control layer is a software component developed by software engineers to meet the system's requirements. Specifically, the software controller manages the physical components in the infrastructure layer to implement, for example, system-specific data forwarding, security management, and failure recovery algorithms. Figure 1 shows an SDN topology example, containing a controller in the control layer along with three switches and four hosts (such as servers and clients) in the infrastructure layer. Note that this controller is the target of our testing, which is not present in traditional networking systems. In this article, we simplified the SDN layered architecture [25] to clearly explain our contributions by abstracting out network-specific details, e.g., SDN southbound interfaces, that are not important to present this work.

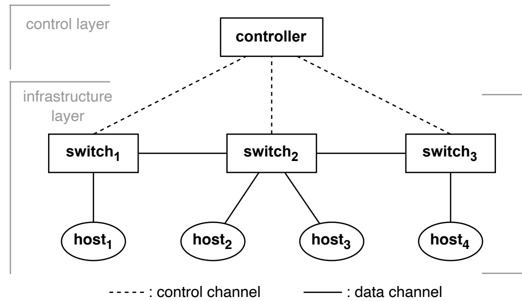

Fig. 1. An SDN topology example containing one controller, three switches, and four hosts.

In traditional networking systems, controllers are typically embedded in the physical devices, operating based on their local configurations. This decentralized control approach contrasts with the centralized control of SDN-systems. The SDN architecture centralizes decision-making, aiming for enhanced efficiency and dynamic network management. However, this can also lead to SDN-specific failures caused by such central point of control.

Message. The SDN controller and switches in a network communicate by exchanging control messages. A control message is encoded as a sequence of values following a specific communication protocol. For instance, OpenFlow [45] is a de facto standard communications protocol used in many SDN-systems [32], enabling communication between the control and infrastructure layers. To exercise the behavior of the controller under test, therefore, testing explores the space of possible control messages.

In traditional networking systems, communication between devices usually employs well-established transport protocols, such as UDP [48] and TCP [49], for data transmission. For exchanging routing and state information, routers and switches use standard routing protocols, such as OSPF [42] and BGP [52]. Since these protocols were designed to serve specific purposes in the context of traditional static networks, the messages they encode are more similar and simpler than those used in SDN. In contrast, SDN messages, such as those in OpenFlow, carry a variety of data including flow setups, modifications, and statistics, allowing for more flexibility in network services. However, this also entails a wider range of potential failures that should be accounted for during testing.

Failure. Like other software components, SDN controllers may have faults that can lead to service failures perceivable by users. These failures can manifest in various forms in the context of SDN-systems. Specifically, previous studies on SDN testing [2, 29, 34, 35, 56] have investigated the following failures relevant to SDN controllers: (1) Controller-switch disconnection. When communications between the controller and the switches are unexpectedly disconnected, the system obviously cannot operate as intended. For example, if a switch does not receive timely commands from the controller due to this disconnection, the switch relies on outdated forwarding rules installed earlier, possibly leading to dropping data flows entirely. (2) SDN operation stall. This failure refers to situations where the required execution of an SDN operation is unexpectedly prevented or delayed. For example, an SDN controller might stall the installation of forwarding rules, affecting the construction of the communication path in the SDN. (3) Incorrect understanding of network status. SDN controllers rely on a centralized view of the network to make control decisions. If this centralized view becomes inconsistent with the actual network, regardless of the reasons, e.g., receiving outdated or incorrect data from the network switches, the controller may fail to make decisions that reflect the network's actual conditions. For example, if a link between

two switches is broken, but the controller's view still considers it operational, the controller fails to instruct the switches to reroute data flows around the broken link, resulting in dropped data flows. (4) Overutilization of resources. SDN controllers may overutilize their processing and memory units due to various reasons, such as handling an unexpectedly high volume of requests. For example, such a failure caused by a distributed denial-of-service (DDoS) attack [40] targeting the SDN controller can lead to slower response times and potential service outages. We note that, in addition to failures specific to SDN controllers, failures observed in traditional networks, such as communication breakdowns among hosts (e.g., data servers and clients) and network performance degradation, are also relevant in SDN-systems as its infrastructure layer usually employs traditional network protocols, such as TCP, UDP, and IP.

In traditional networking systems, failures tend to be localized. If a router or switch fails, only its directly connected neighbours (i.e., localized segments of the network) are affected. Similarly, if a server or client fails, only the applications and users directly reliant on that specific device are impacted. In contrast, SDN-systems have a centralized failure point in the controller. If the controller crashes or loses connection with the switches, the entire network can be affected. This central point of potential failure makes rigorous testing essential to ensure robustness and reliability.

Problem. When developing and operating an SDN-system, engineers must handle system failures that are triggered by unexpected control messages. In particular, engineers need to ensure that the system behaves in an acceptable way in the presence of failures. In an SDN-system, its controller is prone to receiving unexpected control messages from switches in the system [11]. For example, network switches, which are typically developed by different vendors, may send control messages that fall outside the scope of the controller's expectations. Such unexpected messages can also be sent by the switches due to various reasons such as malfunctions and bugs in the switches, as well as inconsistent implementations of a communication protocol between the controller and switches [34]. Furthermore, prior security assessments of SDNs have found several attack surfaces, leading to vulnerable applications and communications protocols that enable malicious actors to send manipulated messages over an SDN [29, 35].

When a failure occurs in an SDN-system, engineers need to determine the conditions under which such failures occur. These conditions define a set of control messages that cause the failure. Identifying such conditions in a precise and interpretable form is in practice useful, as it enables engineers to diagnose the failure with a clear understanding of the conditions that induce it. In addition, engineers can produce an extended set of control messages by utilizing the identified conditions to test the system after making changes to address the failure. In general, any fix should properly address other control messages that induce the same failure. Our work aims to both effectively test an SDN-system's controller by identifying control messages that lead to system failures, and then automatically identify an accurate failure-inducing model that characterizes conditions under which the SDN-system fails. Such conditions define a set of failure-inducing control messages.

3 APPROACH

Figure 2 shows an overview of our ML-guided Fuzzing method for testing SDN-systems (FuzzSDN). Specifically, FuzzSDN relies on fuzzing and ML techniques to effectively test an SDN-system's controller and generate a failure-inducing model that characterizes conditions under which the system fails. As shown in Figure 2, FuzzSDN takes as input a test procedure and a failure detection mechanism defined by engineers.

A test procedure consists of three steps: initializing the SDN-system, executing the test scenario, and tearing down the system. In the initialization step, the procedure configures the entities in the SDN-system, such as the controller, switches, and connections, bringing the system to a

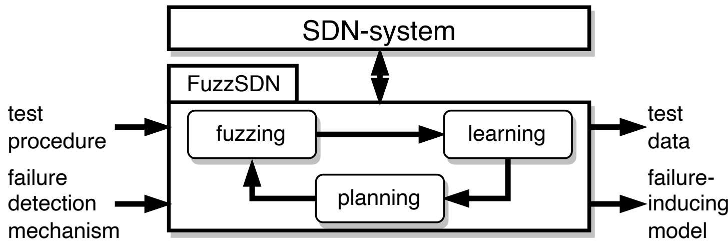

Fig. 2. An overview of our ML-guided Fuzzing method for testing SDN-systems (FuzzSDN).

desired (ready) state for executing the test scenario. For example, during the initialization, the procedure might configure switches with empty or preinstalled forwarding tables, which define how to forward data packets through the SDN. This system configuration allows for testing how the controller instructs the switches in the given setting. The test scenario represents a specific operation (use case) of the SDN-system, involving the exchange of control messages between the controller and switches that FuzzSDN can fuzz. For example, if the test scenario specifies a data transmission from one host to another connected to the SDN, executing the test scenario leads to the exchange of control messages between the controller and switches. These messages are exchanged to discover the locations of the two hosts in the SDN and to instruct the relevant switches with the proper forwarding tables to enable data transmission. In the teardown step, the procedure resets the SDN-system to its default (pristine) state after executing the test scenario, since FuzzSDN requires multiple independent executions of the test procedure.

A failure detection mechanism acts as a test oracle, determining whether the system fails or successfully completes the given test procedure. For example, depending on the test procedure, it detects instances of communication breakdown, controller crashes, or performance degradation. Since an SDN-system provides monitoring tools that enable engineers to oversee system behavior and performance, implementing such a failure detection mechanism is straightforward.

FuzzSDN then outputs test data and a failure-inducing model. The data includes the set of control messages that induced the failure detected by the failure detection mechanism. A failure-inducing model abstracts such test data in the form of conditions and probabilities pertaining to the failure.

As shown in Figure 2, FuzzSDN realizes an iterative process consisting of the following three steps: (1) The fuzzing step sniffs and modifies a control message passing through the control channel from the SDN switches to the controller. The fuzzing step repeats the execution of the input test procedure and modifies only one selected control message for each execution of the system. It then produces a labeled dataset that associates fuzzing outputs (i.e., modified control messages) and their consequences in the system (i.e., system failure or success). (2) The learning step takes as input the labeled dataset created by the fuzzing step and uses a supervised learning technique, e.g., RIPPER [13], to create a failure-inducing (classification) model. This model identifies a set of control messages that cause the failure defined in the failure detection mechanism. Over the iterations of FuzzSDN, such models are used to guide the behavior of the fuzzing step in the next iteration. (3) The planning step instructs the fuzzing step based on the failure-inducing model created by the learning step. Following such instructions, the fuzzing step then efficiently explores the space of control messages to be fuzzed and adds new data points (i.e., modified control messages and their consequences in the system) to the existing labeled dataset. The updated dataset is then used by the learning step to produce an improved failure-inducing model. FuzzSDN stops the iterations of the fuzzing, learning, and planning steps when the accuracy of the output failure-inducing model

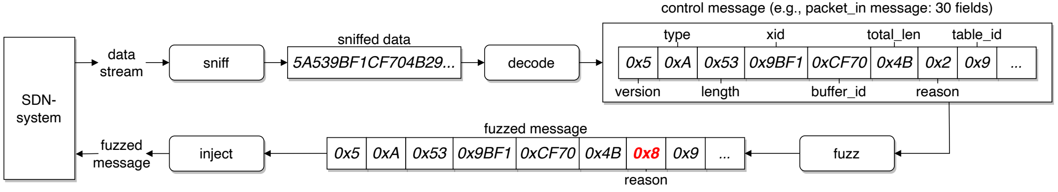

Fig. 3. A data flow example of fuzzing a control message (e.g., packet_in message).

reaches an acceptable level or the execution time of FuzzSDN exceeds an allotted time budget. Below, we explain each step of FuzzSDN in detail.

We assume that engineers using FuzzSDN have expertise in SDN, thus equipping them with the capability to provide required inputs: test procedures and failure detection mechanisms. They should also possess knowledge of the SDN protocol to understand a failure-inducing model that characterizes a set of failure-inducing control messages. Such assumptions are reasonable since engineers need to devise test procedures and failure detection mechanisms when testing their SDN-systems, independently of FuzzSDN, and must know the SDN protocol as it defines the input space of the SDN controller under test.

3.1 Fuzzing step: Initial fuzzing

During the fuzzing step, FuzzSDN manipulates control messages in an SDN-system to cause a system failure. To do so, FuzzSDN utilizes a man-in-the-middle attack, which is a well-known security attack technique in the network domain [14]. The attack technique enables FuzzSDN to intercept control messages transmitting through the control channel and inject modified messages. In addition, FuzzSDN pretends to be both legitimate participants (i.e., SDN switches and controllers) of the control channel. Hence, the system under test is not aware of FuzzSDN while it is running. We omit network-specific details of the attack technique, as they are not part of our contributions; instead, we refer interested readers to the relevant literature [14].

Figure 3 shows a data flow example that illustrates how our fuzzing technique manipulates a control message. As the control channel in the SDN-system transmits a data stream, the fuzzing step first sniffs it as a byte string. It then decodes the sniffed string according to the adopted SDN protocol (e.g., OpenFlow [45]) to identify a control message to be fuzzed. In Figure 3, the byte string 0x5A539BF1CF704B29 is sniffed and is then identified as a packet_in control message encoded in 30 fields according to OpenFlow. Note that a packet_in message is one of the control messages sent by a switch to a controller in order to notify that the switch receives a packet. For details of the packet_in message, we refer readers to the OpenFlow specification [45].

FuzzSDN fuzzes the control message by accounting for the syntax requirements (i.e., grammar) defined in the SDN protocol and then injects the fuzzed message into the control channel in the system. We note that FuzzSDN could apply a simple random fuzzing method that replaces the sniffed string with a random string. However, the majority of byte strings generated by random fuzzing would be invalid control messages that would be immediately rejected by the SDN message parser in the system [29]. FuzzSDN therefore accounts for the SDN protocol in order to generate valid control messages that test software components beyond the message parsing layer of the system, which is a desirable feature in practice [29].

Initial fuzzing. At the first iteration of FuzzSDN, since a failure-inducing model is not present, the fuzzing step behaves as described in Algorithm 1. Given a control message *msg*, the algorithm modifies it and returns a fuzzed message *msg′*. As shown on line 1, the algorithm first randomly

Algorithm 1 Initial fuzzing

Input:
msg: control message to be fuzzed
Output:
msg′: control message after fuzzing

1: $F \leftarrow select_rand_fields(msg)$
2: $msg' \leftarrow msg$
3: **for all** $f \in F$ **do**
4: $msg' \leftarrow replace(msg', f, rand_valid(f))$
5: **end for**
6: **return** msg'

selects a set F of fields in *msg*. For each field f in F, the algorithm replaces the original value of f with a new value randomly selected from its value range (lines 2-5). Hence, Algorithm 1 operates in linear time, relative to the number of message fields ($|F|$). Note that our ML-guided fuzzing method is described in Section 3.4. For example, in Figure 3, the algorithm modifies the sniffed packet_in message by replacing the reason field value 0x2 with 0x8, which is randomly chosen within its value range.

Data collection. To generate a failure-inducing model, FuzzSDN uses a supervised ML technique [61] that requires a labeled dataset the fuzzing step generates. Specifically, at each iteration of FuzzSDN, the fuzzing step is executed n times based on an allotted time budget. Each execution of the fuzzing step (re)runs the input test procedure (Figure 2) and modifies a control message. FuzzSDN then monitors the system response to the modified control message *msg* using the failure detection mechanism (Figure 2). We denote by *presence* (resp. *absence*) the label indicating that the failure is present (resp. absent) in the system response. For each iteration i of FuzzSDN, the fuzzing step creates a labeled dataset D_i by adding n tuples $(msg_1, l_1), \ldots, (msg_n, l_n)$ to D_{i-1}, where a label l_j could be either *presence* or *absence* and $D_0 = \{\}$. We note that FuzzSDN uses an accumulated dataset $D = D_1 \cup \ldots \cup D_i$ to infer a failure-inducing model at each iteration i.

3.2 Learning step

We cast the problem of learning failure-inducing models into a binary classification problem in ML. Given a labeled dataset obtained from the fuzzing step, the learning step infers a prediction model, i.e., a failure-inducing model, that classifies a control message as either the *presence* or *absence* class. A classification result predicts whether or not a control message induces a system failure as captured by the detection mechanism.

FuzzSDN aims at providing engineers not only with failure-inducing control messages but also accurate conditions under which the system fails (as described in Section 1). Hence, we opt to use a learning technique that produces an interpretable model [41]. Engineers could use the prediction results to identify a set of control messages predicted to induce system failures as a test suite for testing the system. An interpretable model would help engineers figure out why the failure occurs in the system. We encode fields (e.g., version, type, and length in Figure 3) of a control message as features in a labeled dataset so that an interpretable ML technique builds a failure-inducing model using fields as features. For example, such a model could explain that the system fails when receiving a control message with an incompatible version number, which is a higher version than the system supports.

FuzzSDN employs RIPPER (Repeated Incremental Pruning to Produce Error Reduction) [13], an interpretable rule-based classification algorithm, to learn a failure-inducing model. We opt to use RIPPER as it generates pruned decision rules that are more concise and thus interpretable than commonly used decision trees (e.g., C4.5 [50]), which are prone to the replicated subtree problem [61]. Further, RIPPER has been successfully applied to many software engineering problems involving classification and rule inference [9, 22, 27]. Given a labeled dataset D, RIPPER produces a set R of decision rules. A decision rule is a simple IF-condition-THEN-prediction statement, consisting of a condition on the fields of a control message (e.g., $version > 5 \wedge length \geq 10$) and a prediction indicating either the *presence* or *absence* class.

For a decision rule, the learning step measures a confidence score [61] to estimate the accuracy of the rule in predicting the actual class of control messages that satisfy the rule's condition. Specifically, given a labeled dataset D and a decision rule r, we denote by t the total number of control messages in D that satisfy the condition of r, i.e., the condition is evaluated to true. Among these t control messages, often, there are some control messages whose labels defined in D do not match r's prediction. We denote by f the number of such control messages. The learning step computes a confidence score $c(r)$ of r by $c(r) = (t - f)/t$. For example, given a rule r: IF $version > 5 \wedge length \geq 10$ THEN $class = presence$, suppose 88 control messages in a labeled dataset D satisfy the condition of r and 7 out of the 88 control messages are labeled with *absence*. Then, the confidence score $c(r)$ is $(88 - 7)/88 = 0.92$. FuzzSDN uses the confidence score $c(r)$ in the planning step to guide our fuzzing strategy.

3.3 Planning step

The planning step guides fuzzing based on a failure-inducing model, i.e., a set R of decision rules inferred from the learning step. Given an original control message to be fuzzed, the main idea is to generate a set of modified control messages using R. To this end, the planning step exploits these decision rules to generate effective control messages that induce failures.

Imbalance handling. Algorithm 2 describes how the planning step uses the set R of decision rules inferred in the current iteration of FuzzSDN to guide the fuzzing step for the next iteration. We note that the planning step accounts for the imbalance problem [61] that usually causes poor performance of ML algorithms. In a labeled dataset, when the number of data instances of one class is much higher than such number for another class, ML classification models have a tendency to predict the majority class. In our study, such models are not practically useful, as engineers are more interested in control messages that cause system failures.

As shown on lines 1-3 of Algorithm 2, the planning step first counts the number *minor* (resp. *major*) of minority (resp. majority) control messages in the given labeled dataset D. Given the number n of control messages to be fuzzed in the next iteration, lines 4-6 of the algorithm then estimate the number $minor'$ (resp. $major'$) of control messages associated with the minority (resp. majority) class to be added to D to create a balanced dataset, containing $|D| + n$ control messages. Specifically, FuzzSDN needs $(|D|+n)/2 - minor$ control messages associated with the minority class to balance D in the next iteration. Table 1 shows examples of *minor*, *major*, $minor'$, and $major'$ at each iteration of FuzzSDN computed by Algorithm 2. For example, at the first iteration of FuzzSDN, when $minor = 10$, $major = 190$, and the number of control messages to be fuzzed $n = 200$, $minor'$ is calculated as $(200 + 200)/2 - 10 = 190$ and $major' = 10$.

Budget distribution. Lines 7-20 of Algorithm 2 describe how the planning step distributes the number n of control messages to be fuzzed in the next iteration to each decision rule $r \in R$. As shown on lines 9-13 of the algorithm, for each rule $r \in R^{minor}$ associated with the minority class, the planning step decides to use r for fuzzing based on its relative confidence score and the number $minor'$ of control messages estimated on line 5. Specifically, the planning step associates r with the

Algorithm 2 Planning

Input:

D: labeled dataset

R: decision rules

n: number of control messages to be fuzzed

Output:

B: budget distribution to the decision rules R

1: //count numbers of majorities and minorities in D
2: $minor \leftarrow get_num_minorities(D)$
3: $major \leftarrow get_num_majorities(D)$
4: //estimate numbers of majorities and minorities after the next iteration
5: $minor' \leftarrow \min((|D| + n)/2 - minor, n)$
6: $major' \leftarrow n - minor'$
7: //assign budgets to minority rules
8: $B \leftarrow \{\}$
9: $R^{minor} \leftarrow get_minority_rules(R)$
10: **for all** $r \in R^{minor}$ **do**
11: $\quad b \leftarrow minor' \times c(r)/sum_c(R^{minor})$
12: $\quad B \leftarrow B \cup (r, b)$
13: **end for**
14: //assign budgets to majority rules
15: $R^{major} \leftarrow get_majority_rules(R)$
16: **for all** $r \in R^{major}$ **do**
17: $\quad b \leftarrow major' \times c(r)/sum_c(R^{major})$
18: $\quad B \leftarrow B \cup (r, b)$
19: **end for**
20: **return** B

number of times r will be applied to fuzz control messages, i.e., $minor' \times c(r)/sum_c(R^{minor})$, where $c(r)$ denotes the rule's confidence score (described in Section 3.2) and $sum_c(R^{minor})$ is defined by $\sum_{r \in R^{minor}} c(r)$, the sum of confidence scores of the rules in R^{minor}. The algorithm therefore weighs the rules according to their confidence scores in order to maximize the chance of correct predictions. When fuzzing a control message guided by a rule r with a high confidence score (e.g., 0.99), the system response to the fuzzed control message would highly likely match the prediction of r. Lines 14-19 describe how the planning step handles rules associated with the majority class, which is the same as on lines 9-13. For example, at the first iteration of FuzzSDN shown in Table 1, let R be $\{r_1, r_2, r_3\}$ where r_1 and r_2 are associated with the minority class (e.g., *presence*) and r_3 with the majority class (e.g., *absence*). Given R, if $c(r_1) = 0.8$, $c(r_2) = 0.7$, and $c(r_3) = 0.8$, then Algorithm 2 distributes $minor' = 190$ (resp. $major' = 10$) to r_1 and r_2 (resp. r_3) as

Table 1. Examples of *minor*, *major*, *minor′*, and *major′* computed by Algorithm 2, when the number n of control messages to be fuzzed is 200.

iteration	$\|D\|$	*minor*	*major*	*minor′*	*major′*
1	200	10	190	190	10
2	400	125	275	175	25
3	600	248	352	152	48
4	800	380	420	120	80
5	1000	495	505	105	95
6	1200	600	600	100	100

follows: $190 \times 0.8/(0.8 + 0.7) = 101$ and $190 \times 0.7/(0.8 + 0.7) = 89$ (resp. $10 \times 0.8/0.8 = 10$). For the second iteration, FuzzSDN then plans to apply r_1 101 times, r_2 89 times, and r_3 10 times to fuzz control messages. Algorithm 2 operates in linear time, relative to the number of rules ($|R|$), inferred by the learning step.

We note that during early iterations of FuzzSDN, the obtained datasets are likely imbalanced because control messages causing system failures are typically difficult to discover via purely random fuzzing, since most fuzzed messages are detected and addressed by the system under test to prevent such failures (see Table 1 and our experiment results in Section 4.6). In addition, due to the small sizes of training datasets in early iterations of FuzzSDN, RIPPER is often not able to produce accurate failure-inducing models. But as FuzzSDN continuously iterates the three steps within an allotted time budget, according to Algorithm 2, training datasets are becoming more balanced and larger (see Table 1), enabling RIPPER to produce increasingly accurate failure-inducing models. Furthermore, given S the space of all possible control messages, once a dataset is balanced, the algorithm enables the fuzz step to explore not only the space P of control messages that likely cause failures but also the remaining space $S \setminus P$ of control messages. Note that RIPPER infers a set of rules' conditions (which define P): $r_1, \ldots, r_k$, that are associated with the minority class and a single rule's condition (which define $S \setminus P$) in the form of $\neg r_1 \land \ldots \land \neg r_k$ for the majority class.
Progress monitoring. To monitor the progress of FuzzSDN, the planning step uses the standard precision and recall metrics [61] (described in Section 4.4). In our context, a high level of precision indicates that the inferred failure-inducing model is able to accurately predict the failure of interest. A failure-inducing model with high recall indicates that most of the control messages actually inducing the failure satisfy the failure-inducing conditions in the model. Hence, a failure-inducing model with a high level of precision and recall is desirable. To compute precision and recall values, FuzzSDN uses the 10-fold cross-validation technique [61]. In 10-fold cross-validation, a dataset D is split into 10 equal-size folds. Nine folds are used as a training dataset and the other one fold is retained as a test dataset. This process is thus repeated 10 times to compute precision and recall values.

3.4 Fuzzing Step: ML-guided Fuzzing

From subsequent iterations of FuzzSDN, the fuzzing step utilizes a set R of decision rules inferred by the learning step according to a budget distribution B computed by the planning step. Using a rule $r \in R$, the fuzzing step modifies a sniffed control message to satisfy the condition of r. Further, the fuzzing step employs a mutation operator to diversify fuzzed control messages beyond those restricted by R. Below we describe the fuzzing step in detail.

Algorithm 3 describes a fuzzing procedure that modifies a sniffed control message *msg* using a budget distribution B. As shown on lines 1-5 of the algorithm, the fuzzing step first chooses a

Algorithm 3 ML-guided Fuzzing

Input:
msg: control message to be fuzzed
B: budget distribution to the decision rules R
mu: mutation rate

Output:
msg': control message after fuzzing
B': budget distribution after fuzzing

```
1:  //fuzz a control message based on a rule
2:  (r, b) ∈ B
3:  F ← get_fields(r, msg)
4:  F' ← solve(r)
5:  msg' ← replace(msg, F, F')
6:  //update the budget distribution
7:  B' ← B \ {(r, b)}
8:  if b − 1 > 0 then
9:      B' ← B ∪ {(r, b − 1)}
10: end if
11: //mutate the fuzzed control message
12: for all f ∈ all_fields(msg) \ F do
13:     if rand(0, 1) ≤ mu then
14:         msg' ← replace(msg', f, rand(f))
15:     end if
16: end for
17: return msg', B'
```

budget assignment $(r, b) \in B$, where r denotes a rule to apply in fuzzing, and b denotes how many times the rule r will be exploited by the fuzzing step. Given the rule r, the algorithm selects a set F of message fields in *msg* that appear in the condition of r (line 3). Using an SMT solver [12], the algorithm solves the condition of r to find a set F' of message fields that satisfy the condition (line 4). Specifically, we use Z3 [15] – a well-known and widely used SMT solver – to solve such conditions. Line 5 of the algorithm then replaces the original fields F with the computed fields F'. For example, when FuzzSDN fuzzes a control message guided by the condition $version > 5 \wedge length \geq 10$, it assigns 6 to the *version* field and 20 to the *length* field of the control message as the assignments satisfy the condition.

Algorithm 3 modifies a single control message *msg* and outputs one fuzzed message msg'. Hence, the fuzzing step executes the algorithm n times to generate n number of fuzzed control messages. As shown on lines 6-10, the algorithm updates the budget distribution B with $(r, b-1)$ indicating that the rule r has been applied once. Note that the fuzzing step reruns the system under test for each execution of the algorithm.

As shown on lines 11-17 of Algorithm 3, the fuzzing step leverages a mutation technique to diversify fuzzed control messages. Recall that lines 1-5 of the algorithm modify only the message fields that appear in decision rules. Without mutation, decision rules inferred in the first iteration of FuzzSDN would determine the message fields being modified in all subsequent iterations, while other message fields would remain unchanged. Such a fuzzing method might miss important failure-inducing rules related to other unchanged fields.

The fuzzing step employs a uniform mutation operator [59] that randomly selects fields in a control message with a mutation rate *mu* and changes the fields' values to random values within their ranges. As shown on line 12 of Algorithm 3, the fuzzing step selects the fields in the sniffed control message *msg* that are not present in the exploited rule r. Hence, the return message msg' (line 14) also satisfies the condition of r, as mutated fields cannot affect it. For example, suppose a packet_in message encoded in 30 fields is sniffed to be fuzzed by FuzzSDN, and its version and length fields are included in the condition $version > 5 \wedge length \geq 10$ and hence fuzzed (lines 1-10). In this setting, FuzzSDN randomly selects fields (e.g., *reason* and *table_id*) that are not present in the condition, and then mutates the selected fields by assigning new random values within their ranges (lines 11-16).

We note that the computational complexity of Algorithm 3 is primarily determined by line 4, which uses Z3. The remaining computations scale linearly with the number of message fields. In our context, as described in Section 3.2, the rule r to be solved by Z3 is concise. Therefore, Algorithm 3 is expected to run in practical time, as empirically evaluated in our experiments (Section 4.6).

4 EVALUATION

In this section, we present our empirical evaluation of FuzzSDN. Our full evaluation package is available online [44].

4.1 Research Questions

RQ1 (comparison): *How does FuzzSDN perform compared with state-of-the-art testing techniques for SDNs?* We investigate whether FuzzSDN can outperform existing techniques: DELTA [35] and BEADS [29] described in Section 4.4. We choose these techniques as they rely on fuzzing to test SDN-systems and their implementations are available online. Note that none of the prior methods that identify failure-inducing inputs [24, 30] account for the specificities of SDNs; hence, they are not applicable.

RQ2 (usefulness): *Can FuzzSDN learn failure-inducing models that accurately characterize conditions under which a system fails?* We investigate whether or not FuzzSDN can infer accurate failure-inducing models. In addition, we compare the failure-inducing conditions identified by FuzzSDN with those reported in the literature [29] to assess if these conditions are consistent with analyses from experts.

RQ3 (scalability): *Can FuzzSDN fuzz control messages and learn failure-inducing models in practical time?* We analyze the relationship between the execution time of FuzzSDN and network size. To do so, we conduct experiments with systems of various network sizes.

4.2 Simulation Platform

To evaluate FuzzSDN, we opt to use a simulation platform that emulates physical networks. Specifically, we use Mininet [31] to create virtual networks of different sizes. In addition, as Mininet employs real SDN switch programs, the emulated networks are very close to real-world SDNs. Hence, Mininet has been widely used in many SDN studies [29, 35, 55]. We note that FuzzSDN can be applied to test actual SDN-systems. However, using physical networks is prohibitively expensive for performing the types of large experiments involved in our systematic evaluations of FuzzSDN. We ran all our experiments on 10 virtual machines, each of which with 4 CPUs and 10GB of memory. These experiments took ≈45 days by concurrently running them on the 10 virtual machines.

4.3 Study subjects

We evaluate FuzzSDN by testing two actual SDN controllers, i.e., ONOS [4] and RYU [54], which are still maintained actively and have been widely used in both research and practice [29, 34–36, 56, 64].

Both controllers are implemented using the OpenFlow SDN protocol specification [45]. Since FuzzSDN fuzzes OpenFlow control messages, it can test any SDN controller that implements the OpenFlow specification. Regarding virtual networks, we synthesize five networks with 1, 3, 5, 7, and 9 switches controlled by either ONOS or RYU. In each network, all switches are interconnected with one another, i.e., fully connected topology. Each switch is connected to two hosts that can emulate any device that sends and receives data streams, e.g., video and sound streams. We note that our study subjects, i.e., 5×2 SDN-systems built on the five networks controlled by ONOS and RYU, are representative of existing SDN studies and real-world SDNs. For example, DELTA [35] (resp. BEADS [29]) was evaluated with an SDN-system including two (resp. three) switches controlled by ONOS and RYU, as running experiments with SDNs requires large computational resources [3]. Shin et al. [55] introduced an industrial SDN-system developed in collaboration with SES, a satellite operator, which contains seven switches controlled by ONOS.

4.4 Experimental setup

EXP1. To answer RQ1, we compare FuzzSDN with DELTA [35] and BEADS [29], which are applicable to our study subjects. DELTA is a security assessment framework for SDNs that enables engineers to automatically reproduce known SDN-related attack scenarios and discover new attack scenarios. For the latter, DELTA relies on random fuzzing that randomizes all fields of a control message without accounting for the specifics of the OpenFlow protocol. BEADS is an automated attack discovery technique based on fuzzing that assumes the OpenFlow protocol, aiming at generating fuzzed control messages that can pass beyond the message parsing layer of the system under test. We note that we reused the implementations available online. However, we had to adapt them in order to make them work in our experiments, though we minimized changes, since the original executables of DELTA and BEADS did not work even after discussions with the authors.

In EXP1, we use two synthetic systems with one switch controlled by either ONOS or RYU. For the test procedure (see Section 3) in EXP1, we use the pairwise ping test [8], applied in many SDN studies [16, 29, 35, 39], that allows us to detect whether or not hosts can communicate with one another. Regarding the failure detection mechanism (see Section 3) in EXP1, we consider switch disconnections as system failures since both DELTA and BEADS analyzed switch disconnections in their experiments. We further note that EXP1 identifies switch disconnections as failures only when they lead to a communication breakdown and the system fails to locate the causes of the failures, i.e., no relevant log messages related to the failures. EXP1 fuzzes the packet_in message [45], which has 57 bytes encoded in 30 fields, as SDN switches send this message to the controller in the execution of the test procedure, and both DELTA and BEADS fuzz it. For details of OpenFlow messages, we refer readers to the OpenFlow specification [45]. We compare the number of fuzzed control messages that cause the switch disconnection failure across fuzzing approaches.

EXP2. To answer RQ2, we evaluate the accuracy of failure-inducing models inferred by FuzzSDN. To this end, we compare the models obtained by FuzzSDN with those produced by our baselines extending DELTA and BEADS. In addition, we examine our failure-inducing models in light of the literature [29, 35] discussing failure-inducing conditions.

EXP2.1. As baselines, we extend DELTA and BEADS, named DELTA^L and BEADS^L, to produce failure-inducing models. DELTA^L (resp. BEADS^L) encodes the fuzzing results obtained by DELTA (resp. BEADS^L) as a training dataset (see the dataset format described in Section 3.1) and uses RIPPER to learn a failure-inducing model from it. Unlike FuzzSDN, DELTA^L and BEADS^L do not leverage the inferred failure-inducing models to guide their fuzzing.

For ensuring fair comparisons of FuzzSDN, DELTA^L, and BEADS^L, EXP2.1 creates a test dataset containing 5000 fuzzing results for each method. EXP2.1 then measures the accuracy of the failure-inducing models obtained by the three methods using the standard precision and recall metrics [61].

Additionally, EXP2.1 measures imbalance ratios of the datasets obtained at each iteration of the three methods using the imbalance ratio metric [61].

We compute precision and recall values as follows: (1) precision $P = TP/(TP+FP)$ and (2) recall $R = TP/(TP+FN)$, where TP, FP, and FN denote the number of true positives, false positives, and false negatives, respectively. A true positive is a control message labeled with *presence* (see Section 3.1) and correctly classified as such. A false positive is a control message labeled with *absence* (see Section 3.1) but incorrectly classified as *presence*. A false negative is a control message labeled with *presence* but incorrectly classified as *absence*. We compute the imbalance ratio of a dataset as follows: imbalance ratio $I = 1 - (minor/major)$, where *minor* and *major* denote the number of control messages in the dataset D, labeled with the minority and majority class, respectively. In EXP2.1, we use two synthetic systems with one switch controlled by either ONOS or RYU. EXP2.1 applies the pairwise ping test and fuzzes 8000 packet_in messages with FuzzSDN, DELTA^L, and BEADS^L.

EXP2.2. Jero et al. [29] manually inspected their SDN testing results obtained with BEADS and identified some conditions on message fields that led to system failures. EXP2.2 examines our failure-inducing models to assess the extent to which our models are consistent with their manual analysis results.

For EXP2.2, we use two synthetic systems with one switch controlled by either ONOS or RYU. EXP2.2 fuzzes the following five types of control messages: packet_in (57 bytes, 30 fields), hello (8 bytes, 4 fields), barrier_reply (8 bytes, 4 fields), barrier_request (8 bytes, 4 fields), and flow_removed (55 bytes, 22 fields), which are manipulated by BEADS. We randomly selected these five message types from the 16 types of control messages analyzed in the prior study of BEADS, to keep the expected cost of running our experiments manageable (see Section 4.5). In EXP2.2, FuzzSDN fuzzes 8000 control messages of each type. To fuzz the barrier_request and the barrier_reply control messages, we use a test procedure that connects switches to an SDN controller as it generates the control messages. For the remaining types of control messages, we use the pairwise ping test [8]. Regarding failure types, EXP2.2 detects unexpected broadcasts from switches and unexpected switch disconnections as failures. These are studied in existing work (BEADS). The broadcasting mechanism of ARP (Address Resolution Protocol) [47] is used to discover host locations in the SDN. If broadcasting occurs unexpectedly, it may lead to the installation of incorrect forwarding rules on the switches in the SDN, possibly resulting in information leakage (since data can be forwarded to unintended hosts) or connectivity losses. Regarding the other types of failure, if the communication between the controller and the switches is unexpectedly disconnected, the SDN-system obviously cannot operate as intended. This is because the controller monitors the network status based on the messages received from the switches and the behavior of the switches is directed by the controller.

EXP3. To answer RQ3, we study the correlation between the execution time of FuzzSDN and the 2 $\times$ 5 synthetic systems (described in Section 4.3) with 1, 3, 5, 7, and 9 switches controlled by either ONOS or RYU, respectively. We measure the execution time of each iteration of FuzzSDN and the execution time of configuring Mininet and the SDN controllers. Our conjecture is that the execution time of FuzzSDN does not depend on network sizes. However, we also conjecture that there is a correlation between the configuration time of Mininet and SDN controllers and network sizes. Such configuration time includes the time for initializing controllers and Mininet, creating virtual networks, and activating controllers. EXP3 uses the pairwise ping test.

4.5 Parameter Tuning

Recall from Section 3 that FuzzSDN must be configured with the following parameters: number of control messages to be fuzzed at each iteration, mutation rate, and RIPPER parameters. For tuning the parameters, we ran initial experiments relying on hyperparameter optimization [61] based on

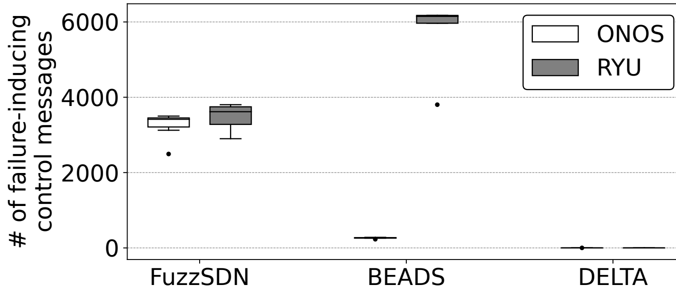

Fig. 4. Comparing FuzzSDN, BEADS, and DELTA based on the number of fuzzed control messages that cause the switch disconnection failure. The boxplots (25%-50%-75%) show distributions of the numbers of failure-inducing control messages obtained from 10 runs of EXP1, testing either ONOS or RYU.

guidelines in the literature [28, 61]. In our initial experiments, we assessed 10 configurations of the parameters' values to select the best one to be used in further experiments. We selected these 10 configurations using a grid search [61]. To select the best configuration, for each configuration, we ran FuzzSDN for four days to ensure there were no notable changes in the results and measured the precision and recall values of the obtained failure-inducing model. For our experiments, we set the number of control messages to be fuzzed at each iteration to 200 and the mutation rate to $1/|F|$, where $|F|$ denotes the number of fields in a control message to be fuzzed. The parameter values of RIPPER used in our experiments can be found in our repository [44].

To fairly compare FuzzSDN and the other approaches (i.e., DELTA, BEADS, DELTA^L, BEADS^L), we assign to them the same computation budget: four days for ONOS and two days for RYU. Within this budget, FuzzSDN generates a balanced dataset (described in Section 3.2), and precision and recall values of the inferred failure-inducing models reach their plateaus. We note that the configuration time of RYU to run FuzzSDN is approximately half that of ONOS. Hence, we set different budgets so that FuzzSDN fuzzes similar numbers of control messages for ONOS and RYU. Since FuzzSDN is randomized, we repeat our experiments 10 times.

The parameters of FuzzSDN and our experiments can certainly be further tuned to improve the accuracy of FuzzSDN. However, we were able to convincingly and clearly support our conclusions with the selected configuration, using the study subjects (described in Section 4.3). Hence, this article does not report further experiments on optimizing those parameters.

4.6 Experiment Results

RQ1. Figure 4 compares FuzzSDN, BEADS and DELTA when testing the two study subjects controlled by either ONOS or RYU (EXP1). The boxplots depict distributions (25%-50%-75% quantiles) of the numbers of control messages that cause the switch disconnection failure. The results shown in the figure are obtained from 10 runs of EXP1. To fairly compare FuzzSDN, BEADS, and DELTA, they were assigned the same computation budget: four days for ONOS and two days for RYU (as described in Section 4.5).

As shown in Figure 4, for ONOS, FuzzSDN generates significantly more failure-inducing control messages than BEADS and DELTA. On average, 3425 failure-inducing control messages are generated by FuzzSDN, in contrast to 270 by BEADS and 3 by DELTA. Most of the control messages manipulated by DELTA are filtered out by the message parsing layers of the controllers, which is consistent with the finding reported in the BEADS study [29]. Regarding the application of BEADS to RYU, the situation is apparently more complicated. RYU is in fact much less robust than ONOS

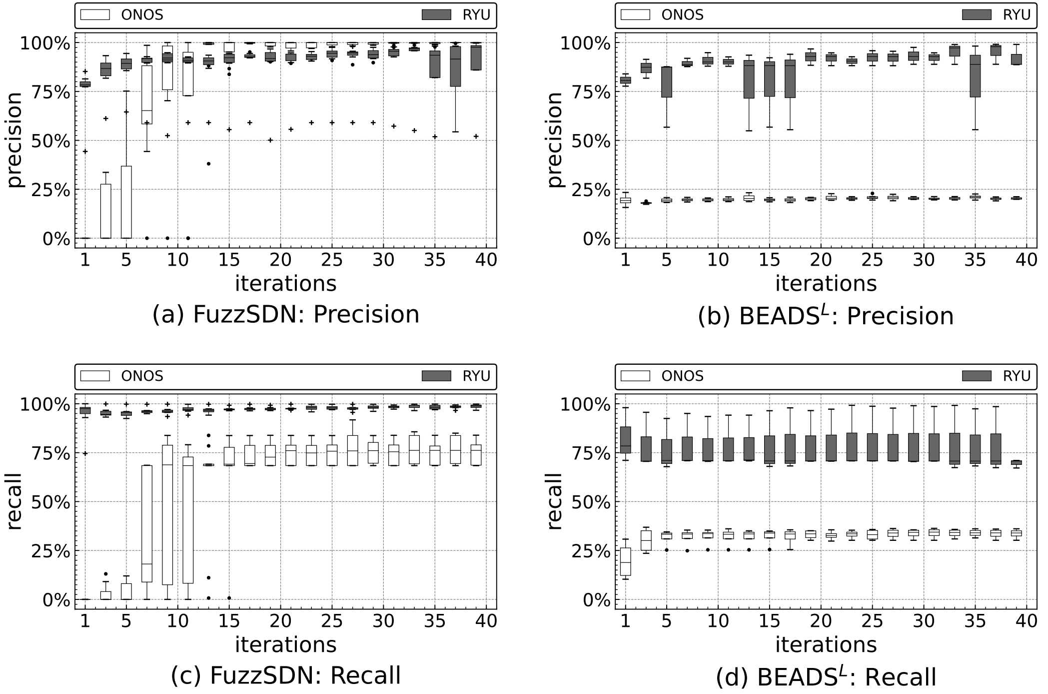

Fig. 5. Comparing distributions of precision and recall values obtained from FuzzSDN and BEADS^L that test the systems controlled by either ONOS or RYU (see EXP2.1). The boxplots (25%-50%-75%) show distributions of precision (a, b) and recall (c, d) values obtained from 10 runs of EXP2.1.

when handling fuzzed control messages. It is therefore easy to fuzz messages leading to failures with RYU. But recall that FuzzSDN aims at generating a balanced labeled dataset containing control messages that are associated with both the *presence* and *absence* of failures in similar proportions (Section 3.2). This is not the case of BEADS which then generates a very large proportion of failure-inducing control messages with RYU, more than that observed with FuzzSDN.

The answer to **RQ1** *is that* FuzzSDN significantly outperforms BEADS and DELTA. In particular, our experiments show that FuzzSDN is able to generate a much larger number of control messages that cause failures when the SDN controller is relatively robust to fuzzed messages (e.g., ONOS). Such robustness is a desirable and common feature in industrial SDN controllers.

RQ2. Figure 5 compares precision (a, b) and recall (c, d) values obtained from FuzzSDN and BEADS^L for the study subjects controlled by either ONOS or RYU and the test dataset (EXP2.1). The boxplots in Figures 5(a) and 5(b) (resp. 5(c) and 5(d)) show distributions (25%-50%-75% quantiles) of precision (resp. recall) values over 40 iterations of the methods obtained from 10 runs of EXP2.1. Note that each iteration of BEADS^L adds 200 fuzzed control messages (the same as FuzzSDN) to a dataset and learns a failure-inducing model. In contrast to FuzzSDN, BEADS^L does not use the failure-inducing model to guide fuzzing. We omit the results obtained by DELTA^L because the labeled dataset it created contains only a few failure-inducing control messages, as reported in RQ1.

As shown in Figure 5, the failure-inducing models obtained by FuzzSDN yield higher precision and recall than those obtained by BEADS^L over 40 iterations. The results show that, after 20

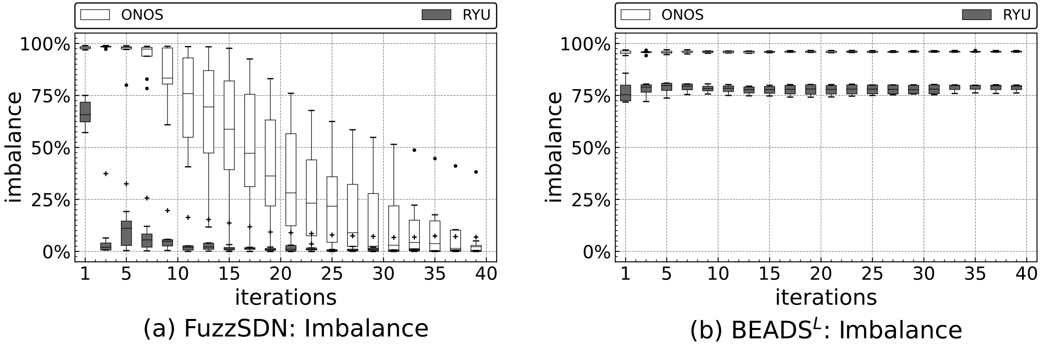

Fig. 6. Comparing distributions of imbalance ratios obtained from FuzzSDN and BEADS^L that test the synthetic systems controlled by either ONOS or RYU (see EXP2.1). The boxplots (25%-50%-75%) show distributions of imbalance ratios obtained from 10 runs of EXP2.1.

iterations, there are no notable changes in precision and recall values. Specifically, for ONOS (resp. RYU), FuzzSDN achieves, on average, a precision of 99.8% (resp. 95.4%) and a recall of 75.5% (resp. 96.7%) after 20 iterations. In contrast, BEADS^L achieves, for ONOS (resp. RYU), on average, a precision of 20.9% (resp. 90.5%) and a recall of 29.9% (resp. 70.7%) after 20 iterations. The 20 iterations of FuzzSDN took, on average, 2.33 days for ONOS and 1.10 days for RYU.

Figure 6 shows the comparison of imbalance ratios for datasets obtained from FuzzSDN and BEADS^L that test the study subjects controlled by either ONOS or RYU (EXP2.1). The boxplots depict distributions of imbalance ratios over 40 iterations of the methods, compiled from 10 runs of EXP2.1. Note that the lower the imbalance ratio, the more balanced the dataset. In Figure 6, we omitted the results obtained by DELTA^L as they provide no additional findings, which are discussed below.

As shown in Figure 6, the datasets generated by FuzzSDN are significantly more balanced than those generated by BEADS^L over 40 iterations. Specifically, after 40 iterations, FuzzSDN achieves, on average, an imbalance ratio of 5.47% for ONOS and 1.38% for RYU. In contrast, BEADS^L achieves, on average, an imbalance ratio of 96.2% for ONOS and 78.9% for RYU. DELTA^L produces datasets that are even more imbalanced than the other two methods, with, on average, an imbalance ratio of 99.9% for both ONOS and RYU over 40 iterations. The results thus indicate that FuzzSDN effectively addresses the imbalance problem over iterations, leading to accurate characterization of failure-inducing models (see the precision and recall results in Figure 5). However, BEADS^L and DELTA^L, which do not tackle the imbalance problem, produce highly imbalanced datasets across all iterations, leading to lower precision and recall of failure-inducing models (see Figure 5).

Table 2 presents the summary of our experiment results obtained from EXP2.2. In the experiments, recall that FuzzSDN fuzzes the following five types of control messages: packet_in, hello, flow_removed, barrier_request, and barrier_reply. For each control message type, the table shows the message size, the number of rules generated by FuzzSDN, the number of fields in the inferred rules, and the number of fields that appear in failure-inducing strategies reported in a prior study [29]. Such strategies were manually defined by analysts, e.g., the switch disconnection failure can occur when changing the version, type, and length fields in a packet_in message. The “all included?” column in the table indicates whether or not the fields in the existing failure-inducing strategies appear in the failure-inducing model obtained by FuzzSDN.

Table 2. Summary of the EXP2.2 results. Five types of control messages are fuzzed for each experiment with the system controlled by ONOS.

message type	message size	# rules (FuzzSDN)	# fields (FuzzSDN)	# fields (manual)	all included?
packet_in	57b,30f	32	12	3	yes
hello	8b,4f	21	4	3	yes
flow_removed	55b,22f	12	4	3	yes
barrier_request	8b,4f	8	4	3	yes
barrier_reply	8b,4f	3	3	3	yes

b: bytes, f: fields

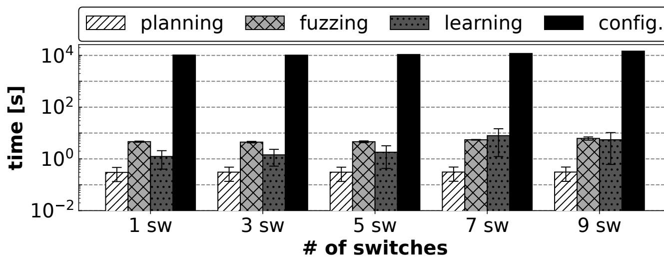

Fig. 7. Comparing execution times of FuzzSDN when varying network sizes as follows: 1, 3, 5, 7, and 9 switches (see EXP3). The bars show the mean execution times of each step of FuzzSDN and the mean configuration times of ONOS and Mininet computed based on 40 iterations of FuzzSDN. The vertical lines on the bars show the standard errors of the mean values.

From Table 2, we see that, for each failure case, all fields in the corresponding failure-inducing strategy described in existing work appear in the failure-inducing models obtained by FuzzSDN. Hence, the results show that FuzzSDN does not miss any important field related to system failures. However, FuzzSDN discovers more fields relevant to failures than the ones reported in prior work. For example, FuzzSDN found 32 rules with 12 fields for the packet_in experiment. Nevertheless, we believe that inspecting those 32 precise rules is considerably more efficient for understanding failure-inducing conditions than manually inspecting 8000 fuzzed control messages. Regarding our results for RYU, we refer the reader to our repository [44] since our findings are similar to those of ONOS.

> *The answer to* **RQ2** *is that* FuzzSDN generates accurate failure-inducing models in practical time, thus faithfully characterizing failure conditions. In addition, the failure analysis results reported in the literature are consistent with the failure-inducing models produced by FuzzSDN.

RQ3. Figure 7 shows the mean execution times of the planning, fuzzing, and learning steps of FuzzSDN and the mean configuration times of Mininet and ONOS for each iteration of FuzzSDN. The figure presents the results obtained from EXP3 using five study subjects with 1, 3, 5, 7, and 9 switches controlled by ONOS. Note that the y-axis of the figure is a 10-base log scale in seconds.

As shown in Figure 7, the execution times of the three FuzzSDN steps do not depend on the network sizes. For each FuzzSDN iteration that generates, on average, 200 fuzzed control messages,

the planning step takes 300ms, the fuzzing step 5.1s, and the learning step 3.5s, which align with our expectations for executing them in practical time. However, 200 configurations of Mininet and ONOS at each iteration takes, on average, 2.85h with 1 switch, 3.00h with 3 switches, 3.02h with 5 switches, 3.31h with 7 switches, and 4.07h with 9 switches, which dominate the overall testing times. Recall that the configuration time of RYU takes approximately half that of ONOS. Even though the configuration times of an SDN controller and Mininet are not in the scope of our study, the results indicate a technical bottleneck to be further investigated to shorten testing time. Regarding the memory requirement, our experiments consumed, at most, 1.2GB of memory, including FuzzSDN, an SDN controller, and the simulation platform. In particular, ONOS used ≈1GB of memory and Mininet ≈15MB of memory per switch. Hence, FuzzSDN does not add significant overhead to the current simulation practice for SDNs.

The answer to **RQ3** *is that* the execution time of FuzzSDN has no correlation with the size of network. Hence, FuzzSDN is applicable to complex systems with large networks.

4.7 Threats to Validity

Internal validity. To mitigate potential internal threats to validity, our experiments compared FuzzSDN with two state-of-the-art solutions (DELTA and BEADS) that generate failure-inducing control messages for testing SDN controllers. Though DELTA and BEADS were our best options, they do not generate failure-inducing models. Therefore, we extended them to produce failure-inducing models and support the comparison of FuzzSDN with these baselines.

External validity. The main concern regarding external validity is the possibility that our results may not generalize to different contexts. In our experiments, we applied FuzzSDN to several SDNs and two actively maintained SDN controllers, i.e., ONOS and RYU. We ensured that our synthetic SDN-systems, consisting of five networks with 1, 3, 5, 7, and 9 switches, controlled by either ONOS or RYU, are representative of existing SDN studies and real-world SDN-systems (e.g., emergency management systems [55]). Further, the SDN-systems used in our experiments are more complex and larger than those previously used to assess DELTA and BEADS, which contain at most three switches. In general, the performance of SDN fuzzing techniques, such as FuzzSDN, DELTA, and BEADS, is not correlated with the size of SDN, as these techniques sniff and modify a control message passing through the control channel between the SDN switches and the controller. Specifically, sniffing a network interface and fuzzing a network packet that passes through the interface does not depend on the size of an SDN-system. To evaluate FuzzSDN in a realistic setting, we used actual SDN controllers and switch software while emulating only the physical networks, including links, host devices, and switch devices.

The prototype implementation of FuzzSDN supports OpenFlow, which is a de facto standard SDN protocol used in many SDN-systems [29, 34–36, 39, 56]. As a result, we were able to apply FuzzSDN to actual SDN controllers (i.e., ONOS and RYU) and compare it with existing tools (i.e., DELTA and BEADS), given their support for OpenFlow. To apply FuzzSDN to SDN-systems that employ other SDN protocols, such as Cisco OpFlex [57] and ForCES [26], one must adapt the sniffing and injecting steps of FuzzSDN (see Figure 3) to correctly decode and encode control messages, respectively. However, this adaptation does not affect the fuzzing, learning, and planning steps of FuzzSDN. We therefore expect that, while the adaptation requires engineering effort to update the sniffing and injecting steps, it does not impact FuzzSDN's performance.

FuzzSDN is developed to be generally applicable to any SDN-system. Our evaluation package and the FuzzSDN tool are available online [44] to (1) facilitate reproducibility of our experiments and (2) enable researchers and practitioners to use and adapt FuzzSDN. Nevertheless, further case studies in other contexts, including industry SDN-systems that employ different SDN protocols, as well as

user studies involving practitioners, remain necessary to further investigate the generalizability of our results.

Limitations. FuzzSDN requires users to provide a test procedure (e.g., pairwise ping test) and select a control message (e.g., packet_in) to be fuzzed. These design choices enable FuzzSDN to generate failure-inducing messages and models within reasonable time budgets (e.g., 4 days for ONOS and 2 days for RYU). Automatically exploring possible use scenarios (i.e., test procedures) and sequences of messages, while accounting for state changes in the SDN controller, can help engineers reduce their manual efforts in testing (e.g., providing test procedures and selecting a message to be fuzzed). However, efficiently and effectively exploring such spaces is a hard problem that requires further research. Given its promising results, we believe that FuzzSDN is nevertheless a practical solution and serves as a solid foundation for researchers and practitioners to further enhance automation in testing SDN controllers.

5 RELATED WORK

In this section, we discuss related work in the areas of SDN testing, fuzzing, and characterizing failure-inducing inputs.

SDN testing. Testing SDNs has been primarily studied in the networking literature targeting various objectives, such as detecting security vulnerabilities and attacks [2, 5, 11, 29, 35, 43, 64], identifying inconsistencies among the SDN components (i.e., applications, controllers, and switches) [34, 36, 56], and analyzing SDN executions [19, 39, 58]. Among these, we discuss SDN testing techniques that rely on fuzzing, as they constitute the most closely related research. Woo et al. [62] proposed RE-CHECKER to fuzz RESTful services provided by SDN controllers. RE-CHECKER fuzzes an input file, encoded in JSON format, that a network administrator uses to specify network policies (e.g., data forwarding rules). This results in generating numerous malformed REST messages for testing RESTful services in SDN. Dixit et al. [17] presented AIM-SDN to test the implementation of the network management datastore architecture (NMDA) [6] in SDN. AIM-SDN randomly fuzzes REST messages to test the NMDA implementation in SDN with regard to the availability, integrity, and confidentiality of datastores. Shukla et al. [56] developed PAZZ that aims at detecting faults in SDN switches by fuzzing data packet headers, e.g., IPv4 and IPv6 headers. Albab et al. [1] presented SwitchV to validate the behaviors of SDN switches. SwitchV uses fuzzing and symbolic execution to analyze the p4 [7] models that specify the behaviors of SDN switches. Lee et al. [33, 34] introduced AudiSDN that employs fuzzing to detect policy inconsistencies among SDN components (i.e., controllers and switches). AudiSDN fuzzes network policies submitted by administrators through the REST APIs. To increase the likelihood of discovering inconsistencies, AudiSDN employs rule dependency trees derived from the OpenFlow specification, which restrict valid relationships among rule elements. In contrast to these existing methods, FuzzSDN fuzzes SDN control messages to test SDN controllers (as DELTA and BEADS do). Furthermore, FuzzSDN uses ML to guide fuzzing and learn failure-inducing models.

Fuzzing. To efficiently generate effective test data, fuzzing has been widely applied in many application domains [38]. The research strands that most closely relate to our work are fuzzing techniques based on ML for testing networked systems [10, 65]. Chen et al. [10] proposed a fuzzing technique for testing cyber-physical systems, which contain sensors and actuators distributed over a network. The proposed approach relies on a deep learning technique to fuzz actuators' commands that can drive the CPS under test into unsafe physical states. Zhao et al. [65] developed SeqFuzzer that enables engineers to test communication systems without prior knowledge of the systems' communication protocols. SeqFuzzer infers communication protocols using a deep learning technique and generates test data based on the inferred protocols. Unlike these prior research threads, FuzzSDN applies ML and fuzzing in the context of testing SDN-systems. In addition,

FuzzSDN employs an interpretable ML technique to provide engineers with comprehensible failure-inducing models.

Besides the fuzzing techniques mentioned above that leverage ML for testing networked systems, coverage-aware fuzzing techniques have been used in many studies across different domains [21, 46, 63]. For example, AFL [63] is a mutational, coverage-guided fuzzer that uses compile-time instrumentation and genetic algorithms to automatically generate test cases that uncover previously unexplored internal states in the program under test. AFL++ [21] is a fuzzing framework that expands on AFL, incorporating many state-of-the-art techniques that enhance fuzzing performance, thus enabling researchers to evaluate different combinations of such techniques. However, applying such fuzzing tools, developed in other contexts, to test SDN controllers is far from being straightforward as there are differences in inputs, outputs, and states. For example, the notion of coverage, e.g., statements and branches, used in coverage-aware fuzzing tools is suitable for testing stateless programs, where outputs depend solely on inputs and do not depend on any memory of past interactions. However, an SDN controller is a stateful program. It takes as input sequences of control messages from the switches in the network, processes these messages, and outputs appropriate sequences of responses. Note that the controller's response is determined by the currently received message and its internal state, which is determined by previously processed messages. AFLNet [46], which extends AFL to test servers, is proposed to address this issue by utilizing state coverage rooted in finite state machines. In AFLNet, finite state machines are constructed using the response codes (also known as status codes) from network protocols, which indicate the result of a client's request to a server. However, AFLNet is not applicable when such response codes are not available, as in our context. Hence, we need a different notion of coverage when testing SDN controllers. In addition to the coverage issue, existing techniques that enhance fuzzing performance for single programs are not easily applicable to testing SDN controllers. For example, the forkserver technique implemented in AFL++, which uses the fork mechanism to reduce the high cost of initialization, is not applicable to our context. Since an SDN controller interacts with multiple switches connected to various hosts, testing the controller requires the costly initialization of not only the controller but also the other components in the SDN-system. Further, testing the controller impacts the states of these other components, making it difficult to efficiently reset and maintain a consistent testing environment. While coverage-aware fuzzing tools, such as AFL++, provide significant advances in testing stateless programs, applying them to test SDN controllers therefore raises difficult challenges.

Characterizing failure-inducing inputs. Recently, a few research strands aimed at characterizing input conditions under which a system under test fails [24, 30]. Gopinath et al. [24] introduced DDTEST that abstracts failure-inducing inputs. DDTEST aims at testing software programs, e.g., JavaScript translators and command-line utilities, that take as input strings. For abstracting failure-inducing inputs, DDTEST uses a derivation tree that depicts how failure-inducing strings can be derived. Kampmann et al. [30] presented ALHAZEN that learns circumstances under which the software program under test fails. ALHAZEN also targets software programs that process strings. ALHAZEN relies on ML to learn failure-inducing circumstances in the form of decision trees. Compared to our work, we note that the context of these research threads is significantly different from our application context: SDNs. Hence, they are not applicable to generate test data and learn failure-inducing models for SDN controllers. To our knowledge, FuzzSDN is the first attempt that applies ML for guiding fuzzing and learning failure-inducing models by accounting for the specificities of SDNs.

6 CONCLUSIONS

We developed FuzzSDN, an ML-guided fuzzing method for testing SDN-systems. FuzzSDN employs ML to guide fuzzing in order to (1) generate a set of test data, i.e., fuzzed control messages, leading to failures and (2) learn failure-inducing models that describe conditions when the system is likely to fail. FuzzSDN implements an iterative process that fuzzes control messages, learns failure-inducing models, and plans how to better guide fuzzing in the next iteration based on the learned models. We evaluated FuzzSDN on several synthetic SDN-systems controlled by either one of two SDN controllers. Our results indicate that FuzzSDN is able to generate effective test data that cause system failures and produce accurate failure-inducing models. Furthermore, FuzzSDN's performance does not depend on the network size and is hence applicable to systems with large networks.

In the future, we plan to extend FuzzSDN to account for sequences of control messages. Fuzzing multiple control messages at once poses new challenges due to message intervals, message dependencies, and state changes in the system under test. To learn failure-inducing models from control message sequences, we will further investigate ML techniques that process sequential data. In addition, we will extend FuzzSDN to generate diverse test cases, defined by message sequences, aiming at maximizing state coverage during testing of SDN controllers. To this end, we will conduct further research to define and quantify the diversity of test cases and the state coverage of SDN controllers, and incorporate these into our fuzzing framework. In the long term, we plan to further validate the generalizability and usefulness of FuzzSDN by applying it to additional SDN-systems and conducting user studies.

DATA AVAILABILITY

Our evaluation package and the FuzzSDN tool are available online [44] to (1) increase the reproducibility of our experiments and (2) enable researchers and practitioners to use and adapt FuzzSDN.

ACKNOWLEDGMENTS

This project has received funding from SES, the Luxembourg National Research Fund under the Industrial Partnership Block Grant (IPBG), ref. IPBG19/14016225/INSTRUCT, the Science Foundation Ireland grant 13/RC/2094-2, and NSERC of Canada under the Discovery and CRC programs.

REFERENCES

[1] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: automated SDN switch validation with P4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 365–379.

[2] Abdullah M. Alshanqiti, Safi Faizullah, Sarwan Ali, Maria Khalid Alvi, Muhammad Asad Khan, and Imdadullah Khan. 2019. Detecting DDoS Attack on SDN Due to Vulnerabilities in OpenFlow. In *Proceedings of the 2019 International Conference on Advances in the Emerging Computing Technologies*. 1–6.

[3] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. 2018. Distributed SDN Control: Survey, Taxonomy, and Challenges. *IEEE Communications Surveys & Tutorials* 20 (2018), 333–354.

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the 3rd Workshop on Hot topics in Software Defined Networking*. 1–6.

[5] Suman Sankar Bhunia and Mohan Gurusamy. 2017. Dynamic attack detection and mitigation in IoT using SDN. In *Proceedings of the 27th International Telecommunication Networks and Applications Conference*. 1–6.

[6] Martin Björklund, Jürgen Schönwälder, Philip A. Shafer, Kent Watsen, and Robert Wilton. 2018. Network Management Datastore Architecture (NMDA). RFC 8342.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Computer Communication Review* 44, 3 (2014), 87–95.

[8] Robert T. Braden. 1989. *Requirements for Internet Hosts - Communication Layers.* Information RFC 1122. Internet Engineering Task Force (IETF).
[9] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. 2020. Planning for untangling: Predicting the difficulty of merge conflicts. In *Proceedings of the 42nd International Conference on Software Engineering*. 801–811.
[10] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 962–973.
[11] Juan Camilo Correa Chica, Jenny Cuatindioy Imbachi, and Juan Felipe Botero. 2020. Security in SDN: A comprehensive survey. *Journal of Network and Computer Applications* 159 (2020), 1–23.
[12] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer.
[13] William W. Cohen. 1995. Fast Effective Rule Induction. In *Proceedings of the 12th International Conference on Machine Learning*. 115–123.
[14] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. 2016. A Survey of Man In The Middle Attacks. *IEEE Communications Surveys & Tutorials* 18, 3 (2016), 2027–2051.
[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceeding of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.
[16] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Proceedings of the 22nd Network and Distributed System Security Symposium*. 1–16.
[17] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. 2018. AIM-SDN: Attacking Information Mismanagement in SDN-datastores. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 664–676.
[18] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. 2013. Scalable Network Virtualization in Software-Defined Networks. *IEEE Internet Computing* 17 (2013), 20–27.
[19] Ramakrishnan Durairajan, Joel Sommers, and Paul Barford. 2014. Controller-agnostic SDN Debugging. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo (Eds.). 227–234.
[20] Ramon Ferrús, Harilaos Koumaras, Oriol Sallent, George Agapiou, Tinku Rasheed, M-A Kourtis, C Boustie, Patrick Gélard, and Toufik Ahmed. 2016. SDN/NFV-enabled satellite communications networks: Opportunities, scenarios and challenges. *Journal of Physical Communication* 18 (2016), 95–112.
[21] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies*.
[22] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*. 789–800.
[23] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 50–59.
[24] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 237–248.
[25] Evangelos Haleplidis, Kostas Pentikousis, Spyros G. Denazis, Jamal Hadi Salim, David Meyer, and Odysseas G. Koufopavlou. 2015. *Software-Defined Networking (SDN): Layers and Architecture Terminology.* Information RFC 7426. Internet Research Task Force (IRTF).
[26] Joel M. Halpern, Robert Haas, Doria Avri, Ligang Dong, Weiming Wang, Hormuzd M. Khosravi, Jamal Hadi Salim, and Ram Gopal. 2010. *Forwarding and Control Element Separation (ForCES) Protocol Specification.* Information RFC 5810.
[27] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel C. Briand. 2021. Can Offline Testing of Deep Neural Networks Replace Their Online Testing? *Empirical Software Engineering* 26, 90 (2021), 1–30.
[28] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated Machine Learning: Methods, Systems, Challenges* (1 ed.). Springer.
[29] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. 2017. BEADS: Automated Attack Discovery in OpenFlow-Based SDN Systems. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses*. 311–333.
[30] Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1228–1239.
[31] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 1–6.

[32] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. 2014. Network Innovation using OpenFlow: A Survey. *IEEE Communications Surveys & Tutorials* 16 (2014), 493–512.

[33] Seungsoo Lee, Seungwon Woo, Jinwoo Kim, Jaehyun Nam, Vinod Yegneswaran, Phillip A. Porras, and Seungwon Shin. 2022. A Framework for Policy Inconsistency Detection in Software-Defined Networks. *IEEE/ACM Transactions on Networking* 30, 3 (2022), 1410–1423.

[34] Seungsoo Lee, Seungwon Woo, Jinwoo Kim, Vinod Yegneswaran, Phillip A. Porras, and Seungwon Shin. 2020. AudiSDN: Automated Detection of Network Policy Inconsistencies in Software-Defined Networks. In *Proceedings of the 39th IEEE Conference on Computer Communications*. 1788–1797.

[35] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. 2017. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proceedings of the 24th Network and Distributed System Security Symposium*. 1–15.

[36] Yahui Li, Zhiliang Wang, Jiangyuan Yao, Xia Yin, Xingang Shi, Jianping Wu, and Han Zhang. 2019. MSAID: Automated detection of interference in multiple SDN applications. *Computer Networks* 153 (2019), 49–62.

[37] Jiajia Liu, Yongpeng Shi, Lei Zhao, Yurui Cao, Wen Sun, and Nei Kato. 2018. Joint Placement of Controllers and Gateways in SDN-Enabled 5G-Satellite Integrated Network. *IEEE Journal on Selected Areas in Communications* 36, 2 (2018), 221–232.

[38] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47 (2021), 2312–2331. Issue 11.

[39] Canini Marco, Venzano Daniele, Perešíni Peter, Kostić Dejan, and Rexford Jennifer. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*. 127–140.

[40] Jelena Mirkovic and Peter Reiher. 2004. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. *SIGCOMM Computer Communication Review* 34, 2 (2004), 39—-53.

[41] Christoph Molnar. 2022. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable* (2 ed.). https://christophm.github.io/interpretable-ml-book

[42] John Moy. 1998. *OSPF Version 2*. Information RFC 2328. Ascend Communications, Inc.

[43] Saurav Nanda, Faheem Zafari, Casimer DeCusatis, Eric Wedaa, and Baijian Yang. 2016. Predicting network attack patterns in SDN using machine learning approach. In *Proceedings of the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks*. 167–172.

[44] Raphael Ollando, Seung Yeob Shin, and Lionel C. Briand. 2023. [Artifact Repository] Learning Failure-Inducing Models for Testing Software-Defined Networks. https://doi.org/10.6084/m9.figshare.20701354.v1.

[45] Open Networking Foundation. 2015. *OpenFlow Switch Specification, Version 1.5.1*. Specification ONF TS-025. Open Networking Foundation.

[46] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification*. 460–465.

[47] David C. Plummer. 1982. *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. Information. Internet Engineering Task Force (IETF).

[48] Jon Postel. 1980. *User Datagram Protocol*. Information RFC 768. USC/Information Sciences Institute.

[49] Jon Postel. 1981. *Transmission Control Protocol*. Information RFC 793. USC/Information Sciences Institute.

[50] Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers.

[51] Wajid Rafique, Lianyong Qi, Ibrar Yaqoob, Muhammad Imran, Raihan Ur Rasool, and Wanchun Dou. 2020. Complementing IoT Services Through Software Defined Networking and Edge Computing: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials* 22, 3 (2020), 1761–1804.

[52] Yakov Rekhter, Tony Li, and Susan Hares. 2006. *A Border Gateway Protocol 4 (BGP-4)*. Information RFC 4271. Internet Engineering Task Force (IETF).

[53] Christian Röpke and Thorsten Holz. 2015. SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*. 339–356.

[54] RYU Project Team. 2014. *RYU SDN Framework* (1 ed.). RYU Project Team.

[55] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, Chetan Arora, and Frank Zimmer. 2020. Dynamic adaptation of software-defined networks for IoT systems: A search-based approach. In *Proceedings of the 15th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 137–148.

[56] Apoorv Shukla, Said Jawad Saidi, Stefan Schmid, Marco Canini, Thomas Zinner, and Anja Feldmann. 2020. Toward Consistent SDNs: A Case for Network State Fuzzing. *IEEE Transactions on Network and Service Management* 17, 2 (2020), 668–681.

[57] Michael Smith, Robert Adams Edward, Mike Dvorkin, Youcef Laribi, Vijoy Pandey, Pankaj Garg, and Nik Weidenbacher. 2016. *OpFlex Control Protocol*. Internet Draft draft-smith-opflex-03. Internet Engineering Task Force.

[58] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 518–532.
[59] El-Ghazali Talbi. 2009. *Metaheuristics: From design to implementation* (1 ed.). John Wiley & Sons.
[60] Tao Wang, Fangming Liu, and Hong Xu. 2017. An Efficient Online Algorithm for Dynamic SDN Controller Assignment in Data Center Networks. *IEEE/ACM Transactions on Networking* 25 (2017), 2788–2801.
[61] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. 2016. *Data mining: practical machine learning tools and techniques* (4 ed.). Elsevier.
[62] Seungwon Woo, Seungsoo Lee, Jinwoo Kim, and Seungwon Shin. 2018. RE-CHECKER: Towards Secure RESTful Service in Software-Defined Networking. In *Proceedings of the 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks*. 1–5.
[63] Michał Zalewski. 2016. American Fuzzy Lop — Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt
[64] Peng Zhang. 2017. Towards rule enforcement verification for software defined networks. In *Proceedings of the 2017 IEEE Conference on Computer Communications*. 1–9.
[65] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An Industrial Protocol Fuzzing Framework from a Deep Learning Perspective. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*. 59–67.