Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

SDN Spotlight: A real-time OpenFlow troubleshooting framework

Ramtin Aryan [a,d,*], Anis Yazidi [d,b,c], Frode Brattensborg [d], Øivind Kure [a], Paal Einar Engelstad [a,d]

[a] Department of Technology Systems, University of Oslo, Norway
[b] Department of Computer Science, NTNU, Norwegian University of Science and Technology, Trondheim, Norway
[c] Department of Plastic and Reconstructive Surgery, OuS, Oslo University Hospital, Oslo, Norway
[d] Department of Computer Science, OsloMet – Oslo Metropolitan University, Norway

ARTICLE INFO

Article history:
Received 4 May 2021
Received in revised form 3 March 2022
Accepted 7 March 2022
Available online 16 March 2022

Keywords:
Software-Defined Networking (SDN)
OpenFlow
Probing packet
Troubleshooting
Verification
Policy violation

ABSTRACT

Troubleshooting in SDN-based networks is still a cumbersome task that can overwhelm human attention. Various anomalies, such as installation failure, disordered rules, and loops, remain unnoticed even when the most recent detection methods are used. In this paper, we address the issue of verifying SDN policies by actively probing the data plane. SDN Spotlight is presented as an anomaly detection framework that tries to detect installation failures, rule conflicts, and loops. In contrast to recent work, such as Monocle and Pronto, SDN Spotlight verifies a chain of rules using a single probing packet. This approach also reduces the number of monitoring rules, which has a direct effect on saving TCAM memory usage and minimizing the packet matching time. SDN Spotlight addresses two problems: verifying rule installation and forwarding behavior verification. Within the SDN Spotlight framework, we introduce two different approaches for forwarding anomaly detection: Hedge-SDN Spotlight and Open-SDN Spotlight. Furthermore, we devise an efficient and fast probe generation algorithm that generates one single probing packet per chain of rules. As opposed to other related work, Hedge-SDN Spotlight does not yield false positives and false negatives when detecting loops and forwarding failures. The results of the experiment demonstrate that SDN Spotlight is much faster than the SDNProbe and SDN traceroute method, in some cases by a factor of up to seven times as fast.

1. Introduction

Nowadays, more devices are interconnected, data centers are expanding, businesses encourage bring your own device (BYOD) policies, the Internet of Things (IoT) is on the rise, and end-users span over multiple consumer devices.

As a result of this growth, traditional networks have become complex, hard to manage, prone to errors and logical flaws [1], resulting in time-consuming management and fault handling [2]. Computer network troubleshooting tends to be labor-intensive and requires intricate work. Network outages can seem almost unavoidable and are due to a number of reasons. Typical reasons for an outage include, but are not limited to, human errors such as faulty configurations or malicious activity, equipment malfunction, and force majeure events.

Therefore, networks require frequent reconfiguration in routing, QoS, firewall, etc. The speed and scale of these changes lead to network instability. For this reason, a programmable network is needed to increase network flexibility and to keep the side-effects caused by the changes to a minimum [3]. The advent of Software-Defined Networking (SDN) and its layered architecture has allowed for a dynamic approach to managing and configuring networks. Centralizing the control functionality enables the programmability of network control functions and elements. With the rise of SDN and its high level of abstraction, computer networks are expected to enjoy an optimized dataflow, smarter automation, and added flexibility. By utilizing the capabilities of the OpenFlow protocol, verification of the overall network state is achievable from the central point of view of controllers.

However, this new architecture is prone to errors [4] and bugs [5]. In a broad survey, Kreutz et al. [1] divided common errors into the following categories: controller logic bugs, race conditions, and software bugs and performance disturbance. The categories address problems such as reachability issues, events being processed in different orders, and transient failures caused by CPU spikes. The architectural change and the programmability provided by SDN both allow and encourage the emergence of new approaches to troubleshooting, re-thinking workflow, and the development of new troubleshooting tools.

* Corresponding author at: Department of Technology Systems, University of Oslo, Norway.
E-mail addresses: ramtina@ifi.uio.no (R. Aryan), anisy@oslomet.no (A. Yazidi), frodeb@protonmail.com (F. Brattensborg), oivind.kure@its.uio.no (Øi. Kure), paal.engelstad@its.uio.no (P.E. Engelstad).

https://doi.org/10.1016/j.future.2022.03.014

Researchers have discovered various flaws [6], ranging from faulty protocol implementations to switches prematurely reporting rules as being installed [7]. Firmware errors [8], loss of rule-update messages, or batch-level rule update acknowledgments can result in missing rule faults [6], where rules in the data plane are reported as being installed, or updated, while in fact they are not active or working.

Recently, a suite of research, including Monocle [9], SDNProbe [10], and Pronto [11], has focused on testing the data plane in SDN using probe generation. A probe or probing packet is a specific test packet that can be monitored via the controller and is used to test a specific route. Throughout the research, however, one test rule is always inserted per OpenFlow entry. This leads to an excessive increase in the size of the OpenFlow tables and, unfortunately, wastage of the already scarce TCAM[1] memory, and an increase in the packet matching time. Furthermore, these methods endeavor to modify the target rule for testing. SDNProbe modifies the rules and copies rules to a new flow table to ensure that the normal packets are not affected. However, the above-mentioned research does not address the case of a group of actions in OpenFlow rules, such as Set, Queue, etc.

This paper aims to detect the forwarding failures and loops at the data plane level. Verifying the presence of the installed rules on network devices also falls under the aims of this paper. Therefore, the proposed method does not only detect downlinks and faulty ports, but also uncovers firmware failures. The Hedge-SDN Spotlight and Open-SDN Spotlight verify policy violations and loops using two different approaches. In this paper, we use the terms Hedge-SDN Spotlight and Hedge, and Open-SDN Spotlight and Open interchangeably.

The proposed approaches are efficient since they only use a *very small* number of test packets and monitoring rules as compared to Monocle and Pronto, for which the number of tests can be in the order of thousands, depending on the size of the OpenFlow table. Moreover, the proposed framework is capable of verifying rules with different actions, such as Go, Set, Forward, Drop, Queue, without causing any side-effects for regular network traffic.

The primary contributions of this paper are as follows:

- The Hedge and Open approaches not only verify the forwarding policies on a specific route, but are also able to detect loops, a feature that is missing in Monocle and SDNProbe. Physical failures, such as a downlinks or physical port failure, can also be detected using our suggested approach.
- SDN Spotlight framework adds a minimal number of monitoring rules to the switches. In fact, only a handful of monitoring rules, also called catch-rules, are installed to track the probing packets. This leads to negligible overhead in terms of test rules compared to existing approaches that can deploy thousands of test rules per switch.
- A rapid probe generation algorithm is presented to improve the probing performance compared to previous research. The Hedge approach can verify a certain route that includes rules with a drop value or rewrite the packet header as the action field with high accuracy.
- The SDN Spotlight framework is able to simultaneously check multiple rules using a single probing packet independently of their actions.
- A single catch-rule is added per switch, which decreases the negative effect on the matching time of the switch and, consequently, on the network performance.
- The Hedge methods does not lead to false positives and false negatives in the detection process.

[1] Ternary content-addressable memory (TCAM): high-speed memory capable of searching its entire content in a single clock cycle.

The remainder of the paper is organized as follows. Section 2 presents a brief background on the SDN concept and OpenFlow protocol. In Section 3, we provide a comprehensive overview of the state-of-the-art related to troubleshooting in SDN networks. Section 4 presents the SDN Spotlight architecture. In Section 5, the forwarding failure method is discussed. The loop detection method and accuracy analysis are presented in Section 6 and Section 7, respectively. Finally, the evaluation results are presented in Section Section 8.

2. Background

SDN is a new paradigm for network management that attempts to control the network in a centralized manner via a programmable controller, which possesses a general view of the network topology and statistics. The main goal of the SDN paradigm is to enable the prompt and dynamic configuration of the network in order to improve its performance and enable better monitoring. SDN architecture dissociates the control plane from the data plane, and consists typically of a controller and switches. The switches do not possess any form of intelligence, and just implement the forwarding rules that are dictated to them by the controller, who has an end to end overview of the paths. The controller can manage the network flows dynamically and applies new configurations based on the network condition [12].

The SDN concept offers three main advantages: centralized control, network programmability, and easy integration of virtualization [13]. The SDN idea became applicable since the OpenFlow protocol [14] was standardized in 2009. OpenFlow gives the user remote access to the switches and routers. The protocol prepares a context for modifying the configuration of network appliances remotely and in real-time.

OpenFlow is a standard protocol managing the forwarding behaviors of SDN switches from multiple vendors. It facilitates the SDN controller's management and monitoring of the SDN switches. The protocol programmatically and dynamically controls the forwarding behaviors of SDN switches, and via OpenFlow protocol it sends messages to switches to control the forwarding behavior of a network. An OpenFlow switch can have more than one flow table, which is called a chain. When a packet enters an OpenFlow switch, the packet is checked with flow tables, respectively. Rules in flow tables have three sections, including priority, match field, and action. The priority field defines which rule must be selected if the packet is matched with match fields of several rules. The chosen rule applies the action to the packet according to the corresponding choices: forwarding the packet to a specific port, dropping, or modifying the packet header.

3. Related work

Verifying rule installation and policy verification is of the utmost importance to network administrators. A significant amount of research has focused on this problem in the context of SDN. In this section, we review some of the main research in this area.

Packet history-based monitoring. NetSight [15] tries to improve network visibility by capturing what are known as packet histories. By transparently transposing the control channel between the switches and the SDN controller, NetSight can listen in on information from packets passing by and send the information up to the application. Postcards are created at each node of the packet's journey and contain packet headers, the matching flow table entry, and the output port of the corresponding switch.

RuleScope [16] is a method for accurately and efficiently inspecting the forwarding process. The tool detects forwarding faults in the data plane by looking at missing rules and priority faults and using customized probing packets. Probing with respect to the dependencies and a calculated expected outcome paired with postcards of Netsight [15] will help find priority faults. SERVE [17] computes the desired behavior of a specific rule or switch, and then crafts and injects a probe into the network. The data plane is configured to export the probes after processing. By using SERVE, faster overall state verification is achieved as in other papers, such as [9,18].

Tag-based probes and monitoring rules. Monocle [9] verifies the newly installed rule by generating a distinct probing packet. Before probes are injected into the data plane, catch rules are installed on the switches in order to return the probes to the controller. However, Monocle does not suggest any solutions for testing rules with drop action or for the rules that forward packets to end clients. By tracing network packet trajectories, SDN traceroute [18] tries to verify a sequence of rules in switches and pinpoint forwarding problems in switch and controller logic. To be able to trap the probes sent from the controller, SDN traceroute first colors each switch using a graph color algorithm, and then inserts a small number of high-priority catch-rules into the switches. The probing packets are specially crafted, as they make use of the three 802.1p priority bits in the 802.1q tag in the frame header. SDNProbe [10] is another probe-based troubleshooting method that reduces bandwidth usage by minimizing the number of probing packets. Moreover, randomized test flows are used to improve fault detection accuracy. Previous research, such as ATPG [5] and Pronto [11], have tried to minimize the number of test packets by using a greedy approach, which is an NP-Complete problem and leads to sub-optimal results. However, SDNProbe applies this optimization in polynomial time. Aryan et al. [19] present a Per-rule approach based on the test packet and monitors rules to verify the existence of installed rules on the OpenFlow switches. This paper applies the Per-rule approach in the proposed method to localize the faulty rule.

Proactive troubleshooting. VeriFlow [20] aims to provide verification capabilities for network-wide invariants in real-time. The tool tries to deal with faulty changes before they are applied to the data plane. Being able to track the invariant state makes it possible to block a change if the outcome of the change breaks the desired state. This is done by implementing a slim layer between the controller and the network. Instead of checking the entire network for each change as other research does, [21–23], VeriFlow uses a different method. VeriFlow relies on slicing the network policy into so-called Equivalence Classes (EC), where each class consists of a set of packets affected by the same forwarding actions throughout the network.

NEAt [24] also sits between the controller and the data layer and tries to check new policies and solve possible failures. It tries to check the updates in real-time, repairs possible policy-violating updates on-the-fly, and installs the corrected updates in the forwarding devices. Real-time policy checking requires a fast response, so NEAt uses ECs based on VeriFlow [20] to build a model of packet-forwarding behavior. Then, for each new policy or update, the affected ECs are computed and the checking method is specifically applied.

In summary, the three families of approaches we discussed in this section present a trade-off between detection accuracy and cost. Packet history-based monitoring approaches not only detect anomalies, but also localize them. However, in order to achieve this objective, this type of approach has to generate massive amounts of extra traffic via postcard packets.

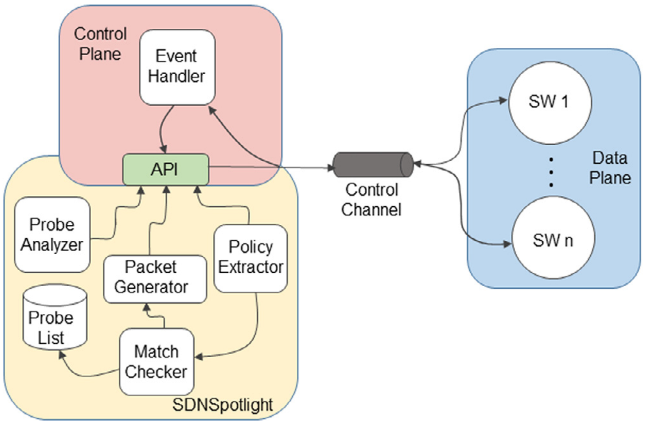

Fig. 1. SDN Spotlight design.

Approaches using tag-based probes and monitoring rules limit the number of probing packets to try and save the network bandwidth. Nevertheless, this optimization usually results in an increase in the rate of false negatives and false positives. The proactive troubleshooting group of approaches verifies the new policies before installing them on forwarding devices. This means that anomalies can be prevented by predicting the side effect of a rule before installation. On the other hand, the prediction method cannot guarantee that all possible anomalies will be detected, particularly when detecting anomalies that occur due to physical failures. Moreover, this group of approaches focuses only on the new rules, and it is not a practical choice when checking the existing installed policies.

4. Intuition and overview of SDN Spotlight

SDN Spotlight attempts to detect policy violations, loops, physical faults, and firmware failures, while verifying rule installation in the data layer. The detection procedure uses a probe-based method to verify a chain of rules via a negligible number of probing packets.

As outlined in Fig. 1, SDN Spotlight is designed as an API that communicates with the controller. The expected path for each specific flow is derived from the high-level policies. The policy extractor handles this process. Meanwhile, some monitoring rules, called catch-rules, are designed to collect the probes. Catch-rules are installed in two different ways in our proposed approaches: the Open approach and the Hedge approach. These approaches are explained in detail in Section 5. The probe is eventually injected into the network, and if there is a violation or firmware fault, then the probe will be caught by an unexpected catch-rule and forwarded to the controller. By an unexpected catch-rule, we mean a catch-rule belonging to a switch that it is not a member of the expected route. These steps are managed by the match checker and packet generator modules. Moreover, the controller will receive a timeout alert in the event of a physical failure or loop. The probe analyzer module keeps track of timeout and performs the analysis of the returned probe.

This section discusses the main building blocks of the SDNSpotlight framework.

4.1. Expected path for header spaces

In our previous work [25] we have presented a formal approach based on second-order logic in order to generate queries with disjoint header spaces to test a policy. In this paper, we have used the same formalism in order to describe rules and queries. We refer the interested reader to paper [25] for more details.

Algorithm 1 aims to find a set of packets that exclusively match the target rule while not matching any of the rules with a higher priority value than the target rule. To achieve this, we follow the raining 2D-Box model logic [26] that we also used in [25] for query generation. According to the raining 2D-Box model, in order to find a set of packets matching only the target rule, we need to recursively exclude all the sets of packets from the header space of the target rule that match one of the rules above it.

The complexity of Algorithm 1 can be expressed as the product of the number of rules that have a higher priority value than the target rule (N), and cost of the subtracting process (s), that is $O(N \times s)$. We assume that s is constant, then $O(N)$.

Algorithm 1: Generating queries from the ingress switch rules.

```
   Input: RuleList
   ;                     /* RuleList represents flow table. */
   Output: QueryList
   ;  /* QueryList represents a list of packet header for each
    target rule. */
1  foreach target_rule in RuleList do
2      foreach rule in RuleList do
3          if rule.Index < target_rule.Index then
4              target_rule = target_rule − rule;
5          end
6      end
7      QueryList ← target_rule
8  end
9  return QueryList;
```

The Tracing function [25] uses queries as input in order to map the expected path of each possible flow. The output consists of a set of packets that traverse the same path and reach the same node at the end. In this sense, the Tracing function retrieves all expected flows through the specific switch(s) based on the high-level policies. The Tracing function is presented in [25] and, for the sake of completeness, we provide the algorithm in Algorithm 2. According to the algorithm, the Tracing function calculates which rules match the input query. Then it predicts the following destinations based on the action of the match rules. The tracing function continues this process recursively until the action resulting from the match rule is 'drop' or 'end client'.

The complexity of Algorithm 2 can be expressed as a product of the number of rules (N), and the sum of subtraction cost (s) and cost of the matching process (m), that is $O(N \times (s + m))$. We can assume that s, m are constant, then $O(N)$.

Algorithm 2: Tracing function

```
    Input: Query, Starting_Node, Route
    Output: All Nodes in the route, Sub-set query
1   Route ← Starting_Node;
2   Rules ← Starting_Node.Rules;
3   foreach rule in Rules do
4       if Query ∩rule.Condition then
5           Route ← rule.Action, Query ∩ rule.Condition;
6           if rule.Action = Client Or rule.Action = Drop then
7               return Route;
8           end
9           Query ← Query - rule.Condition
             Tracing_Function(Query∩rule.Condition,rule.Action, Route);
10      end
11  end
```

At this juncture, we will use a simple example to present the idea. Let us assume that the packet header space consists of two items: source IP and destination IP. Fig. 2 illustrates this idea using a geometric representation. According to Fig. 2, the header space has two dimensions. Based on the order of rules in the ingress switch, half of the header space (part A) is forwarded to switch 1, while the rest of it (part B) is forwarded to switch 2. As shown in Fig. 2 and according to the order of the rules for switch 1 and switch 2, switch 3 receives just half of the header space from switch 1 (part A') and half of the header space from switch 2 (part B'). Switch 4 also receives the rest of the header space from switch 1 and switch 2 (parts A'' and B''). Therefore, the Tracing function returns {ingress switch, switch 1, switch 3 } as an output for A' header space and {ingress switch, switch 2, switch 3} for B' header space. For the header space A'' and B'' the routes are {ingress switch, switch 1, switch 4} and {ingress switch, switch 2, switch 4}, respectively. Parts A', B', A'' and B'' are disjoint and the probing packets can be selected from these header spaces to verify the order of rules and network slicing. This process is time-consuming and inefficient due to the frequent changes in the flow tables. For this reason, the incremental approach [27] is a fast and efficient solution that can be used to cover the frequent rule updates in SDN. This approach tries to only apply the reprocessing of the path of flows to the updated rules and affected parts.

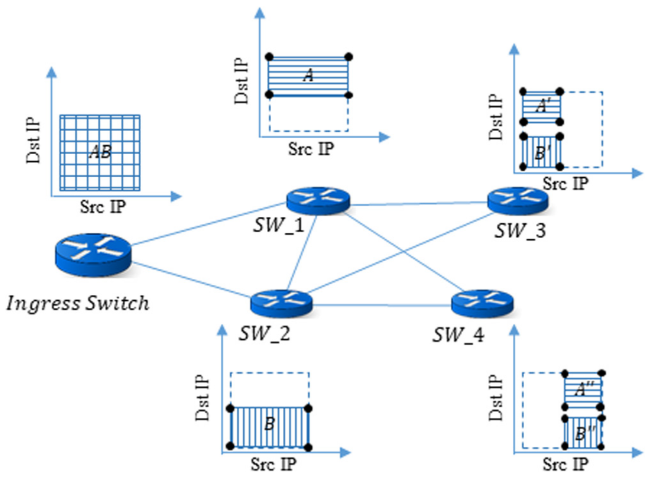

Fig. 2. The effect of trace function on the packet header spaces.

4.2. Generating the probing packets

In this paper, probing refers to the technique used to explore or examine the functionality of rules in the data plane of SDN architecture. In this process, a specific packet, which is called a probing packet or probe, is sent to the data plane via the controller. Based on the predicted path and the header space of each flow, a specific probe is generated for the test.

According to Fig. 2, all members of each header space at the end nodes are matched with the specific sequence of rules that create an expected path. It is not feasible to generate a probe for each member of the header space, since not only is it a time-intensive process, but it also requires a substantial portion of bandwidth. For this reason, only some marginal members of header space, also known as the corner values [28], are used to generate the probes. As Acharya and Gouda [28] demonstrate, using the corner values can guarantee that the verification algorithm has a low error probability (less than 0.06% and, in more practically reasonable cases, as small as 0.01%).

The probing generation process operates on a 5-tuple set consisting of the source and destination IP address, source and destination port number, and the protocol. Please note that we

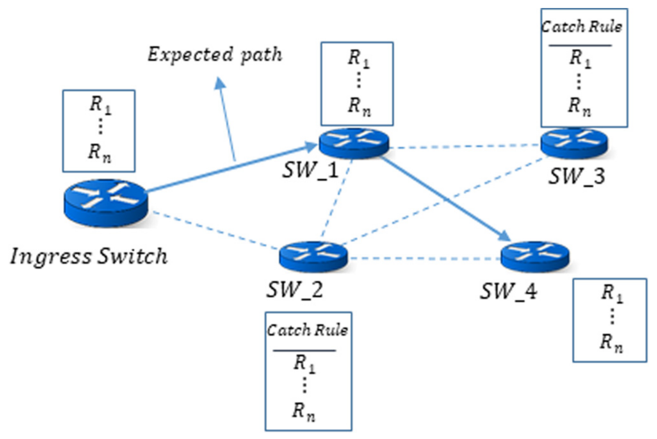

Fig. 3. Installing catch-rules on the switches in the neighborhood of the expected path.

can easily use a higher number of fields without having to change the approach.

The probing packet is the same as normal traffic except for a unique value in the header value. This unique value is applied to distinguish the probing packet. For this reason, the 6-bit Differentiated Services Code Point (DSCP) [29] field in the 8-bit Differentiated Services (DS) field from the IP header is utilized. In order to increase the number of probing packets, we use the DSCP field with a VLAN priority value [30]. Since the probing process is assumed to have 5 tuples, the DSCP and VLAN values do not have any effect on normal network traffic. However, if the number of tuples in the probing packet is increased and includes VLAN or DSCP, we should assign a new field to distinguish the probing packet from normal traffic.

4.3. Generating the catch-rules

The probes can only be matched via catch-rules. The DSCP and VLAN priority fields are used together to allow matching between catch-rules and probes. Therefore the catch-rules do not cause any side-effects for normal network traffic. As shown in Fig. 3, these rules should be installed on the top of the flow table as high priority rules in order to avoid the shadowing anomaly [25]. We use "*OFPPCONTROLLER*" as an action for the catch-rule that forwards the probes to the controller. The OFPPCON-TROLLER action is supported by OpenFlow V1.3 [31] and later versions.

4.4. Timeout computation

SDN Spotlight needs to have a traverse estimation time for a probe packet that is handled by the Probe Analyzer module (Fig. 1). This estimation is required to test the rules with drop action, and also for loop investigation.

The timeout value for the loop detection process is computed based on the delay value for each link in the expected path and is provided by the controller. Moreover, the process time for each switch in the expected path should be added to the timeout value. Therefore, the timeout value is calculated as follows:

$$expected\_delay = \sum_{i \in path} (d_i + p_i) \tag{1}$$

That *path* refers to the expected path, and d_i represents the delay value of the connection to switch i. Moreover, the processing time for switch i is shown as p_i. The link delay, d_i, is retrieved from OpenFlow link statistics while p_i is usually estimated to be constant (although it depends on the position of the rule being matched). In our experiment, the timeout is chosen to be 3 times the expected delay along the path. In some rare cases, the probing packet might have an "unexpected" delay that exceeds the timeout. In such a case, the loop detection process starts after the timeout, and, hopefully, is stopped as soon as the "delayed" probe is received by the controller.

5. Forwarding anomaly detection

In this paper, we aim to detect forwarding anomalies via the probing method. The probing process attempts to examine the specific flow to check whether or not it follows the expected path. The target flow and its predicted path are generated based on Algorithm 1 and Algorithm 2. Then the probing packets are created from the corner values of the header space (see Section 4.1). The catch rules are built and installed on the switches in the neighborhood of the expected path. In addition, one catch-rule is installed on the final node of the expected path and the controller will receive the packet if no failure takes place. Therefore, the probing packet will be sent to the controller via the catch-rules in the neighborhood of the path if the probe does not follow the expected path because of rule installation faults or rule order failure.

The SDN Spotlight uses the Per-rule method which was presented by the authors of this paper in [19] to examine the installation of the specific rule. Moreover, to check the specific path in the network, two approaches, called Hedge-SDN Spotlight and Open-SDN Spotlight, are implemented. SDN Spotlight can also detect multiple faulty rules in the data plane.

During the probing process, the probing packet traverses the network just like normal network traffic. This operation is compatible with the various types of actions in OpenFlow rules such as Go, Set, Forward, Drop, and Queue. Moreover, packet duplication can also be handled by the probing procedure and the cloned probes can be caught by the catch-rules in the same way as they are in the single probe scenario. Nevertheless, for the sake of simplicity, in this paper we will just focus on forwarding rules and scenarios with a random single faulty rule.

5.1. Hedge-SDN Spotlight approach

The Hedge approach can test the path based on the predefined probe after defining the specific probe and its expected path. Initially, the algorithm installs the uniform catch-rules on all switches in the neighborhood of the expected path, in the same way as the sample provided in Fig. 4. The Per-rule approach verifies the installation of the catch-rules. It builds a boundary around the path, which is why we call it the Hedge approach. If the SDN Spotlight receives the probe from an unwanted node, this means that there is a failure. If the last rule in the target path a rule with a drop action or forwarding action to an end client, then the last node should be checked using the Per-rule approach. Therefore, the target path is checked until the final node. In order to verify the target rule in the last node, the catch-rule in the last node should be removed, and the Per-rule should be applied. Moreover, the header of the received probe is checked with the expected header to improve accuracy. More details are given in Section Section 7.

The complexity of the Hedge approach can be presented as $O((M \times c) + p + L)$ where M refers to the number of neighbors, c indicates the cost of installing the catch-rules, p shows the cost of generating the probes, and L refers to the length of the path. We can assume that c and p are constants, thus the complexity can be written as $O(M + L)$. For instance, in Fig. 4, the target path is SW_3, SW_4, and SW_5. Unified catch-rules are installed on SW_1, SW_2, SW_6, and SW_7 as neighbor switches, and SW_5

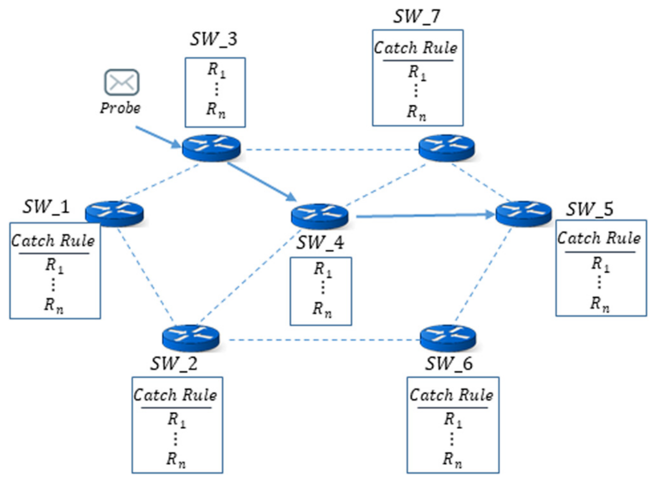

Fig. 4. An example illustrating the Hedge-SDN Spotlight.

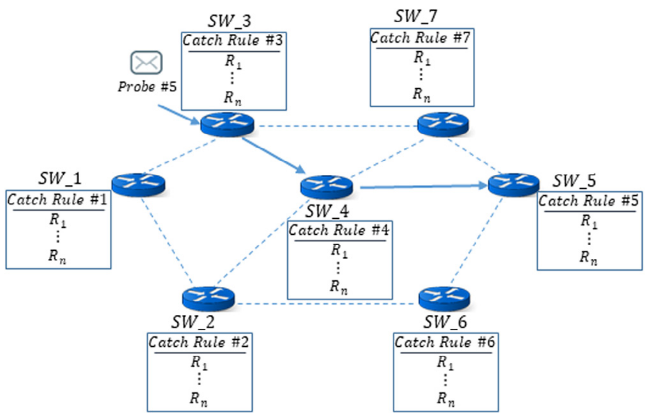

Fig. 5. An example illustrating the Open-SDN Spotlight.

as the final node. Then the probing packet is sent to SW_3 and the timeout value is set based on the path (see Section 4.4). If the controller receives the probe from SW_5, then we can conclude that there is no failure. However, if the probing packet is sent to the controller by SW_2, this means that SW_4 is the faulty switch. In another scenario, if SW_7 receives the probing packet, then the algorithm cannot detect the faulty switch since SW_7 has a connection with SW_3 and SW_4. Then a new probe should be sent via SW_4. If the new probe is received by SW_7, SW_4 is the faulty one; otherwise SW_3 is the faulty switch. To find the faulty rule, the Per-rule method should be run on the faulty switch via the same probing packet.

5.2. Open-SDN Spotlight approach

Like the Hedge approach, the Open approach tries to detect the policy violation for the specific path, however it is based on a different idea. In contrast to the Hedge approach, the Open approach installs catch-rules with unique IDs on all network switches. The idea behind using a unique ID is to allow the catch-rules in the Open approach to only hook the probing packets with the same ID. Just as the Hedge approach, the Per-rule approach verifies the installation of the catch-rules. The Open approach uses an iterative algorithm. In the first iteration, a probing packet with the ID of the last node in the expected path is sent through the network and a timeout value is set based on the expected path (see Section 4.4). The next iteration starts if the target switch cannot catch the probing packet. In the next iteration, the previous node of the target switch of the last iteration is set as the target node. This process is repeated until the target switch hooks the probe. In this case, the target switch of the last iteration is deemed to be the faulty node. Although the Open approach can detect the faulty switch, we need an extra mechanism in order to localize the exact faulty rule of that switch. The Per-rule approach can be used for this purpose.

The complexity of the Open approach can be expressed as $O((V \times c) + p + L)$, where V shows the number of nodes in the network, c presents the cost of installing catch-rules, p refers to the cost of generating the probes, and L shows the length of the path. In this approach, we can assume that c and p are constants, then $O(V + L)$.

As shown in Fig. 5, the expected path includes SW_3, SW_4, and SW_5 and unique catch-rules are installed on all switches. In the first iteration, a probe with ID (#5) is injected. If there is no failure, the packet will be hooked by SW_5 before the timeout is over. Otherwise, there is at least one failure among SW_3, SW_4, and SW_5. In the next iteration, SW_4 is set as the target switch and a new probe with ID (#4) is created and sent, and a new timeout value is set for this step. If the probing packet is not caught by the catch-rule #4 the next iteration is started. In the third iteration, the target switch is SW_3 and the new packet with ID (#3) is sent and a new timeout set for this step. If the probing packet is caught by catch-rule #3 this means that SW_4 is the faulty switch. As mentioned previously, we can use the Per-rule method to detect the faulty rule in SW_4.

The iterative approach uses one probing packet for each iteration. However, it can be time-intensive if the length of the target path is increased. To optimize performance, we have designed a parallel approach. The parallel approach generates the probing packet for each member of the target path and simultaneously sends them through the network. Thus, the controller finds out which packet is hooked by its switch and which one is not. As we will see later in Section 8.4.3, although the detection processing time decreased, the number of probing packets and bandwidth usage increased.

6. Loop detection

Detecting the loops in complex networks is a challenging problem. Detection accuracy and detection speed are the critical criteria when analyzing the performance of the loop detection procedure.

The loop detection process involves two steps, the first of which is the probing process. When the probing starts, a timeout value (see Section 4.4) is set for the procedure, which means that after this time, if the controller does not receive the probe, then there is a loop in the path. In some cases, an unexpected delay causes the probe to be sent back to the controller after timeout expiration. In this scenario, the second step is terminated right after the controller receives the probe.

The second step tries to detect the root cause of the loop. This step is implemented via different algorithms for the Hedge approach, rather than the Open-SDN Spotlight. The details are explained below.

6.1. The loop detection procedure in Hedge-SDN Spotlight

The loop detection algorithm for the Hedge approach classifies the loop scenarios into two groups. In the first group, a loop involves nodes along the expected path and other nodes in their neighborhood. As shown in Fig. 6, by virtue of the Hedge approach, the probe is caught by the first neighbor switch and

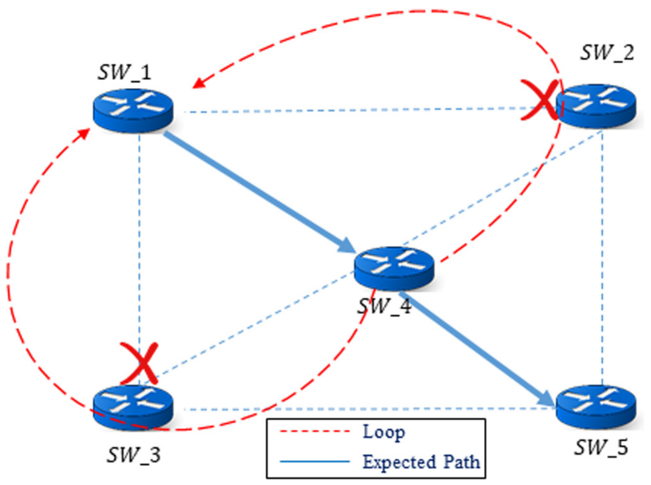

Fig. 6. Loop detection based on the Hedge-SDN Spotlight.

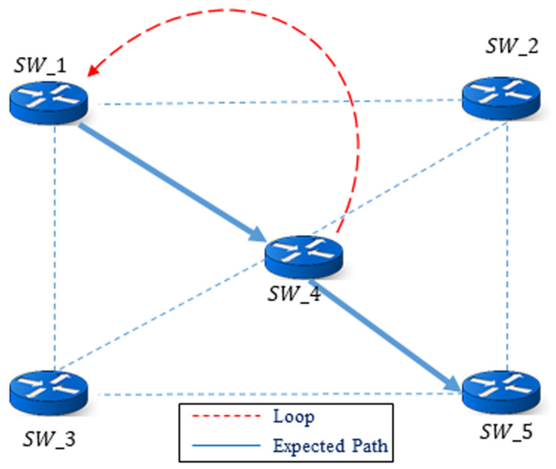

Fig. 7. Loop in the expected path.

forwarded to the controller. Then the probe analyzer module inside the SDN Spotlight finds out whether an unexpected path receives the packet. If this is the case, then the Per-rule can find the faulty rule by checking the previous switch. In this scenario, the loop is detected during the probing process (first step) and it is thus not necessary to check the timeout value.

The second group of loops can be considered to be the worst case scenario for the Hedge approach, as will be explained later. As shown in Fig. 7, in this case the loop is exclusively located inside the expected path. Therefore, the probe cannot be caught by the switches in the neighborhood. In this scenario, the probing process is finished due to timeout.

The second step is an iterative procedure to detect the root cause of the loop, and it is launched after the timeout. As shown in Fig. 8, a catch-rule is installed on the last node of the expected path, which is SW_5 in Fig. 8, and a new probing packet is sent. If the new catch-rule does not receive the new probe, then a new catch-rule is installed on the previous node, which is SW_4 in Fig. 8. This process will be continued until the probe is caught by the last installed catch-rule. In this case, we will find out whether the next node on the expected pass is the faulty node. In the example shown in Fig. 8, the catch-rule in SW_1 in the third iteration catches the probe, and thus we find out that the next node, SW_4, is the faulty one. To find the faulty rule, we check the faulty switch using the Per-rule approach. To stop the probes

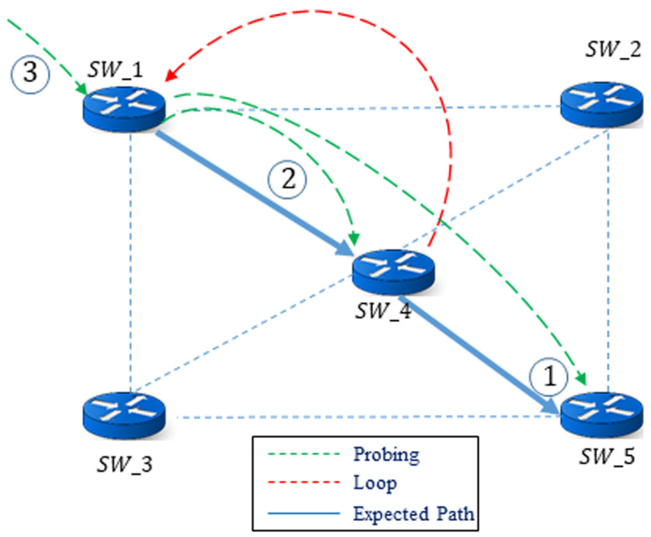

Fig. 8. Detecting the loop in the expected path via Hedge-SDN Spotlight.

that are in the loop during the detection procedure, the catch rule of the faulty node is modified to collect these probes and send them back to the controller.

6.2. The loop detection procedure in Open-SDN Spotlight

In the Open approach, in contrast to the Hedge approach, the loops can be detected irrespective of where they are located. As shown in Fig. 9, after reaching the timeout value, the iterative detection algorithm is launched. First, the probe with the ID of the last node on the expected path, which is SW_5 in this example, is sent. If the probe is not caught, a new probe with the ID of the previous node, which is SW_4 in this example, is sent. Since the Open-SDN Spotlight approach installs a specific catch-rule for each switch in the network, extra catch-rules are not required. Thus, the iterative probing process can be implemented in parallel and the detection performance can be optimized. When a node receives the probe, this means that the node in the previous step is the faulty one. Afterwards, the faulty rule is located via the Per-rule approach. As in the Hedge approach, in order to stop the probes in the loop during the detection procedure, the catch-rule of the faulty node is modified to collect all probes and send them back to the controller.

7. Detection accuracy analysis

In this paper, two main approaches are proposed to detect the policy violation in the data plane. Misreporting a correct policy as a root cause of failure, i.e., a false positive (FP), and the mis-detection of an unexpected node or installation error, i.e., a false negative (FN), are the critical challenges for the proposed methods. Therefore, the possibility of an FP or FN is the main criterion for evaluating the method's accuracy. In this section, we analyze the accuracy of the proposed methods and find out which scenarios trigger an FP or FN.

7.1. Hedge-SDN spotlight approach

The Hedge approach tries to check the expected path for specific network traffic. In this approach, all possible misrouted probes will be hooked since catch-rules are installed on the neighboring switches. Moreover, the root cause of the loop inside the expected path can be identified, as explained in Section 6.1.

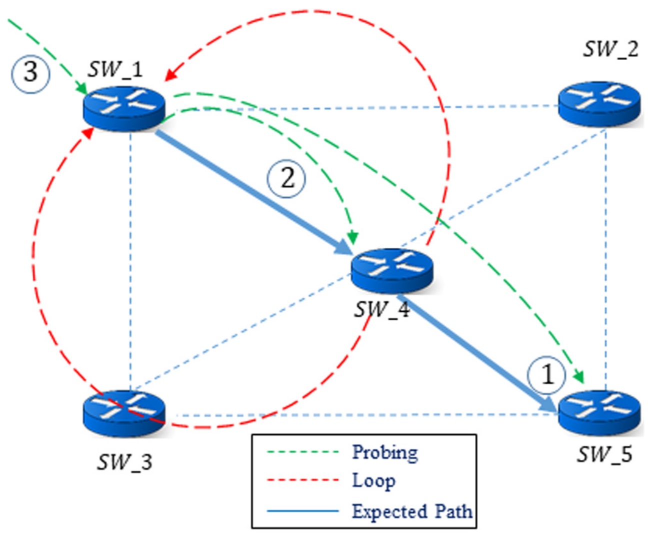

Fig. 9. Detecting the loop in the expected path via Open-SDN Spotlight.

However, some scenarios pose challenges when it comes to the accuracy of the method. For example, consider the case where there is a rule that modifies the packet that has the same action as the target rule. If a rule priority failure occurs, the probing packet will match the wrong rule instead of the target rule. Then the probing packet will still be sent to the expected next node but with a new header value, i.e., a false negative. In the next step, a new probe will be matched with an unexpected rule and will be redirected to the wrong node. So, the probe will be caught by the catch-rule, resulting in a failure. Nevertheless, the wrong switch will be reported as the faulty node, i.e., a false positive.

For example, the expected path in Fig. 10 is SW_1, SW_2, SW_3, SW_4. To follow this path, the probe should be matched with the rule without header modification. If an order conflict occurs in SW_ 2 and the probe is matched with a modification action, then a false negative occurs, since the probe does not leave the expected path. In SW_3, the probe with a new header is matched with an unexpected rule and is forwarded to SW 5 instead of SW_4. In this step, the probe is hooked by the unexpected node (SW_5) and SDN Spotlight reports the failure. If the SDN Spotlight does not check the header of the probing packet, it reports SW_3 as the faulty node, which results in a false positive because SW_2 is the faulty one. However, the Hedge-SDN Spotlight not only checks the hooker switch of the probe against the expected switch, but also compares the probe header with the expected one. So, the SDN Spotlight checks the previous nodes when the modified probing packet is sent via an unexpected switch. Therefore, there are no false positives or false negatives in the Hedge approach.

7.2. Open-SDN Spotlight approach

The Open approach tries to verify the expected path by sending a probing packet that targets the final node on the path. Whenever the expected node receives the probe, we can conclude that there is no failure. However, the probing packet can reach the final node without following the expected path. As shown in Fig. 11, the expected path includes SW_3, SW_4, SW_5, and the final node is SW_5. However, there is a failure in SW_3 and, consequently, the probe is forwarded to SW_7 instead of SW_4. In the last step, the probe is forwarded to SW_5, which is expected, and the algorithm does not detect the failure, i.e., a false negative. In contrast to the Open approach, the Hedge approach can detect the failure since there is a catch-rule in each switch in the neighborhood of the expected path. As shown in Fig. 12, the deviated probe is caught by SW_7 and the algorithm detects the failure in the first step. The summary of the comparison between the SDNSpotlight approaches and between SDN traceroute and SDNProbe are presented in Table 1.

Table 1
Comparisons of detection accuracy.

	Forwarding failure	Loop	False positive	False negative
Per-Rule-SDNSpotlight [19]	✓	✓	✗	✗
Hedge-SDNSpotlight	✓	✓	✗	✗
Open-SDNSpotlight	✓	✓	✗	✓
SDNProbe [10]	✓	✗	✗	✓
SDNTraceroute [18]	✓	✓	✗	✗

Most importantly, we observe that within the SDNSpotlight framework, Per-rule-SDNSpotlight and Hedge-SDNSpotlight are superior to Open-SDNSpotlight when it comes to different aspects of accuracy. In fact, both Per-rule-SDNSpotlight and Hedge-SDNSpotlight do not yield false positive or false negative detection results, and at the same time are able to detect forwarding failures and loops, while Open-SDNSpotlight suffers from possible false positives.

8. Experimental results and evaluation

The SDN Spotlight is implemented using C++ and Python. The API that has been developed for the probing operation is implemented in the Ryu [32] controller in accordance with OpenFlow V1.3 or a higher version.

SDN Spotlight generates a graph model based on the network topology and the flow entries in switches. Then it calculates all possible paths and sets of packets with the same forwarding behavior that might enter the network [25]. This part is implemented in C++ to support parallelization and improve performance. Frequent changes in network policies are handled via the incremental approach presented in our previous work [27]. For troubleshooting purposes, SDN Spotlight crafts specific packets to test the corresponding path in the network. Then the Python API in Ryu injects the probing packet into the network and waits for the feedback.

For the experimental step, we have developed several test networks with Mininet [33], which is a data plane emulator. In this section, the performance of Hedge-SDN Spotlight and Open-SDN Spotlight is evaluated. Moreover, the performance of our SDN Spotlight framework is compared with SDN Traceroute, SDNProbe, and Per-rule approach.

8.1. Real-life environment

To demonstrate the capability of SDN Spotlight in a realistic environment, we mimic a real-life scenario using a generated topology based on a real ISP configuration from the Rocketfuel dataset [34]. Fat-tree networks, which are one of the standard topologies for the large networks, data centers, etc., with 10 different configurations, are used to test the effect of network size, length of the path, and the number of neighbors for each path member on the performance of the different methods.

8.2. Evaluation metrics

We consider the processing time needed to generate the model, the probing processing time, the number of the probing packets, and the number of additional installed rules as metrics for comparing the performance of SDN Spotlight with SDNProbe,

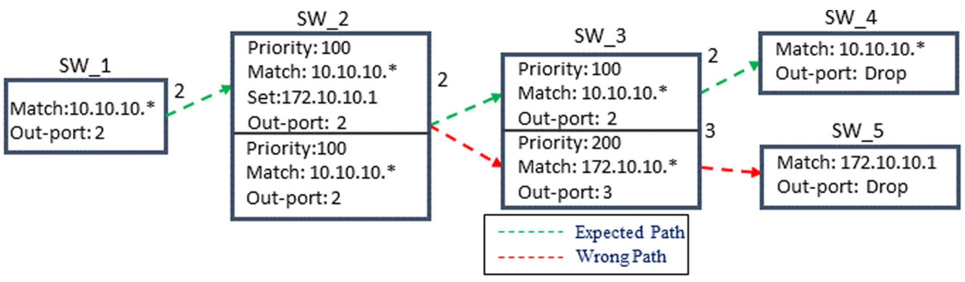

Fig. 10. False negatives and False positives in same-time.

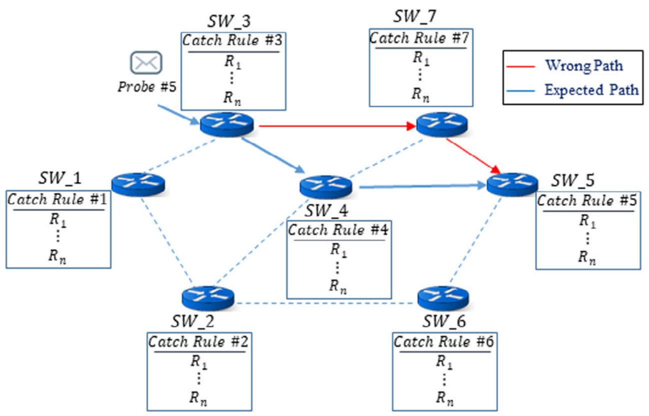

Fig. 11. False negative example in the Open-SDN Spotlight approach.

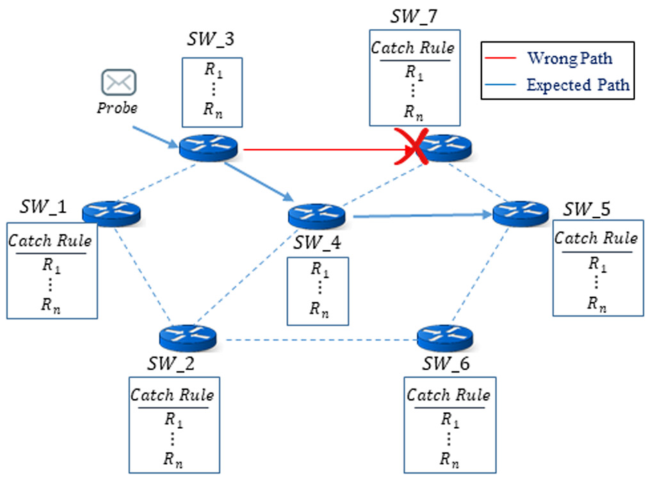

Fig. 12. The way that the Hedge-SDN Spotlight covers the false negatives in the Open-SDN Spotlight approach.

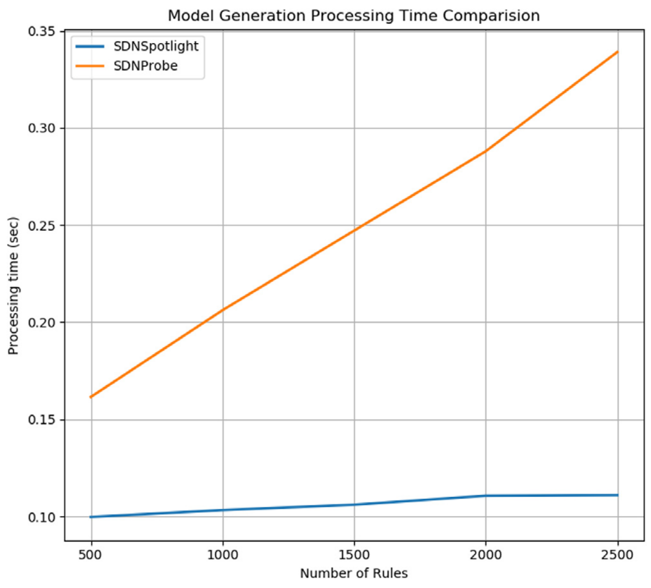

Fig. 13. Model generation processing time comparison.

Per-rule, and the SDN traceroute method. The aforementioned metrics help us compare the overhead and the speed of methods.

Since the accuracy of the Hedge approach comes at the expense of dynamic updates and the installation of the catch-rules, we also assess the installation time for the catch-rules in the Hedge approach, which can be seen as the overhead for a solution that is free of false positives and false negatives.

8.3. Validation

In this paper, we use the same topology and configuration to compare the result of the SDNSpotlight with the SDNProbe, and SDN traceroute. To this end, and in the same manner as some noteworthy past research [9,10,18,20], we install a fixed set of OpenFlow rules on switches and then randomly remove one rule at a time from the data plane, so as to generate a failure. Then we validate the methods by comparing the output of each method.

8.4. Results

In this section, the effect of network size on different methods is presented. The evaluation is divided into different steps: model generation, installing additional rules, probing, loop detection, and overhead analysis. We have explained each step and its corresponding results in detail below.

8.4.1. Generating the model

The SDN Spotlight and SDNProbe methods in the first step rely on generating a graph model based on the network policies and topology. Then, all paths and testing packets are defined based on this model. In this step, we compare the effect of different network sizes on the modeling, calculating all valid paths and defining the packet header for each path. The performance of model generation in SDN Spotlight and SDNProbe is evaluated via 5 networks that consist of 10, 20, 30, 40, and 50 switches, respectively. All the switches consist of 50 rules. The result of the comparison is shown in Fig. 13.

The results show that SDN Spotlight generates the network model faster than SDNProbe. Moreover, SDN Spotlight uses an incremental approach [27], which boosts the performance of generating the new model after each policy modification.

Table 2
Comparison between SDN Spotlight and SDNProbe approaches in catch-rule installation.

Number of switches	10	20	30	40	50
Number of paths	471	880	1224	1513	1757
Number of neighbors	5	10	15	20	25
Number of catch-rules in SDNSpotlight	6	11	16	21	26
Number of catch-rules in SDNProbe	942	1760	2448	3026	3514

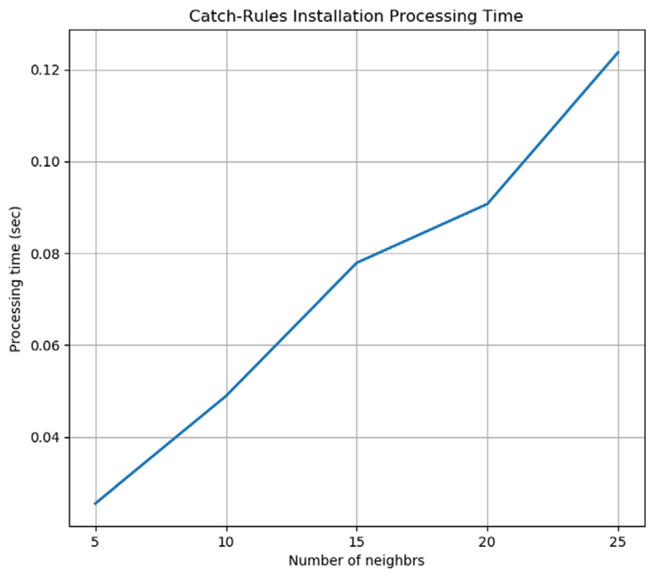

Fig. 14. Catch-Rule installation processing time.

8.4.2. Installing additional rules

In this section, the number of newly installed rules via the proposed method and the SDNProbe are compared for networks of different sizes and topologies, and are presented in Table 2.

According to the SDNProbe approach, two new rules should be installed for each path. Therefore, a significant amount of rules should be installed if there are a large number of paths. As described in Table 2, the number of paths is larger than the number of switches in the neighborhood. Therefore, installing a large number of catch-rules in the SDNProbe approach negatively affects the processing time compared to the SDN Spotlight approach. For instance, the processing time required to install additional rules in the SDN Probe approach for 50 switches is 3 s. However, SDN Spotlight needs only 0.12 s to install the additional rules for 50 switches. Moreover, for the topology with 50 switches, SDNProbe installs 3514 catch-rules. However, SDNSpotlight installs 26 catch-rules for the same topology, which means that SDNSpotlight installs less than 0.8% of the installed rules in SDNProbe. This demonstrates that SDNSpotlight uses less memory in the switches and also uses less network bandwidth when testing a routing policy.

Furthermore, we study the performance of SDN Spotlight for a different number of switches in the neighborhood of the test path. As presented in Fig. 14, the installation process shows an almost linear pattern as we vary the number of neighboring switches. Moreover, this process can be time-intensive when using the Hedge approach for a large number of neighbors that might exceed the number of test paths. The result shows that the processing time for a large network with 25 neighbors for each node is 12 ms, which is fast.

8.4.3. Probing

In this section, we evaluate and compare the processing time for the probing step using the different methods.

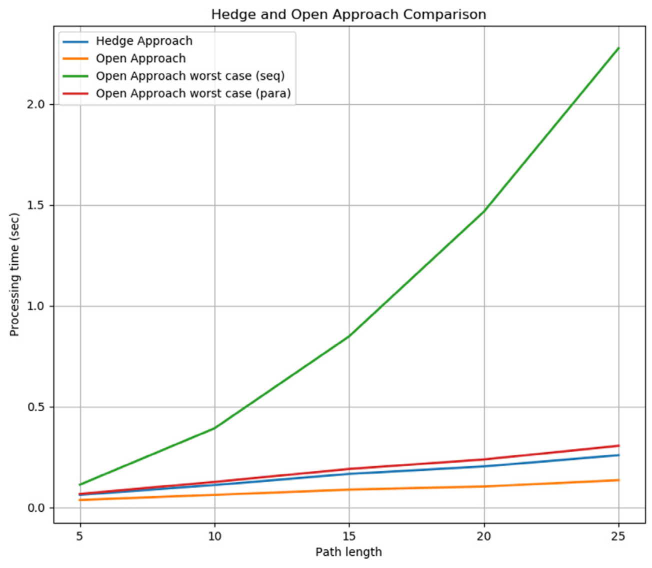

Fig. 15. Comparison between the Hedge- and Open-SDN Spotlight.

We present two different approaches within the SDN Spotlight framework. In this section, we evaluate these approaches based on different path lengths. The assessment procedure includes the worst case scenario, in addition to scenarios with no policy failure. In the worst case scenario, there is one failure in the last node, which makes the detection procedure slower.

Moreover, the parallel and sequential approaches of the Open-SDN Spotlight method are evaluated. In the sequential approach, the path members are checked one after another, however, all the path members are checked simultaneously in the parallel approach.

As illustrated in Fig. 15, the Open approach is fastest whenever there is no failure. However, if there is a failure, the Hedge approach performs better. This Hedge approach has the same processing time irrespective of whether there is a failure or not. The Open method with the sequential approach has the slowest processing time if there is a failure. Parallelizing the Open approach yields a higher performance as compared to the sequential approach, but this requires a separate probing packet for each path member.

Although the Open approach does not require separate catch-rules to be installed for each new path, it is time-consuming to have to find the root cause of the failure.

As mentioned in Table 1, the Per-rule and Hedge-SDN Spotlight have high accuracy in comparison to the other aforementioned methods. For this reason, we have compared the performance of these methods. A comparison between the Per-rule method and Hedge-SDN Spotlight is shown in Fig. 16. As expected, the Per-rule approach is time-intensive as compared to the Hedge-SDN Spotlight, since it tries to check every single rule. In contrast, Hedge-SDN Spotlight tries to check rules that are on the same path, using only one probe.

To evaluate the SDN Spotlight method, we compare the performance of both the Hedge and Open approaches with the SDNProbe, which has similar functionality. Moreover, we implemented the SDN traceroute [18] method for comparison purposes. As shown in Fig. 17, the Hedge approach performs better in scenarios where the number of neighboring switches is lower that than the number of paths. However, SDNProbe and SDN traceroute perform more slowly, since they have to install several catch-rules.

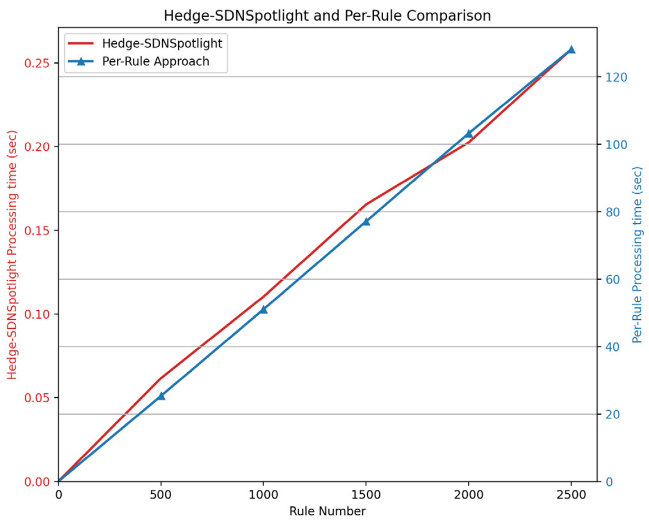

Fig. 16. Comparison between the Hedge-SDN Spotlight and Per-Rule probing processing time.

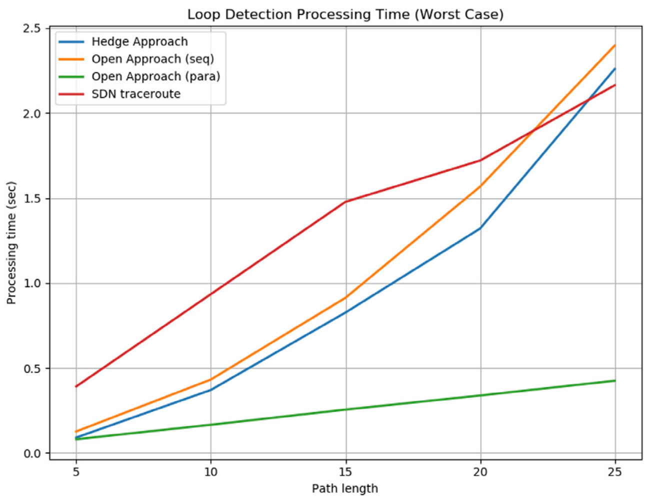

Fig. 18. Loop detection processing time comparison.

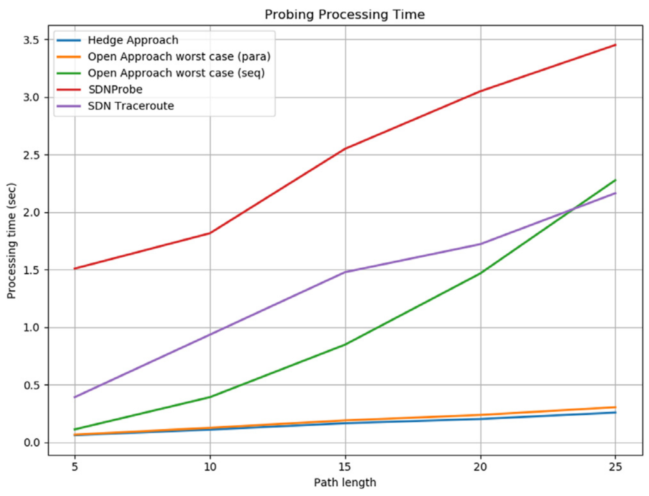

Fig. 17. Probing processing time comparison.

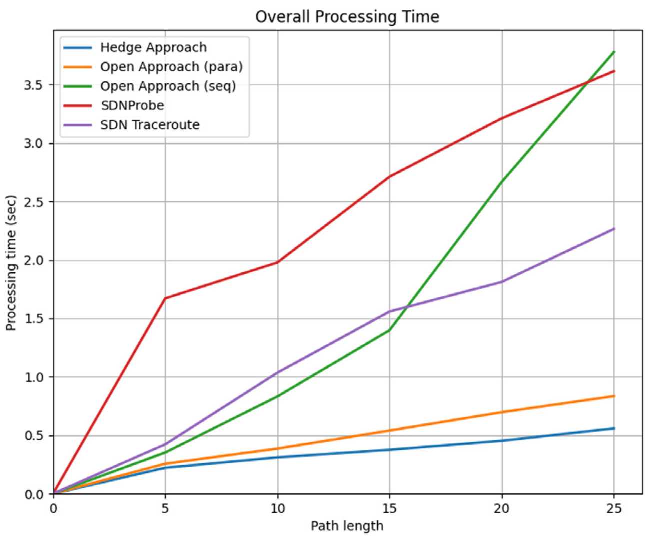

Fig. 19. Overall processing time comparison.

8.4.4. *Loop detection*

As mentioned in Section 6, a loop detection procedure has been developed for the Hedge and Open-SDN Spotlight approaches. In this section, we evaluate the performance of the loop detection procedure for the proposed methods and compare them. SDNProbe does not define a procedure for loop detection, however, SDN traceroute detects the loop during the probing process by finding out whether the probe passed through a switch for a second time. We compared the performance of the SDN Spotlight and traceroute approaches for the worst case scenario and the result is presented in Fig. 18. The worst case scenario refers to a scenario where the last node in the path is the root cause of a loop. Moreover, for the Hedge approach, the loop is inside the expected path. So, the detection procedure tries to check all of the test path members one by one, as explained in greater detail in Section 6.

In regards to Fig. 18, SDN traceroute shows the slowest performance. However, it can detect the loop during the probing process and does not need an extra procedure to detect the root cause behind the failure. Open-SDN Spotlight with a sequential approach performs more slowly than the Hedge approach. Although the Hedge approach has high accuracy, it has a higher processing time than the parallel approach of Open-SDN Spotlight. As shown in Fig. 18, SDN traceroute performs better if the length of the loop is greater than 20 nodes. In this case, SDN traceroute is faster than Hedge-SDN Spotlight, although when using the parallel approach, Open-SDN Spotlight still detects the loop faster.

In Section 8, we present comparisons between the SDNSpotlight approaches and some other approaches using different performance metrics related to the different involved operations: model generation, probing process, and loop detection process. A comparison of the processing time between the proposed methods and other related work is given in Fig. 19. The result is based on the scenario, which contains a loop and a forwarding failure. The overall processing time includes generating a model, installing catch-rules, creating and sending probing packets, and analyzing the probes.

According to Fig. 19, SDNProbe is the slowest method for the target length between 5 to 20 nodes in terms of the overall processing time. In contrast, when it comes to probing and loop

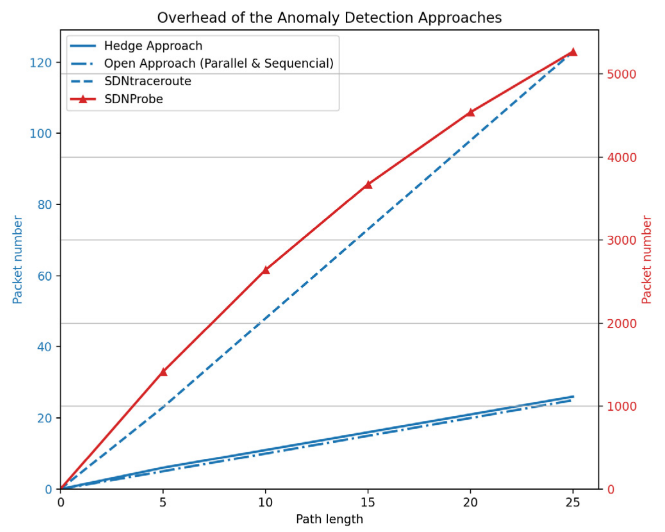

Fig. 20. The overhead comparison between different anomaly detection methods.

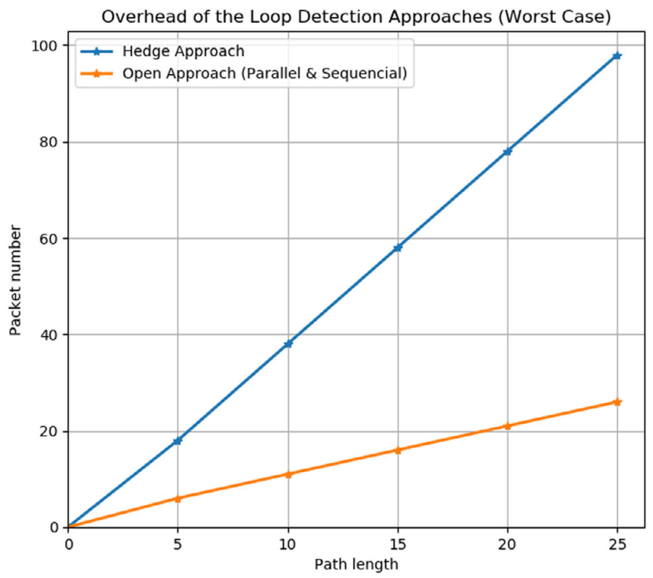

Fig. 21. The overhead comparison between the loop detection method of the Hedge and the Open approaches in the worst case.

detection, the sequential Open approach is slower than SDNProbe and SDN traceroute for the target path that is longer than 15 nodes. The overhead of checking each node of the target path in the sequential Open approach causes a sharp increase in the processing time for the longer routes.

8.4.5. Overhead analysis

The overhead of the detection methods in the network is one of the important metrics for evaluation. In this section, the overhead of the Hedge and Open approaches in the network is analyzed and compared with the SDN traceroute and SDNProbe methods. When we talk about overhead, we refer to the number of packets that are generated in order to install the catch-rules and probing packets.

As shown in Fig. 20, SDNProbe has the biggest overhead in the network since it generates additional rules for all possible flows in order to verify a specific flow and detect the root cause of the failure.

SDN Traceroute needs two probing packets for each hop. Moreover, in each step it should update the catch-rules in the neighbors of the target switch. Therefore, it has a higher overhead than the SDN Spotlight framework approaches. Since SDN traceroute uses a hop-based approach, it has less overhead than SDNProbe.

The Hedge and Open approaches generate fewer probing packets than the SDN traceroute method. Moreover, the number of catch-rules that the SDN Spotlight approaches install is notably lower than the number of flows in the SDNProbe method. In contrast to SDN traceroute, the approaches in SDN Spotlight do not need to change the catch-rules frequently in the probing scenario.

The Open approach entails less overhead in the network than the Hedge approach, since it only installs catch-rules once and does not need to change them. On the other hand, the Open approach for detecting the root cause of failure needs more probing packets. This is why there is no significant difference between the overheads of the Hedge and Open approaches.

Contrary to previous studies, the Hedge and Open approaches have a specific method to detect the root cause of loops. The overhead of detecting the root cause of a loop via both approaches in the worst case is compared in Fig. 21.

As shown in Fig. 21, the Open approach has less overhead in the network than the Hedge approach. The Hedge approach will install additional rules several times in the worst case scenario to detect the root cause. The Open approach generates a similar number of packets in both parallel and sequential methods. However, the parallel method sends all packets simultaneously, which occupies more network bandwidth than the sequential method.

9. Conclusion

SDN policy verification and loop detection are important research topics. Although several policy violation detection approaches exist in the literature, most of them try to detect violations among the rules that are installed on the forwarding devices. Such approaches cannot detect failures of rule installation or physical port errors. Moreover, few recent studies have attempted to address the anomaly detection issue by using the common idea of installing a test rule per target rule. This leads to an excessive increase in the size of OpenFlow tables and, unfortunately, to wastage of already scarce TCAM memory, and an increase in the packet matching time. In this paper, we propose an efficient probe-based detection framework that uses a negligible number of test rules. Our method detects firmware OpenFlow rule installation, forwarding anomalies, and loops. Moreover, physical failures such as port failure or unstable links can be discovered by our suggested mechanism. We introduce the concept of catch-rules to hook the probing packet and forward it to the controller for analysis. Moreover, the proposed method supports all types of rules: forward, drop, and set. Our framework consists of two main methods: *Hedge*, and *Open*. The *Hedge* and *Open* methods have the same goal and can be seen as alternative methods that each have advantages and disadvantages. The *Hedge* approach is both free of false positives and false negatives, but this comes at the expense of an extra overhead in terms of installing more catch rules, while the *Open* approach has a more lightweight detection procedure, with a low likelihood of false-positive and false-negative results. The results of the experiment are very promising and show that this method can be used in a production environment. As future work, we would like to extend our method in order to not only detect the root cause of failures, but also to resolve them while automatically taking the network invariants into account. Moreover, we are working on optimizing the algorithm for parsing network nodes and using a binary search approach to detect the root cause of a failure.

CRediT authorship contribution statement

Ramtin Aryan: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing – original draft, Writing – review & editing, Visualization. **Anis Yazidi:** Conceptualization, Methodology, Supervision, Writing – review & editing, Project administration. **Frode Brattensborg:** Conceptualization, Methodology, Software, Validation, Formal analysis, Writing – original draft, Writing – review & editing, Visualization. **Øivind Kure:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Paal Einar Engelstad:** Conceptualization, Methodology, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] D. Kreutz, F.M. Ramos, P. Verissimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: A comprehensive survey, Proc. IEEE 103 (1) (2015) 14–76.

[2] T. Benson, A. Akella, D.A. Maltz, Unraveling the complexity of network management, in: NSDI, 2009, pp. 335–348.

[3] M.M. Tajiki, B. Akbari, N. Mokari, Optimal qos-aware network reconfiguration in software defined cloud data centers, Comput. Netw. 120 (2017) 71–86.

[4] Q. Zhi, W. Xu, Med: The monitor-emulator-debugger for software-defined networks, in: IEEE INFOCOM 2016-the 35th Annual IEEE International Conference on Computer Communications, IEEE, 2016, pp. 1–9.

[5] H. Zeng, P. Kazemian, G. Varghese, N. McKeown, Automatic test packet generation, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, ACM, 2012, pp. 241–252.

[6] M. Kuzniar, P. Peresini, D. Kostić, Providing reliable FIB update acknowledgments in SDN, in: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, ACM, 2014, pp. 415–422.

[7] M. Kuzniar, P. Peresíni, D. Kostic, What you need to know about SDN flow tables, in: PAM, 2015.

[8] M. Kuzniar, P. Peresini, D. Kostic, Proboscope: Data Plane Probe Packet Generation, Tech. Rep., 2014.

[9] P. Perešíni, M. Kuźniar, D. Kostić, Monocle: Dynamic, fine-grained data plane monitoring, in: Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, ACM, 2015, p. 32.

[10] Y.-M. Ke, H.-C. Hsiao, T.H.-J. Kim, Sdnprobe: Lightweight fault localization in the error-prone environment, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2018, pp. 489–499.

[11] Y. Zhao, H. Wang, X. Lin, T. Yu, C. Qian, Pronto: Efficient test packet generation for dynamic network data planes, in: Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, IEEE, 2017, pp. 13–22.

[12] J. Son, R. Buyya, A taxonomy of software-defined networking (SDN)-enabled cloud computing, ACM Comput. Surv. 51 (3) (2018) 1–36.

[13] T. Bakhshi, State of the art and recent research advances in software defined networking, Wirel. Commun. Mob. Comput. 2017 (2017).

[14] K. Greene, TR10: Software-defined networking. mit technology review. URL http://www2.technologyreview.com/article/41-2194/tr10-software-defined-networking/.

[15] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, N. McKeown, I know what your packet did last hop: Using packet histories to troubleshoot networks, in: 11th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 14), 2014, pp. 71–85.

[16] X. Wen, K. Bu, B. Yang, Y. Chen, L.E. Li, X. Chen, J. Yang, X. Leng, Rulescope: Inspecting forwarding faults for software-defined networking, IEEE/ACM Trans. Netw. 25 (4) (2017) 2347–2360.

[17] Y. Zhao, P. Zhang, Y. Wang, Y. Jin, Sdn-enabled rule verification on data plane, IEEE Commun. Lett. (2017).

[18] K. Agarwal, E. Rozner, C. Dixon, J. Carter, Sdn traceroute: Tracing SDN forwarding without changing network behavior, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ACM, 2014, pp. 145–150.

[19] R. Aryan, F. Brattensborg, A. Yazidi, P.E. Engelstad, Checking the OpenFlow rule installation and operational verification, in: 2019 IEEE 44th Conference on Local Computer Networks (LCN), IEEE, 2019, pp. 250–253.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, P.B. Godfrey, Veriflow: Verifying network-wide invariants in real time, in: Presented As Part of the 10th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 13), 2013, pp. 15–27.

[21] E. Al-Shaer, S. Al-Haj, Flowchecker: Configuration analysis and verification of federated OpenFlow infrastructures, in: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration, ACM, 2010, pp. 37–44.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, S.T. King, Debugging the data plane with anteater, in: ACM SIGCOMM Computer Communication Review, Vol. 41, ACM, 2011, pp. 290–301.

[23] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, S. Whyte, Real time network policy checking using header space analysis, in: Proc. 10th USENIX Conf. Networked Syst. Des. Implement, 2013, pp. 99–112.

[24] W. Zhou, J. Croft, B. Liu, E. Ang, M. Caesar, Automatically correcting networks with NEAt, in: 15th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 18), 2018, pp. 595–608.

[25] R. Aryan, A. Yazidi, P.E. Engelstad, Ø. Kure, A general formalism for defining and detecting OpenFlow rule anomalies, in: 42nd IEEE Conference on Local Computer Networks, Institute of Electrical and Electronics Engineers (IEEE), 2017.

[26] T. Chomsiri, C. Pornavalai, Firewall rules analysis., in: Security and Management, Citeseer, 2006, pp. 213–219.

[27] R. Aryan, A. Yazidi, P.E. Engelstad, An incremental approach for swift OpenFlow anomaly detection, in: 2018 IEEE 43rd Conference on Local Computer Networks (LCN), IEEE, 2018, pp. 502–510.

[28] H.B. Acharya, M.G. Gouda, Linear-time verification of firewalls, in: 2009 17th IEEE International Conference on Network Protocols, IEEE, 2009, pp. 133–140.

[29] IANA, Differentiated services field codepoints (dscp). URL http://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.

[30] R.F.C. 4675, Radius attributes for virtual lan and priority support. URL https://tools.ietf.org/html/rfc4675#page-4.

[31] Openflow official webpage, 2020, https://www.opennetworking.org/sdn-resources/openflow (Accessed: 2020-09-29).

[32] Ryu:, 2020, https://ryu.readthedocs.io/en/latest/index.html (Accessed: 2020-09-29).

[33] Mininet:, 2020, http://mininet.org/ (Accessed: 2020-09-29).

[34] Rocketfuel:, 2020, http://research.cs.washington.edu/networking/rocketfuel// (Accessed: 2020-09-29).

Ramtin Aryan received the B.Sc. degree from the Shahroud University of Technology, Iran, in 2009, and the M.Sc. degree from Shiraz University, Iran, in 2013. Ramtin is a Ph.D. fellow at the University of Oslo. His Ph.D. topic is about anomaly detection between OpenFlow rules. Ramtin does research in network security and network administration, and has almost eight years of experience in network security. Ramtin currently works at the Department of Computer Science, Oslo Metropolitan University.

Anis Yazidi received the M.Sc. and Ph.D. degrees from the University of Agder, Grimstad, Norway, in 2008 and 2012, respectively. He was a Researcher with Teknova AS, Grimstad, Norway. He is currently a Full Professor with the Department of Computer Science, Oslo Metropolitan University, Oslo, Norway, where he is leading the Research Group in applied artificial intelligence. His current research interests include machine learning, learning automata, stochastic optimization, and autonomous computing.

Frode Brattensborg received the M.Sc degree from the University of Oslo, Norway in 2018. His M.Sc thesis title is “Rule Verification in Software-Defined Networks by Passively Probing the Data Plane”.

Øivind Kure received the Ph.D. degree from the University of California at Berkeley, Berkeley, CA, USA, in 1988. He is a Full Professor with the Department of Technology Systems, Section for Autonomous Systems and Sensor Technologies Research Group, University of Oslo, Oslo, Norway. He worked as a Senior Researcher and a Research Manager with Telenor Research, Fornebu, Norway, from 1989 to 2000. His current research interests include various aspects of QoS, data communication, performance analysis, and distributed operating systems.

Paal E. Engelstad received the bachelor's degree in physics from the Norwegian University of Science and Technology (NTNU) in 1993, the master's (Hons.) degree in physics from NTNU/Kyoto University, Japan, in 1994, and the Ph.D. degree in computer science from the University of Oslo in 2005. He is currently a Full Professor with Oslo Metropolitan University. He is also a Research Scientist with the Norwegian Defence Research Establishment and a Professor with the Autonomous Systems and Sensor Technologies Research Group, Department of Technology Systems, University of Oslo. He holds a number of patents and has been publishing a number of articles over the past years. His current research interests include fixed, wireless and ad hoc networking, cybersecurity, machine learning, and distributed and autonomous systems.