A General Formalism for Defining and Detecting OpenFlow Rule Anomalies

Ramtin Aryan
Department of Informatics
University of Oslo
Oslo, Norway
Email: ramtina@ifi.uio.no

Anis Yazidi
Department of Computer Science
HiOA
Oslo, Norway
Email:anis.yazidi@hioa.no

Paal Einar Engelstad
Department of Computer Science
HiOA
Oslo, Norway
Email:paal.engelstad@hioa.no

Øivind Kure
NTNU
Trondheim, Norway
Email: okure@item.ntnu

***Abstract*—SDN network's policies are updated dynamically at a high pace. As a result, conflicts between policies are prone to occur. Due to the large number of switches and heterogeneous policies within a typical SDN network, detecting those conflicts is a laborious and challenging task. This paper presents two-fold contributions. First, we devise an offline method for detecting unmatched OpenFlow rules, i.e., those rules that are never fired. At the heart of our scheme is a formal approach for predicting the packet's path inside a SDN network. In this perspective, we proffer the taxonomy: invalid and irrelevant anomalies for the unmatched rules. Second, we introduce a new set of definitions for the intra-anomalies, which might occur when using the OpenFlow rule's *multi-action* feature. We provide some comprehensive experimental results that show the feasibility of our approach and its ability to scale within large SDN network.**

***Index Terms*—Anomaly Definition, Anomaly Detection, Unmatched rules, Software Defined Network, OpenFlow.**

I. Introduction

Packet forwarding in modern networks is a complex process that involves several devices such as switches, routers and firewall operating with different protocols and configurations. A local device's policy might be conflicting with another appliance's policies. The task of management of network middleboxes manually has been proven to be complex, error-prone, costly and inefficient for many large-networked organizations [1]. For instance, more than 1000 configuration errors have been observed in border gateway protocol (BGP) routers [2]. A single misconfigured router is able to compromise the correct operation of the whole Internet for hours [2], [3]. Meanwhile, human errors are known to be the most common contributor to network downtime. Those errors are responsible for 50 and up to 80 percent of network device outages [4]. Errors typically include loops, suboptimal routing, black holes and access control violations. Moreover, manual troubleshooting in large networks is often almost impossible. Software-defined network (SDN) paradigm addresses these challenges by automating the network control process. SDN separates the control plane from the forwarding devices via a standard protocol called "OpenFlow" [5]. SDN controller has control over the state of the network, thus, it possible to analyze network misconfiguration in a centralized manner. Various methods have been presented to deal with misconfiguration challenges. Those methods fall under two main categories: rule-based verification and log-based verification. Rule-based verification examines the semantics of the rules for detecting policy violation. For instance, when it comes to firewall misconfiguration, Al-Shaer and Hamed [6] propose a tool called Firewall policy advisor. In the same context, Rezvani and Aryan [7] resorted to propositional logic to detect policy violations between a new inserted rule and the combination of existing rules. The other family of approaches, log-based methods, detects policy violations based on mining the network appliance's logs [8], [9]. Furthermore, policy checking could also be categorized based on the different aspects that focus on the Intra-and Inter-devices. For example, the work of Al-Shaer and Hamed [6] and Rezvani and Aryan [7] try to detect misconfiguration on the single device, namely, firewall rule anomalies. On the other hand, both VeriFlow [10] and the work of Kazemian et al. [11] are able to check policy violation among connected devices in the network. In this paper, we tackle rule-based approach for detecting misconfigurations in SDN, which supports both multi-action and single-action Openflow rules. We catalogue the main contributions of this paper as follows:

- A new definition for the intra-anomalies has been described for the OpenFlow with *multi-action* feature. This generalizes the state-of-the-art intra-anomalies definitions, which are only based on single-action feature.
- We introduce the nomenclature *irrelevant* and *invalid* rule anomalies for the case of unmatched rules.
- The detection method supports parallelization by generating completely disjoint queries.
- Our query-based proposed method covers whole policy segments, and therefore it is more efficient than the ping-based troubleshooting methods that operate on a packet basis.
- Our suggested method, in contrast to Netplumber and VeriFlow, considers intra-rule dependency in flow tables [12].
- In contrast to the Header Space Analysis (HSA), our method is a priority-based method which makes it compatible with the OpenFlow protocol.

The remainder of the paper is organized as follows. In Section II, we provide a comprehensive overview over the state-of-the-art. Section III discusses our formal tracing method.

In Section IV, the OpenFlow rules' anomalies definition and detection are explained, and finally the evaluation results are presented in Section V.

II. Related Work

In recent years, a significant amount of research has addressed network policy conflict analysis. A notable work is due to Kazemian et al. [11] who introduced a real time policy checking tool based on HSA [11] called NetPlumber. In contrast to the HSA, NetPlumber checks the real time network traffic incrementally. The authors proposed a new formal language to express policy checks, which is fast enough for use as a real-time traffic monitoring. NetPlumber is able to not only detect loops and other invariant violations, but also check sophisticated policy's failures such as: "Web traffic from A to B should never pass through waypoints C or D between 9am and 5pm." Although Netplumber proposes a real-time method for detecting all typical violations, it ignores intra-rule dependencies in flow tables. Both HSA and Netplumber are time-consuming and thus are not suitable for networks with high rate of up and down links. Therefore, this is a major weakness for Netplumber which focuses on real-time environments. Mai et al. [13] tackle the misconfiguration problem by formal analysis of data plane state rather than diagnosing bugs in the control plane. This approach is able to not only detect the "invisible" bugs in routing configuration files, but also unifies the analysis regardless of the many implementations and protocols. The authors try to develop a tool to collect the network devices' Forwarding Information Bases (FIBs) and detect some typical failure by the Boolean functions. The tool is called "Anteater" and can check reachability and consistency of rules among the routers and loops in networks. It combines the data plane and invariants into instances of a Boolean satisfiability problem (SAT), and uses an SAT solver to perform analysis.

Al-Shaer and Al-Haj [9] present a configuration verification tool, which is called "FlowChecker," to validate, analyze and enforce at the run-time OpenFlow end-to-end configuration across multiple federations. It exploits FlowVisor [14], which partitions the network resources into smaller segments. FlowChecker is able to detect both intra-switch and inter-switch misconfiguration in a path of OpenFlow forwarding devices across the same or different infrastructure. It uses Binary Decision Diagram (BDD) to encode the flow tables. Afterwards, it tries to model the inter-connected OpenFlow switches' network via model checker techniques. The method is useful for verifying policy consistency. In addition, validating the configuration correctness in different switches and controllers across the distinct OpenFlow infrastructure also benefited from this tool. Furthermore, it is convenient for debugging reachability and predicting the impact of new policy on the network. Config-Checker [15] is a novel method that models the end-to-end behavior of access control configuration, including routers, IPsec, firewalls and NAT for Unicast and multicast packets. Config-Checker deals primary with security aspects in firewalls. The novelty of the method is the creation of symbolic model checker and its optimization. The model represents the network as a state machine defined by the packet header, and its location on the network hops. Packet header, packet location and the policy define the transitions in the state machine.

Sherwood et al. [14] present a logical isolation approach in one hardware switch, which is compatible with commodity switching chipsets and does not require the use of programmable hardware such as FPGAs or network processors. They develop a tool, which is called "FlowVisor." The tool uses the OpenFlow protocol for applying the policy isolation in the target network and is located between controller and forwarding devices. FlowVisor is a special purpose OpenFlow controller that acts as a transparent proxy between OpenFlow switches and multiple OpenFlow controllers. It prepares segments of network devices, controls them independently in separate logical controller, and guarantees the isolation. FlowVisor can create variant segments based on the combination of the forwarding devices or its ports, packet's address or packet's protocol [16]. However, it has a latency and overhead on the control channel due to use of an additional TLS connection. Son et al. devised a model checking system called "FloVer" [17], a formal approach to prove the conformance of dynamically produced OpenFlow flow rules against non-bypass security properties, including those with set and go to table actions. The authors demonstrate how to translate OpenFlow rules and network security policies into an assertion set, which can then be processed and verified by an SMT solver. This method uses the Yices SMT solver, which is integrated into NOX, a popular OpenFlow network controller. This system verifies that the aggregate of OpenFlow network's policies does not breach the network security and integrity of its policies.

III. Routing Prediction based on Intersection Method

Policies in the SDN-based network are changed frequently. Clarifying the side effects of new policies in a complicated network has always been vital for network administrators. Therefore, proposing an accurate approach for parsing the complex network by an input traffic is of utmost importance. In this Section, we will present a tracing method that is able to predict the route of both single and multiple input packets. The method is compatible with pipeline tables, group tables and required action as stated in OpenFlow 1.1.0 [12].

A. Tracing Function

The function "T" defines a recursive trace route process from a specific node for a single packet. In each iteration, the function detects which rule matches the input packet and detects the next hop consequently.

$$T(X,q):\begin{cases} T(A_{i_x},q) \quad if\,\exists C_{i_x}, A_{i_x}, C_{i_x}, A_{i_x} \in X \\ \quad \Bigg[\Big(\left(C_{i_x} \wedge q\right) \Leftrightarrow A_{i_x}\Big) \wedge \Bigg[\Big(\left(C_{i_x} \wedge q\right) \Leftrightarrow A_{i_x}\Big) \\ \quad \wedge \Big[\nexists C'_{j_x}.C'_{j_x} \in X \Big[(C'_{j_x} \wedge q) \wedge (C'_{j_x} \neq C_{i_x}) \\ \quad \wedge (j > i)\Big]\Big]\Bigg]\Bigg] \\ A_x \qquad if\, A_x = Client\ or\ Drop \end{cases} \tag{1}$$

X denotes a node, which is a set of rules, $X : \{R_{1_x}, R_{2_x}, \cdots\}$. Each rule contains a matching condition C and an action A, which refers to a next hop in the form of $R_{i_x} : (C_{i_x}, A_{i_x})$. The matching condition C includes the ingress_port and packet's header properties such as source IP, destination IP and destination port. q denotes a query representing a packet. The function T returns the next node as a result. The recursive process is terminated whenever the next node is a client or when the drop action is met. Therefore, via the tracing function, we can predict the destination of the input query. i and j are used to denote the rule's order in the flow table. Based on Equation 1, the Tracing_Function is developed and presented in Algorithm 1.

Algorithm 1: Tracing function

```
Input: Query, Starting_Hop
Output: All Hops in the route
Route ← Starting_Hop
Tracing_Function (Starting_Hop, Query, Route)
Rules ← Starting_Hop.Rules
Foreach rule in Rules
    If Query ∧ rule.Condition
        Route ← rule.Action
        If rule.Action = Client Or rule.Action = Drop
            Return Route
        End If
        Tracing_Function(rule.Action, Query, Route)
    End If
End For
```

B. Transfer Function

The function $T_{A\to B}(Q)$ proposes a packet transit process from a node A to node B via a precise input (Q). The matching process relies on the Raining 2D-Box model [18]. In this model, the input is checked sequentially against the higher priority rules. The unmatched part of input is checked further with the next rules.

$$T_{A\to B}(Q) : \forall q \in Q.T(A,q) = B \tag{2}$$

Equation 2 gives the formal definition of the transfer function. According to Fig. 1 and Equation 2, the results of the transfer function can be described as follows:

$$T_{A\to B}(q_1) : \{T_{B\to C}(q_3), T_{B\to D}(q_4, T_{B\to Drop}(q_8))\}$$
$$q_3 \cup q_4 \cup q_8 = q_1$$

$$T_{A\to C}(q_2) : \{T_{C\to E}(q_6), T_{C\to Drop}(q_9))\}$$
$$T_{B\to C}(q_3) : \{T_{C\to E}(q_5), T_{C\to Drop}(q_9))\}$$
$$q_6 \cup q_5 \cup q_9 = q_2 \cup q_3$$

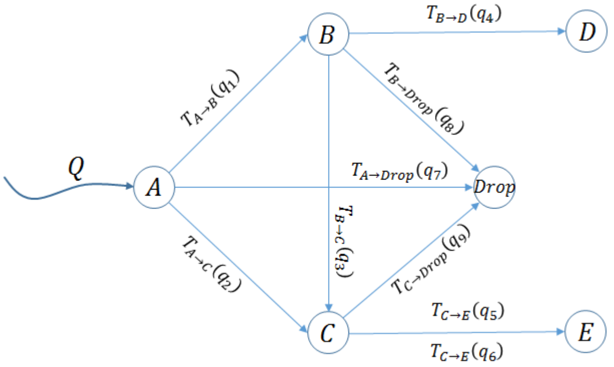

Fig. 1: A Sample directed graph and its transfer function

$$T(B,q_4) : \{D\},\ T(C,q_5) : \{E\}$$

Therefore, based on Equation 1 and Equation 2, $T(B,q_4)$ can be expressed recursively as follows:

$$T(B,q_4) \equiv T_{A\to D}(q_4) \equiv T_{A\to B}(T_{B\to D}(q_4))$$

As it can be clearly seen in Fig. 1, the node E is reached via two branches. The recursive expression of joining of these branches can be described as follows:

$$T(C,q_5) \cup T(C,q_6) \equiv$$
$$T_{A\to E}(q_5) \cup T_{A\to E}(q_6) \equiv$$
$$T_{A\to C}(T_{C\to E}(q_5)) \cup T_{A\to B}(T_{B\to C}(T_{C\to E}(q_6)))$$

In addition, by using the transfer function it is possible to check whether the path taken by the query passes through a specific node or not. As an example and according to Fig. 1, from the result of $T_{A\to D}(q_4)$, it is possible to check whether the query path meets node B or C. The result is shown as follows:

$$B \in T_{A\to D}(q_4),\ C \notin T_{A\to D}(q_4)$$

In order to check all possible routes from a source node via an input query, we shall use the Depth First Search (DFS) algorithm.

C. Reachability Checking

As aforementioned, reachability checking is one of the critical troubleshooting operations for network administrators in complex networks. According to the tracing function (Equation 1) and the transfer function (Equation 2), it is possible to check whether a precise host can connect to a specific host. The reachability checking method is defined based on the second-order logic (Equation 3). The predicate φ has been declared for checking the reachability of hop Y from hop X by a query Q. X and Y denote nodes, which are set of rules.

$$\varphi(X,Y,Q) : \exists q.q \in Q\ \Big[T_{X\to Y}(q)\Big] \tag{3}$$

Since the proposed method is an offline method, we assume that whenever we are faced with table miss packet[1], the tracing

[1] Whenever a table miss packet takes place, the packet gets forwarded to the controller in conformance with the Openflow protocol.

function returns as a final state the last hop where the table miss packet took place. The Reachability_Checking_Function has been developed based on Equation 2 and is described in greater detail in Algorithm 2. According to the algorithm, for each query, we detect the next hop.

For instance, in Fig. 2 a network with all routing tables is presented. According to Equation 3 and Algorithm 2, if the query Q is defined by “inPort=Port3, srcIP=*.*.*.*, dstIP=192.168.20.5, dstport=80”, the reachability predicate $\varphi(A, D, Q)$ returns true.

Algorithm 2: Reachability checking function

```
Input: Query, Starting_Hop, Destination_Hop
Output: Boolean Result
Reachability_Checking_Function (Starting_Hop, Destination_Hop,Query)
Foreach q in Query
    Route ← Starting_Hop
    Hops ← All hops in TRACING_FUNCTION (Starting_Hop, q, Route)
    If the last hop in Hops = Destination_Hop
        Return Route
    End If
End For
Return False
```

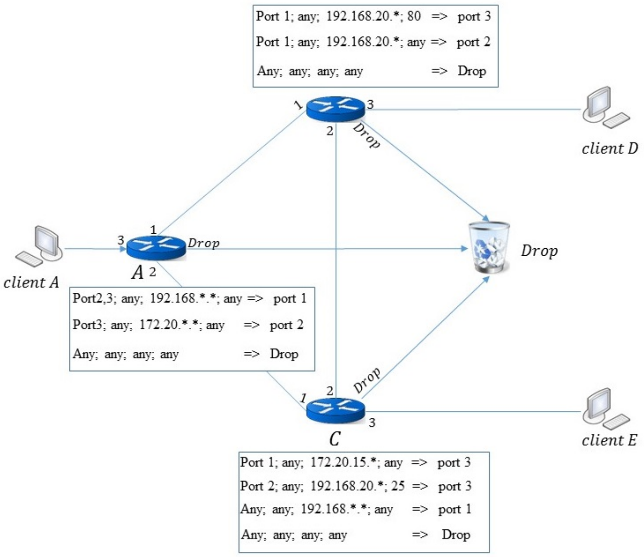
Port 1; any; 192.168.20.*; 80 => port 3
Port 1; any; 192.168.20.*; any => port 2
Any; any; any; any => Drop
1
3
2
Drop
client D
client A
3
1
Drop
A 2
Drop
Port2,3; any; 192.168.*.*; any => port 1
Port3; any; 172.20.*.*; any => port 2
Any; any; any; any => Drop
2
Drop
1
3
C
client E
Port 1; any; 172.20.15.*; any => port 3
Port 2; any; 192.168.20.*; 25 => port 3
Any; any; 192.168.*.*; any => port 1
Any; any; any; any => Drop

Fig. 2: A Sample network with routing tables

IV. Anomaly Detection on SDN-Switches

Based on the formal methods defined in Section III, in this Section, we present an offline anomaly detection approach. The rationale of the approach we propose is to generate queries that contain all possible packets that could pass through the network from all ingress switches. Subsequently, our anomaly detection algorithms are called for detecting possible policy conflicts.

A. Generating Queries

As mentioned previously, our anomaly detection method needs to check all possible packets that might enter the network via the ingress switches. By definition, ingress switches are gateways between the end clients and rest of the network. Fig. 3 sketches an example illustrating the ingress switch concept. In the ingress switch's flow table, the first rule is considered as one query. In order to generate the second query, we subtract the next rule from the previous rules (here the previous rules is merely the first rule). This process continues for the rest of the rules in a flow table of an ingress switch. The query generation operation is described in Equation 4. The queries, which are generated based on a specific flow table, are completely disjoint. Therefore, the queries can be executed in parallel without any specific order.

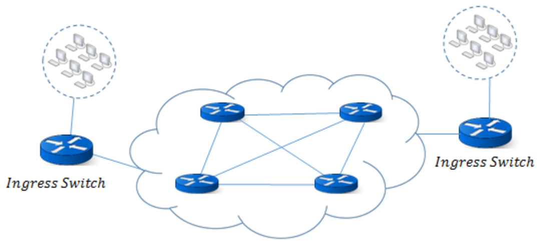

Fig. 3: Ingress switch in network

$$
\begin{aligned}
&Query_1 : Rule_1\\
&Query_2 : Rule_2 - Rule_1\\
&Query_3 : Rule_3 - (Rule_1 \cup Rule_2)\\
&\qquad\qquad \vdots\\
&Query_n : Rule_n - \left(\bigcup_{i=1}^{n-1} Rule_i\right)
\end{aligned}
\tag{4}
$$

Throughout calling transfer function, each rule in the whole network (whether or not it is in the ingress switches) will only be marked as a matched rule if it is matched with at least one query or subquery. At the end of the process, the unmatched rules are further investigated in order to discover the possible anomaly that caused the unmatch. The latter question will be addressed in the next subsection.

B. OpenFlow Rule Anomaly

There are several reasons that cause one rule to be never matched by all possible queries. Al-Shaer and Hamed [6] introduced four types of pairwise anomalies among rules in a firewall: shadowing, correlation, generalization and redundancy. Rezvani and Aryan [7] defines three more anomalies, namely, total shadowing, total generalization and total redundancy. Moreover, inter-anomalies, which might occur in distributed firewalls, have been defined by Al-Shaer and Hamed [19] and categorized as shadowing anomaly and redundant anomaly. Since the OpenFlow-based rules consist of two main parts, Conditions and Actions, the same categorization has been used for intra-anomalies among flow tables' rules. However, as explained by [20], flow tables' rules might have more than one action, i.e., multi-action. Thus, we shall propose a new expression for the OpenFlow rules' intra-anomalies that supports multi-action. To the best of our knowledge, such an aspect was not investigated in the literature before. Therefore, the unmatched rules can be a result of intra or inter-anomalies, which will be defined in greater detail in what follows.

1) Intra-Anomaly for single-action and multi-action:
An intra-anomaly takes place between rules in the same table. According to the [6] and [7], these types of anomalies are categorized into 7 groups. We shall use the bit wise format defined in [7] in order to re-write rules and packets. The formal specification of OpenFlow rule anomalies are put forward as follows.

a) Shadow Anomaly: If rule R_j matches all the packets that match rule R_i, $R_{i_{priority}} < R_{j_{priority}}$ and the two rules have different actions, R_i is shadowed by previous rule R_j. Formally, rule R_i is shadowed by rule R_j if the following condition holds:

$$\begin{gathered} R_{i_{priority}} < R_{j_{priority}} \\ R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable\,(C_i \Rightarrow C_j) \wedge (A_i \oplus A_j) \end{gathered} \tag{5}$$

As per Equation 5, rule R_i is shadowed by the rule R_j for the group of actions, which are true in $(A_i \oplus A_j)$.

b) Correlation Anomaly: Two rules in a flow table are correlated if they have different actions, and the first rule matches some packets that match the second rule and also the second rule matches some packets that match the first rule. Formally, rule R_i and R_j have a correlation anomaly if the following condition holds:

$$\begin{gathered} R_{i_{priority}} < R_{j_{priority}} \\ R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable \\ \Big[\neg (C_i \Rightarrow C_j) \wedge \neg (C_j \Rightarrow C_i) \wedge (C_i \wedge C_j)\Big] \\ \wedge (A_i \oplus A_j) \end{gathered} \tag{6}$$

As described by Equation 6, rule R_i and rule R_j have correlation for the group of actions that are true in $(A_i \oplus A_j)$.

c) Generalization Anomaly: Rule R_j is a generalization of a preceding Rule R_i if they have different actions, $R_{i_{priority}} < R_{j_{priority}}$ and if the rule R_i can match all the packets that match the rule R_j. Formally, rule R_i is generalization of rule R_j if the following condition holds:

$$\begin{gathered} R_{i_{priority}} < R_{j_{priority}} \\ R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable\,(C_j \Rightarrow C_i) \wedge (A_i \oplus A_j) \end{gathered} \tag{7}$$

According to Equation 7, rule R_i and rule R_j have generalization for the group of actions that which are true in $(A_i \oplus A_j)$.

d) Redundant Anomaly: Rule R_i is redundant to Rule R_j if they have same actions, and if the rule R_j can match all the packets that match the rule R_i. Formally, rule R_i is redundant to rule R_j if the following condition holds:

$$\begin{gathered} R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable\Big[(C_i \Rightarrow C_j) \vee (C_j \Rightarrow C_i)\Big] \\ \wedge (A_i \wedge A_j) \end{gathered} \tag{8}$$

As described by Equation 8, rule R_i and rule R_j have redundancy for the group of actions that are true in $(A_i \wedge A_j)$.

e) Total Shadow Anomaly: Rule R_i is totally shadowed by a set of previous rules if the previous rules match all the packets that match the rule R_i, and the rule R_i has different action from the previous rules. Formally, rule R_i is totally shadowed by rules $\{R_1 \cdots R_k\}$ if the following condition holds:

$$\begin{gathered} R_{i_{priority}} < R_{1_{priority}}, \cdots, R_{k_{priority}} \\ R_i : (C_i, A_i)\, R_1 : (C_1, A_1), \cdots R_k : (C_k, A_k) \\ \exists R_i, R_1, \cdots, R_k \in FlowTable\left(C_i \Rightarrow \left(\bigvee_{n=1}^{k} C_n\right)\right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k\right) \oplus A_i\right) \end{gathered} \tag{9}$$

According to the Equation 9, rule R_i and rules in the set: $\{R_1 \cdots R_k\}$ have total shadow for the group of actions that are true in $((\bigvee_{n=1}^{k} A_k) \oplus A_i)$.

f) Total Redundant Anomaly: Rule R_i is a total redundant of a set of rules if the set of rules match all the packets that match the rule R_i, and the rule R_i and the set of rules have the same action. Formally, rule R_i is a total redundant of a set of rules $\{R_1 \cdots R_k\}$ if the following condition holds:

$$\begin{gathered} R_{i_{priority}} < R_{1_{priority}}, \cdots, R_{k_{priority}} \\ R_i : (C_i, A_i), R_1 : (C_1, A_1), \cdots R_k : (C_k, A_k) \\ \exists R_i, R_1, \cdots, R_k \in FlowTable\left(C_i \Rightarrow \left(\bigvee_{n=1}^{k} C_n\right)\right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k\right) \wedge A_i\right) \end{gathered} \tag{10}$$

As per Equation 10, rule R_i and rules in the set: $\{R_1 \cdots R_k\}$ have total redundancy for the group of actions that are true in $((\bigvee_{n=1}^{k} A_k) \wedge A_i)$.

g) Total Generalization Anomaly: Rule R_i is a total generalization of a set of further rules if the rules match all the packets that match the rule R_i , and the rule R_i has different action from the rules. Formally, rule R_i is a total generalization of a set of rules $\{R_1 \cdots R_k\}$ if the following condition holds:

$$\begin{gathered} R_{i_{priority}} > R_{1_{priority}}, \cdots, R_{k_{priority}} \\ R_i : (C_i, A_i), R_1 : (C_1, A_1), \cdots R_k : (C_k, A_k) \\ \exists R_i, R_1, \cdots, R_k \in FlowTable\left(\left(\bigvee_{n=1}^{k} C_n\right) \Rightarrow C_i\right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k\right) \oplus A_i\right) \end{gathered} \tag{11}$$

As described by Equation 11, rule R_i and rules in the set: $\{R_1 \cdots R_k\}$ have total generalization for the group of actions that are true in $((\bigvee_{n=1}^{k} A_k) \oplus A_i)$.

2) Inter-Anomaly:
According to the nomenclature proposed in [19], at any point along the path of a given flow, a preceding switch is called an upstream hop whereas a following switch is called a downstream hop. Among two forwarding devices, when one

or more rules in upstream shadows the specific rule of a downstream hop matched by one or a group of the packet, an Inter-Anomaly takes a place. Note that in this section, we assume that the flow tables are intra-anomaly free. The Al-Shaer and Hamed [19] categorize the inter-anomalies in four groups. In contrast to [19], this paper defines four types of inter-anomalies in a different way, which are the root cause of unmatched rules.

a) Subset Rule Anomaly: A subset rule anomaly occurs if all packets that can be matched with the unmatched rule in a downstream hop, matches with an upstream hop's rule. Formally, rule R_i has a subset rule anomaly with rule R_j if the following conditions hold true:

$$\begin{gathered} R_i : \left(C_i, A_i\right), R_j : \left(C_j, A_j\right) \\ \exists R_i \in SW_i, R_j \in SW_j \quad Upstream\left(SW_j\right) \\ \wedge \left(C_i \Rightarrow C_j\right) \wedge \neg\varphi\left(SW_j, SW_i, C_i\right) \end{gathered} \tag{12}$$

In Equations 12-14, $Upstream()$ represents a predicate that returns true if the input hop is an upstream hop. φ is regarded as a predicate, which is described in Equation 3.

b) Superset Rule Anomaly: A superset rule anomaly occurs if all packets that matched with an upstream hop's rule, can be matched by an unmatched rule in a downstream hop. Formally, rule R_i has a superset rule anomaly with rule R_j if the following condition holds:

$$\begin{gathered} R_i : \left(C_i, A_i\right), R_j : \left(C_j, A_j\right) \\ \exists R_i \in SW_i, R_j \in SW_j \quad Upstream\left(SW_j\right) \\ \wedge \left(C_j \Rightarrow C_i\right) \wedge \neg\varphi\left(SW_j, SW_i, C_j\right) \end{gathered} \tag{13}$$

c) Partial Rule Anomaly: A partial rule anomaly occurs if just parts of packets, that can be matched with an unmatched rule in a downstream hop, are matched by an upstream hop's rule. Formally, rule R_i has a superset rule anomaly with rule R_j if the following condition holds:

$$\begin{gathered} R_i : \left(C_i, A_i\right), R_j : \left(C_j, A_j\right) \\ \exists R_i \in SW_i, R_j \in SW_j \quad Upstream\left(SW_j\right) \\ \wedge\neg\left(C_i \Rightarrow C_j\right) \wedge \neg\left(C_j \Rightarrow C_i\right) \wedge \left(C_i \wedge C_j\right) \\ \wedge\neg\varphi\left(SW_j, SW_i, \left(C_i \wedge C_j\right)\right) \end{gathered} \tag{14}$$

d) Irrelevant Rule Anomaly: The irrelevant rule anomaly occurs if all packets that can be matched with the unmatched rule are matched by different rules, and the paths for each packet are expected by the network administrator. Formally, rule R_i known as an irrelevant rule if the following condition holds:

$$\begin{gathered} R_i : \left(C_i, A_i\right), R_j : \left(C_j, A_j\right) \\ \forall sw \in ingress, \Bigg[\exists rule \in R\Big[\nexists packets, \left(packets \wedge C_{rule}\right) \\ \wedge\Big(T\left(sw, packet\right) \notin Ex\_Path\Big)\Big] \Leftrightarrow irrelevant\left(rule\right)\Bigg] \end{gathered} \tag{15}$$

$Ingress$ represents a set of all ingress switches in the network. R is regarded as a set of all unmatched rules. C_{rule} means rule's condition. Ex_Path refers to a set of expected paths, which are defined by the network administrator. $T()$ represents the transform function, which is described in Equation 2. Finally, $irrelevant()$ denotes a predicate, which returns true if the input is an irrelevant rule.

3) Invalid Rule Anomaly:
If the unmatched rule does not match with any subset of the input queries, it is considered as an invalid rule in the flow table. This anomaly is defined in Equation 16.

$$\forall q \in Q\Big[\exists r \in R\Big[\neg\left(q \wedge C_r\right)\Big] \Leftrightarrow invalid(r)\Big] \tag{16}$$

As expressed by Equation 16, R represents a set of unmatched rules and each member of this set is regarded as r, which is formed as a condition C_r and an action A_r. Q means a set of input queries. Finally, $invalid(r)$ amounts to a predicate, which returns true if the input rule r is recognized as an invalid rule.

C. Anomaly Detection

In the previous subsection, unmatched rules' anomalies are defined and categorized. In this subsection, we will describe the detection method. However, for sake of brevity and due to space limitation, the pseudo-code is not presented here and can be found in [21].

1) Invalid Rule Anomaly Detection:
The unmatched rules are checked based on Equation 16 to detect invalid anomalies. The result determines the rules that might never be matched by all possible queries. This type of anomaly usually occurs when a network administrator updates the network policy and forgets to remove part of the old rules from the same flow tables. For each unmatched rule, the algorithm tries to find a query that has an intersection with the unmatched rule in question. Whenever the function is unable find any intersecting query, the rule is moved to the invalid rules list.

2) Intra-Anomaly Detection:
The intra-anomaly detection operation is executed after the removal of the invalid rules from the unmatched rule list. The conflicting rules together with the anomaly types are reported to the network administrator for making a decision. According to Equations 5 to 7, the Simple_Anomaly_Detection_Function is defined. This function checks shadowing, generalization and correlation anomalies between the unmatched rule and each of its flow table's rules whenever the pair of rules have different actions. Moreover, according to Equation 8, the unmatched rule will be checked with the rules that have same action. The Total_Anomaly_Detection_Function is defined to detect the total anomalies based on Equations 9 to 11. In the first step, the algorithm collects the rules that have lower priority from the unmatched rule and have partial intersection with it. Then, total generalization anomaly condition (Equation 11) is checked for the part of collected rules that have different action than the unmatched rule. In addition, the total redundancy anomaly condition (Equation 10) is checked for the rest of the collection that has the same action as the unmatched rule. For the next part, rules with higher priority than the unmatched rule that partially overlap with it are collected. Then, total shadowing anomaly condition (Equation 9) is checked for

the part of collected rules that have different action from the unmatched rule. Subsequently, the rest of the rules, which have same action with the unmatched rule, are checked for the total redundancy anomaly condition (Equation 10).

3) Inter-Anomaly Detection:
The inter-anomaly detection operation assumes that the flow tables are intra-anomaly free. The conflicting rules together with the detected anomaly types will be reported to the network administrator for making a decision. According to the detection algorithm, for each unmatched rule, all paths from all ingress switches are calculated by the transfer function. Then, each path is compared with the expected path, which is specified by the network administrator. If both paths are the same, then no inter-anomaly is reported. Whenever the paths are different, the Check_Rule_InterAnomaly_Function will be called. Finally, according to Equation 15, the irrelevant rule checking will be performed. The unmatched rule is declared as an irrelevant rule provided that the rule's host does not exist in any administrator's expected path. The Check_Rule_InterAnomaly_Function aims to check and detect the anticipated anomalies between the unmatched rule and the rule that causes the conflict. Therefore, the rule that causes the difference between the query's path and the expected one is found. Then, the type of anomaly is further checked. The subset anomaly condition based on Equation 12 is checked between two rules. If it is not verified, the superset anomaly condition, which is represented by Equation 13, will be inspected. Finally, as per Equation 14, the partial anomaly condition will be checked.

V. Evaluation

In this Section, we evaluate the proposed method, which consists of four main phases: Test Query Generating, Probing Process, Intra-Anomaly Detection and Inter-Anomaly Detection. All methods are implemented in C++ and the experiments are run on a Ubuntu Virtual Machine, with only one core of the Intel core i3, 3.70 GHz CPU and 2GB of RAM.

A. Test Query Generating

In this subsection, the execution time of the test query generation process for different flow table sizes is evaluated. As illustrated by Table I and Fig. 4, the execution time increases dramatically whenever the rule set size exceeds 5000 rules.

Table I: Check rule inter-anomaly Function

Rule Number	500	1,000	2,000	5,000	10,000
Process Time (ms)	31,543	126,104	508,206	3.17×10^8	1.55×10^8

B. Probing Process

For evaluating the probing process, a query, which contains ingress port, source IP, destination IP and destination port, is used. Content of OpenFlow switch tables are fetched by the dump command [22], and then will be prepared by the parsing algorithm for the process. The parsing algorithm retrieves the table, in_port, nw_src, nw_dst, tp_dst and actions fields from the flow table. According to this procedure, with the predefined network's flow tables and the test queries, the execution time is evaluated. The flow tables do not have the pipeline tables and group tables. All the rules have the standard required action. The topology of the test scenario are shown in Fig. 5. As seen in Table II and Fig. 6, the execution time as a function of the number of rules has an almost linear growth.

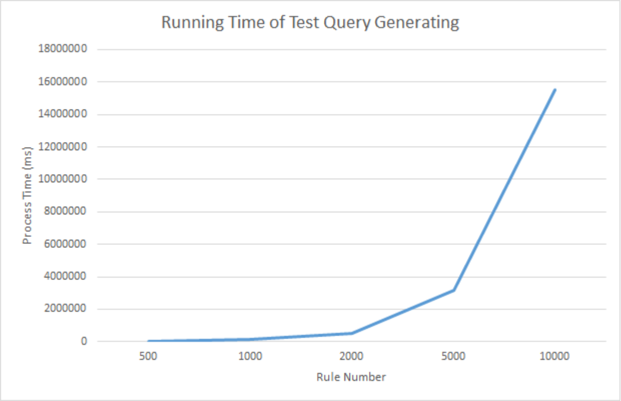

Fig. 4: Processing time results for test query generating process

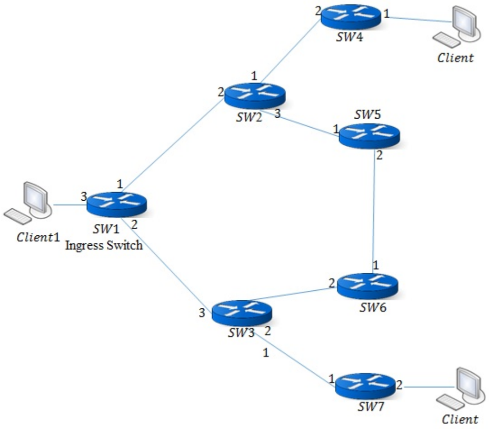

Fig. 5: Experiment's topology

Table II: Check rule inter-anomaly Function

Middle-box Rules	500	1,000	2,000	5,000	10,000
Total Number of Rules	3,500	7,000	14,000	35,000	70,000
Call Transfer Function	43,867	352,459	705,459	1.76×10^4	3.52×10^4
Process Time (ms)	2,151	20,109	40,696	100,829	201,701

C. Intra-Anomaly Detection

In this subsection, the execution time of the intra-anomaly detection is evaluated for different rule sets. By design, the execution time is independent of the type of the anomaly and numbers of detected anomalies. As observed in Table III and

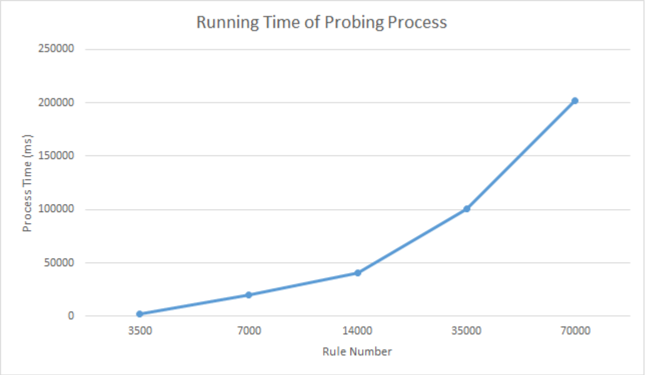

Fig. 6: Processing time results for probing process

Fig. 7, all anomaly detection algorithms have almost the same execution time for each set of rules. However, the execution time is increased dramatically for the set with 10,000 rules.

Table III: Processing time results for intra-anomaly detection

Flow Table Rule Number	500	1,000	2,000	5,000	10,000
Simple Shadow(ms)	4	7	13	25	53
Simple Correlation(ms)	2	10	17	31	68
Simple Generalization(ms)	2	7	12	26	70
Simple Redundancy(ms)	3	8	15	30	69
Total Shadow(ms)	4	7	15	28	72
Total Generalization(ms)	4	7	16	30	76
Total Redundancy(ms)	6	10	17	31	80

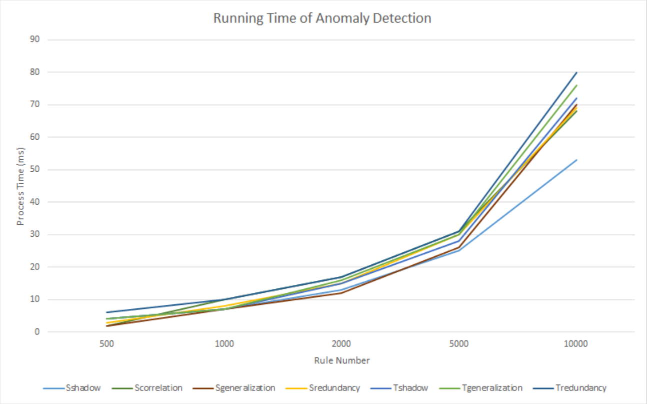

Fig. 7: Processing time results for intra-anomaly detection process

D. Inter-Anomaly Detection

As seen in Equation 12 to 15, the inter-anomaly detection resorts to the probing process. Therefore, the engine execution time follows the same pattern for a different rule number.

VI. Conclusion

SDN rules are usually updated in a dynamic manner, which leads to error prone policies. There have been some valuable studies on rule anomaly detection, however, those studies do not cope with multi-action Openflow rules. In this article, we provide comprehensive and generalized anomaly classification and detection methods for SDN that cover multi-action Openflow rules. Furthermore, we introduce the taxonomy: invalid and irrelevant anomalies for unmatched rules. An offline method is implemented based on the new definitions. The efficiency of our devised anomaly detection procedures are evaluated and the results are very promising. Our detection methods are embarrassingly parallel, which make them a viable solution for large network troubleshooting. As future work, we would like to quantify the gain in terms of execution time induced by the parallel nature of our algorithms.

References

[1] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "A survey on network troubleshooting," *Technical Report Stanford/TR12-HPNG-061012, Stanford University, Tech. Rep.*, 2012.

[2] R. Barrett, S. Haar, and R. Whitestone, "Routing snafu causes internet outage," *Interactive Week, April*, vol. 25, 1997.

[3] K. Butler, T. R. Farley, P. McDaniel, and J. Rexford, "A survey of BGP security issues and solutions," *Proc. IEEE*, vol. 98, no. 1, pp. 100–122, 2010.

[4] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford, "A slick control plane for network middleboxes," in *Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. Netw. - HotSDN '13*, 2013, p. 147.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69, 2008.

[6] E. Al-Shaer and H. Hamed, "Design and implementation of firewall policy advisor tools," *DePaul University, CTI, Tech. Rep*, 2002.

[7] M. Rezvani and R. Aryan, "Analyzing and resolving anomalies in firewall security policies based on propositional logic," *INMIC 2009 - 2009 IEEE 13th Int. Multitopic Conf.*, 2009.

[8] A. El-Atawy, K. Ibrahim, H. Hamed, and E. Al-Shaer, "Policy segmentation for intelligent firewall testing," *2005 First Work. Secur. Netw. Protoc. NPSec, held conjunction with ICNP 2005 13th IEEE Int. Conf. Netw. Protoc.*, vol. 2005, pp. 67–72, 2005.

[9] E. Al-Shaer and 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.

[10] A. Khurshid, W. Zhou, M. Caesar, P. B. Godfrey, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: verifying network-wide invariants in real time," *Present. as part 10th USENIX Symp. Networked Syst. Des. Implement. (NSDI 13)*, pp. 15–27, 2013.

[11] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," *Proc. 10th USENIX Conf. Networked Syst. Des. Implement.*, pp. 99–112, 2013.

[12] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FlowGuard: Building Robust Firewalls for Software-Defined Networks," *3th Work. Hot Top. Softw. Defin. Netw. (HotSDN 2014)*, pp. 97–102, 2014.

[13] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, S. T. King, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, p. 290, 2011.

[14] R. Sherwood, G. Gibb, K.-k. Yap, G. Appenzeller, M. Casado, N. Mckeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," *Network*, p. 15, 2009.

[15] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *17th IEEE International Conference on Network Protocols, 2009. ICNP 2009.* IEEE, 2009, pp. 123–132.

[16] "Flowvisor project webpage," https://openflow.stanford.edu/display/DOCS/Flowvisor, accessed: 2017-02-25.

[17] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *IEEE Int. Conf. Commun.*, 2013, pp. 1974–1979.

[18] T. Chomsiri and C. Pornavalai, “Firewall rules analysis.” in *Security and Management*. Citeseer, 2006, pp. 213–219.

[19] E. S. Al-Shaer and H. Hamed, “Discovery of policy anomalies in distributed firewalls,” in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4. IEEE, 2004, pp. 2605–2616.

[20] “OpenFlow official webpage,” https://www.opennetworking.org/sdn- resources/openflow, accessed: 2017-02-25.

[21] R. Aryan, A. Yazidi, P. Engesltad, and Ø. Kure, “Design and implementation of policy advisor for software defined networks,” *Unabridged Journal version of this paper (Under preparation).*

[22] “OVS Commands Reference version: 15, 2015,. 1032 elwell court, suite 105 palo alto, ca. 94303.”