An Incremental Approach for Swift OpenFlow Anomaly Detection

Ramtin Aryan*†, Anis Yazidi† and Paal Einar Engelstad*†
†Department of Computer Science, OsloMet – Oslo Metropolitan University
*Department of Technology Systems, University of Oslo
Oslo, Norway
Email:ramtina@ifi.uio.no, anisy@oslomet.no, paalen@oslomet.no

Abstract*—Software Defined Networking (SDN) is designed for dynamic policy update where frequent changes are pushed to the forwarding devices. Different offline approaches for detecting misconfiguration anomalies in SDN by taking a snapshot of the state of the network have been developed in the literature. However, the detection process is time-consuming and unfeasible in the case of frequent changes to the OpenFlow tables as well in big size networks containing a large number of rules. This paper presents an *incremental* method for detecting potential anomalies in an online manner, i.e., after one or multiple simultaneous updates in the SDN policy. Whenever the OpenFlow tables are dynamically changed, a *static* approach that rechecks the whole policy is unnecessarily redundant in a sense that most of the policy remains intact. Hence the need for *incremental* verification method to reduce this overhead, and only the subset of the policy that is affected by the update is checked. Two different solutions are proposed based on whether the policy modifications take place in the ingress switches or in the middle switches. We provide some comprehensive experiments to demonstrate the detection performance for the case of single or multiple simultaneous changes in forwarding devices. The experiment results show that the incremental method is drastically faster than the static parallel approach, with a factor up to about 450 times in some cases!

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

I. Introduction

The concept of SDN does not only enable the deployment of dynamic network policies, but also enables dynamic logical topology changes on the top of static network infrastructure. OpenFlow protocol has been designed to facilitate the connection between the controller and forwarding devices. OpenFlow rules have a timeout property that induces a time changing behavior inside the flow tables. This dynamic behavior can lead to several anomalies and unexpected network state. For instance, if some rules in some forwarding devices in a network get expired and removed from the flow tables automatically, the route or even destination of one or many flows could be changed. In addition, some high level policies such as dynamic routing or load balancing can cause group of changes in part of forwarding devices. On the top of that, intrusion detection and prevention system (IDPS) might perform some changes inside the part of forwarding devices as a response to possible attacks or malicious activities. It is laborious to predict the side effects of these sudden modifications. Static approaches for anomaly verification are inefficient in these cases as they require a total recheck of the whole policy. Various methods have been presented to deal with misconfiguration challenges. Some of the legacy offline approaches [1], [2], [3], [4] use formal logic to analyze the network policy snapshot without any inserting any probing packets. On the Other hand, different studies such as NetPlumber[2], Monocole [5], Rulescope [6] and VeriFlow[7] rather resort to packet probing and tracing. The paper tries to use the formal detection solution [1] for uncovering the possible anomalies after flow tables' alternation. The proposed method works differently based on where the policy updates have taken place, ingress switch or middle switch. If the rule has been changed in the ingress switch, the method tries to generate new queries while updating the affected queries. Then, an anomaly detection method is applied both on the new queries and on the affected ones. In the other case that the policy updates take place in middle switches, the queries remain unchanged while the detection method is called for specific queries that match with the switches affected by the policy update. Our approach is an incremental approach as the previous network state is used for checking the current network state. By using such an incremental approach, we are able to improve the verification procedure speed that is necessary in real-life SDN networks with a fair amount of rule updates. Moreover, our incremental approach supports parallelization. We catalogue the main contributions of this paper as follows:

- The proposed method is able to check and detect the possible inter and intra anomalies between flow tables' rules during the flow table updates.
- The detection technique is an incremental procedure that yields high performance and detection speed.
- The suggested process has the capability to run on the separated processors for preventing the overhead of process on the network operating system's resources (controller) and network performance;
- The detection method supports parallelization for checking each path in each iteration.
- The formal process is able to use the anomaly detection technique not only after one rule modification, but also for multiple rule alternations in different switches.

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 anomalies definition and detection. In Section IV, the incremental anomaly detection is 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. [2] who introduced Netplamber that is a real-time network policy checking tool. It tries to check the real-time network traffic in contrast to their previous method, HSA [2]. The proposed method uses a fast formal language to express policy checks for monitoring the real-time traffic. NetPlumber is able to not only detect loops and other invariant violations, but also check the certain node reachability from the specific source node. However, Netplumber ignores intra-rule dependencies. In addition, the both solutions are time-consuming and unsuitable for the SDN networks with a high rate of changes.

Aryan et al. [1] identify rules that never matched via a comprehensive formal detection method. They introduce a set of definitions for the intra-anomalies that might occur when using the OpenFlow rule's multi-action feature. Moreover, they present new definitions of inter-anomalies between OpenFlow switches reckoned as invalid and irrelevant anomalies. Their method is an offline approach that can be also parallelized.

RuleScope [6] categorized the forwarding faults in OpenFlow switches into missing faults and priority faults. Missing fault refers to the failure that happens when a rule is not active in the switch. Priority fault happens when the overlapping rules violate the expected policies. It suggests not only a detection method, but also a troubleshooting solution. The proposed troubleshooting procedure generates customized probing packets one at a time. For increasing the solution's performance, Wen et al. use a dependency graph that is compatible with parallel computation. Moreover, an incremental approach is introduced for improving the algorithm's speed.

Perešíni et al. [5] propose a method, which is called Monocol for detecting forwarding problems based on the hardware or software failures. Monocol is designed as a transparent layer between the controller and forwarding devices. It monitors data plane's rules and after inserting a new rule or rule modification, Monocol tries to generate relevant probe packet based on a Boolean satisfiability (SAT) algorithm and detects possible failures. SAT process is an NP-hard problem. However, they try to suggest an optimum solution for decreasing the computational cost to be practical for the dynamic changes in flow tables.

Anteater [8] tackles the misconfiguration problem by formal analysis. Anteater can check reachability, consistency of rules and loops. The tool can detect bugs in routing configuration file, which is called "Invisible Bugs" via a formal analysis of data plane. The detection process works by collecting the network devices' Forwarding Information Bases (FIBs) and detecting some typical failure by the Boolean functions.

FlowChecker [9] is a configuration verification tool that tries to validate, analyze and enforce at the run-time OpenFlow end-to-end configuration across multiple federations. FlowChecker can detect conflicts in both intra-switch and inter-switch scopes. By resorting to FlowChecker, FlowVisor [10] is able to verify policy consistency, validate configuration correctness in different switches and controllers across the specific SDN network.

Al-Shaer et al. [11] proposed a novel method for modeling the end-to-end behavior of access control configuration, including routers, IPsec, firewalls and NAT for Unicast and multicast packets. They implement a tool that is called Config-Checker based on the method. It tries to model the network as a state machine that the packet header, packet location and the policy represent the transitions in the sate machine.

FlowVisor is developed by Sherwood et al. [10] to apply policy isolation in the target SDN network. It acts as a transparent proxy between OpenFlow switches and controller. FlowVisor tries to create several isolated segments of network devices and control them by the logical controller. The tool can generate segments based on the combination of packets' addresses, packets' protocols, forwarding devices and its ports [12]. Son et al. [13] devised a formal approach to prove the conformance of dynamically produced OpenFlow rules against non-bypass security properties, including those with set and go to table actions. They proposed a model checking system that is called "FloVer." It tries to verify that OpenFlow policies do not lead to any security or integrity breaches in the network.

III. Anomaly Detection

OpenFlow rules have an expiration time that causes changes in the data plane behavior. In addition, existing network applications performing traditional functionality such as routing and load balancing might exhibit adaptive behavior as a response to changes in the network traffic pattern that leads to frequent policy updates to the flow tables. Considering this dynamic behavior and the large number of rules involved in a complex network topology, the prediction of a policy update side effect is necessary for hindering the emergence of misconfigurations. Policy updates can be divided into two main groups. The first group contains the flow tables of ingress switches that are modified, while the second group contains the modified flow tables of middle switches as depicted in Fig. 1.

Our legacy OpenFlow anomaly detection method presented

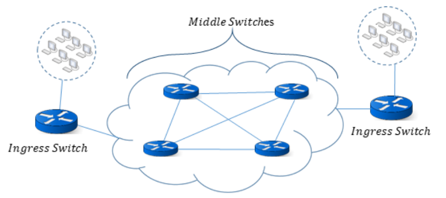


Fig. 1: Ingress Switch and Middle Switch in Network

in [1] suggests five steps: 1) generating queries, 2)detecting unmatched rules, 3) detecting the irrelevant rules, 4) detecting the intra-anomalies and 5) detecting the inter-anomalies. This paper uses these steps and propounds an incremental detection method. The suggested technique is used after each modification, whether inside ingress or middle switches. In following subsections, each detection step succeeding a policy updating is explained in detail.

A. Query Regeneration

Generating queries is an important step for predicting all possible routes of undertaken by the set of all possible packets [1]. According to our approach, the queries might be affected by the policies modification. Therefore, after each update, the queries should be checked and regenerated if necessary. In the following subsections, the query regeneration procedures for the ingress and middle switches after modification are explained.

1) Ingress Switches Modification:

Ingress switches have a main role in anomaly detection according to our formalism found in [1]. The queries are generated from the rules in ingress switches. Therefore, policy update in these switches has a complicated side effect on network policies, and the queries should be regenerated after modifications. However, generating the queries is a time-intensive process. Our idea is to regenerate only the queries that are affected by the new changes. We have two different query regeneration processes for adding a new rule and removing a rule from the flow table.

a) Inserting a New Rule: Since the queries are totally disjointed [1], the queries, which are generated via rules with higher priorities than the new rule are not affected. Hence, new computation should be done for the new rule and the rules with a lower priority. As an example, in Table I, a new rule is added with priority 2. Thus rule 1 is not affected, and its query will remain unchanged, while rules number 3 and 4 should be considered for regenerating the queries. Our next optimization idea consists of only regenerating the queries that have overlap with the new rule. In Table I, rule 3 has no overlap with the new rule (rule 2). Therefore, the query that is generated based on rule 3 should be kept intact. However, rule 4 has overlap with the new rule, and its query should be regenerated.

TABLE I: Modified Flow Table

Priority	Src_IP	Dst_Port	Action
1	10.0.0.1	25,80	Port4
2	**10.0.0.1**	**53**	**Port2**
3	10.0.0.*	80	Port3
4	10.0.0.*	53,23	Drop

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

Finally, our third optimization idea is even in the case of generating a new query, the new query can be shown to be a subset of the old query and therefore it is not necessary to redo all the computations. Queries are generated by the Equation 1 [1]. For instance, Equation 2 shows the optimum computation for the query 4. Query 2 should be generated completely since it related to the new rule (rule 2), and queries 1 and 3 are kept as before since they are not affected by the new rule.

$$\begin{array}{l} \mathbf{Query_2} : \underline{\mathbf{Rule_2}} - Rule_1 \\ \mathbf{Query_4} : Rule_4 - \big(Rule_1 \cup Rule_3 \cup \underline{\mathbf{Rule_2}}\big) \\ \quad \Rightarrow Rule_4 \cap \big(Rule_1 \cup Rule_3 \cup \underline{\mathbf{Rule_2}}\big)' \\ \quad \Rightarrow Rule_4 \cap \Big(Rule_1' \cap Rule_3' \cap \underline{\mathbf{Rule_2'}}\Big) \\ \quad \Rightarrow \Big(Rule_4 \cap Rule_1'\Big) \cap \Big(Rule_4 \cap Rule_3'\Big) \\ \qquad \cap \Big(Rule_4 \cap \underline{\mathbf{Rule_2'}}\Big) \\ \quad \Rightarrow (Rule_4 - Rule_1) \cap (Rule_4 - Rule_3) \\ \qquad \cap \big(Rule_4 - \underline{\mathbf{Rule_2}}\big) \end{array} \tag{2}$$

As a conclusion, the query regeneration process for the Table I can be represented as Equation 3. Please note that according to Equation 3, $Query\_new_4$ is a subset of $Query_4$.

$$\begin{array}{l} Query_1 : not\ changed \\ \mathbf{Query\_new_2} : \underline{\mathbf{Rule_2}} - Rule_1 \\ Query_3 : not\ changed \\ \mathbf{Query\_new_4} : Query_4 \cap \big(Rule_4 - \underline{\mathbf{Rule_2}}\big) \end{array} \tag{3}$$

b) Removing a Rule: Since queries are totally disjointed, the queries, which are generated via rules with higher priorities than the removed rule are not affected. Hence, new computation should be performed for the removed rule and the rules with a lower priority. As an example, in Table I, if rule 2 is removed from flow table, rule 1 is not affected, and its query will be unchanged. Rules number 3 and 4 should be checked for regenerating the queries. According to the next optimization idea, the query regeneration process should try to regenerate the queries that have overlap with the new rule and not the queries that are totally disjointed with the new rule. In Table I, rule 3 has no overlap with the removed rule (rule 2). Therefore, the query that is made based on the rule 3 should not be regenerated. However, rule 4 has overlap with the removed rule, and its query should be regenerated. The last proposed optimization is applied in the regeneration algorithm. Queries are generated by the Equation 1. For the queries that should be regenerated, we can save computation by just subtracting the removed rules from the old queries. For instance, Equation 4 shows an optimized computation for query 4. Query 2 should be removed completely, and queries 1 and 3 are kept intact since they are not affected by the removed rule.

$$\begin{array}{l} \mathbf{Query_2 : removed} \\ \mathbf{Query\_new_4} : \Big((Rule_4 - Rule_1) \cap (Rule_4 - Rule_2) \\ \qquad \cap (Rule_4 - Rule_3) \Big) \\ \qquad \cup \big(Rule_4 \cap \underline{\mathbf{Rule_2}}\big) \\ \quad \Rightarrow Query_4 \cup \big(Rule_4 \cap \underline{\mathbf{Rule_2}}\big) \end{array} \tag{4}$$

The query regeneration process for the Table I can be summarized by Equation 5.

$$\begin{array}{l} Query_1 : not\ changed \\ \mathbf{Query_2 : removed} \\ Query_3 : not\ changed \\ \mathbf{Query\_new_4} : Query_4 \cup \big(Rule_4 \cap \underline{\mathbf{Rule_2}}\big) \end{array} \tag{5}$$

2) Middle Switches Modification:
In the case of modification in the middle switches, the queries that generated based on the ingress switches are not affected, and thus should not be regenerated. Therefore, this case is less complex than the case of policy update in ingress switches.

B. Unmatched Rule Detection

Inserting new rules or removing the existing rules may cause new unmatched rules or lead to remove some rules from the unmatched list. Hence, after updating the queries, the predicting function should be applied again to predict all possible routes for all queries. This is a general process, whether the rule updates are happened in the ingress or middle switches. Accordingly, "Tracing Function" and "Transfer Function" are used that are explained in detail as follows.

1) Tracing Function:
Tracing function "T", defines a recursive tracing process from a specific node via a single packet. In each iteration, the function detects which rule matches the input packet and detects the next hop consequently. Equation 6 represents a formal expression of the tracing function.

$$T(X,q) : \begin{cases} T(A_{i_x}, q) \quad if \exists C_{i_x}, A_{i_x}, C_{i_x}, A_{i_x} \in X \\ \quad \Bigg[\Big((C_{i_x} \wedge q) \Leftrightarrow A_{i_x} \Big) \wedge \Big[\nexists C'_{j_x} . C'_{j_x} \in X \\ \quad \Big[(C'_{j_x} \wedge q) \wedge (C'_{j_x} \neq C_{i_x}) \wedge (j > i) \Big] \Big] \Bigg] \\ A_x \qquad if\, A_x = Client\ or\ Drop \end{cases} \tag{6}$$

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.

2) Transfer Function:
Transfer Function $T_{A \to B}(Q)$, propounds the packets' transmission process from node A to node B via a precise input query (Q). According to this model, the input query is checked sequentially against the higher priority rules. The unmatched part of input is checked further with the next rules. Equation 7 shows the formal definition of transfer function, and as it expresses, transfer function is based on the tracing function.

$$T_{A \to B}(Q) : \forall q \in Q . T(A, q) = B \tag{7}$$

Therefore, if transfer function called for all the queries that are generated based on the ingress switches' flow tables, then the unmatched rules will be found. In the next steps, it will be found out which type of anomaly leads to one rule never be matched with any packets.

C. Irrelevant Anomaly Detection

If the unmatched rule does not match with any subset of the specific queries, it is considered as an invalid rule in the flow table. 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.This anomaly is defined formally in Equation 8.

$$\forall q \in Q \Big[\exists r \in R \Big[\neg (q \wedge C_r) \Big] \Leftrightarrow invalid(r) \Big] \tag{8}$$

D. Intra-Anomaly Detection

An intra-anomaly takes place between rules of a switch's flow tables. According to the [4] and [3], Ramtin et al. [1] suggest a formal specification for the OpenFlow rules' intra-anomalies that supports multi-action rule's format. These intra-anomalies are categorized in seven groups that are described as follows.

1) 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:

$$\begin{array}{c} R_{i_{priority}} < R_{j_{priority}} \\ R_i : (C_i, A_i)\,, R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable\, (C_i \Rightarrow C_j) \wedge (A_i \oplus A_j) \end{array} \tag{9}$$

As per Equation 9, rule R_i is shadowed by the rule R_j for the group of actions that are true in $(A_i \oplus A_j)$.

2) 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:

$$\begin{array}{c} R_{i_{priority}} < R_{j_{priority}} \\ R_i : (C_i, A_i)\,, R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable \\ \Big[\neg (C_i \Rightarrow C_j) \wedge \neg (C_j \Rightarrow C_i) \wedge (C_i \wedge C_j) \Big] \\ \wedge (A_i \oplus A_j) \end{array} \tag{10}$$

As described by Equation 10, rule R_i and rule R_j have correlation for the group of actions that are true in $(A_i \oplus A_j)$.

3) 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:

$$\begin{gathered} R_{i_{priority}} < R_{j_{priority}} \\ R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable\,(C_j \Rightarrow C_i) \wedge (A_i \oplus A_j) \end{gathered} \tag{11}$$

According to Equation 11, rule R_i and rule R_j have generalization for the group of actions that are true in $(A_i \oplus A_j)$.

4) 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:

$$\begin{gathered} R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i, R_j \in FlowTable\Big[(C_i \Rightarrow C_j) \vee (C_j \Rightarrow C_i)\Big] \\ \wedge (A_i \wedge A_j) \end{gathered} \tag{12}$$

As described by Equation 12, rule R_i and rule R_j have redundancy for the group of actions that are true in $(A_i \wedge A_j)$.

5) 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:

$$\begin{gathered} R_{i_{priority}} < R_{1_{priority}}, \cdots, R_{k_{priority}} \\ R_i : (C_i, A_i)\; R_1 : (C_1, A_1), \cdots R_k : (C_k, A_k) \\ \exists R_i, R_1, \cdots, R_k \in FlowTable\left(C_i \Rightarrow \left(\bigvee_{n=1}^{k} C_n\right)\right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k\right) \oplus A_i\right) \end{gathered} \tag{13}$$

According to the Equation 13, 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 $((\bigvee_{n=1}^{k} A_k) \oplus A_i)$.

6) 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:

$$\begin{gathered} R_{i_{priority}} < R_{1_{priority}}, \cdots, R_{k_{priority}} \\ R_i : (C_i, A_i), R_1 : (C_1, A_1), \cdots R_k : (C_k, A_k) \\ \exists R_i, R_1, \cdots, R_k \in FlowTable\left(C_i \Rightarrow \left(\bigvee_{n=1}^{k} C_n\right)\right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k\right) \wedge A_i\right) \end{gathered} \tag{14}$$

As per Equation 14, 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 $((\bigvee_{n=1}^{k} A_k) \wedge A_i)$.

7) 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:

$$\begin{gathered} R_{i_{priority}} > R_{1_{priority}}, \cdots, R_{k_{priority}} \\ R_i : (C_i, A_i), R_1 : (C_1, A_1), \cdots R_k : (C_k, A_k) \\ \exists R_i, R_1, \cdots, R_k \in FlowTable\left(\left(\bigvee_{n=1}^{k} C_n\right) \Rightarrow C_i\right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k\right) \oplus A_i\right) \end{gathered} \tag{15}$$

As described by Equation 15, 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 $((\bigvee_{n=1}^{k} A_k) \oplus A_i)$.

The intra-anomaly detection procedure is applied after the invalid rules detaching. Then, each remained unmatched rule will be checked with the other rules in the same flow tables to find the possible intra-anomalies, whether single or total anomalies.

E. Inter-Anomaly Detection

An inter-anomaly occurs between policies of two different switches. It is assumed that the flow tables in this step, do not have any intra-anomalies. Al-Shaer and Hamed [14] categorize the inter-anomalies in four groups. In contrast to [14], Ramtin et al. [1] define four types of inter-anomalies for OpenFlow based rules that are the root cause of unmatched rules. They are defined as follows.

1) 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:

$$\begin{gathered} R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i \in SW_i, R_j \in SW_j \quad Upstream\,(SW_j) \\ \wedge (C_i \Rightarrow C_j) \wedge \neg\varphi\,(SW_j, SW_i, C_i) \end{gathered} \tag{16}$$

In Equations 16-18, $Upstream()$ represents a predicate that returns true if the input hop is an upstream hop. φ is regarded as a predicate that checkes the reachability of $switch_j$ from $switch_i$ by a specific query [1].

a) 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:

$$\begin{gathered} R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i \in SW_i, R_j \in SW_j \quad Upstream\,(SW_j) \\ \wedge (C_j \Rightarrow C_i) \wedge \neg\varphi\,(SW_j, SW_i, C_j) \end{gathered} \tag{17}$$

2) 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:

$$\begin{gathered} R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \exists R_i \in SW_i, R_j \in SW_j \quad Upstream\,(SW_j) \\ \wedge \neg\,(C_i \Rightarrow C_j) \wedge \neg\,(C_j \Rightarrow C_i) \wedge (C_i \wedge C_j) \\ \wedge \neg \varphi\,(SW_j, SW_i, (C_i \wedge C_j)) \end{gathered} \tag{18}$$

3) 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:

$$\begin{gathered} R_i : (C_i, A_i), R_j : (C_j, A_j) \\ \forall sw \in ingress, \Big[\exists rule \in R \Big[\nexists packets, (packets \wedge C_{rule}) \\ \wedge \Big(T\,(sw, packet) \notin Ex\_Path \Big) \Big] \Leftrightarrow irrelevant\,(rule) \Big] \end{gathered} \tag{19}$$

$Ingress$ 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 that are defined by the network administrator. $T()$ represents the transform function that is described in Equation 6. Finally, $irrelevant()$ denotes a predicate that returns true if the input is an irrelevant rule.
According to the detection algorithm [1], 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 inter-anomaly detection method will be called to check the type of anomaly.

IV. Incremental Anomaly Detection

As explained in Section III, the anomaly detection process is time-intensive. It is quite clear that running the detection method from scratch after each policy modification is not an effective solution. Thus, an incremental approach has been adopted to avoid the redundant process when it comes to the unchanged queries. The detection process for a group of policies' updates in a specific time window is called "one iteration." The length of time window can be defined based on the network's behavior and the rate of policies update. A short time window for networks with a high rate of policy's replacement can improve the performance. However, a short window for low rate of changes might have a negative effect due to the process overhead. In the incremental approach, results of previous step are used for the unchanged part of the network's policies in the next iteration. The incremental design is divided into two routines for two types of policy's alternation. The first detection routine is used for anomaly detection after the policy update in ingress switches. The second routine is an incremental solution for rule updating in the middle switches that checks for the eventual anomalies that might occur after the updates. These approaches are described in detail as follows.

A. incremental Anomaly Detection in Ingress Switches

Subsequent to regenerating the affected queries, the anomaly detection method should be used for uncovering the unwanted side effects after the policy's modification in ingress switches. However, running the detection methods is computationally heavy. On that account, the detection phase after policy modification in the ingress switches should be optimized. Therefore, the routing prediction algorithm is used just for the new queries. The result will be merged with the result of unchanged queries in the previous iteration, and a new unmatched rule list will be generated. Finally, the anomaly detection method, which is explained in Section III is used for detecting possible anomalies. Algorithm 1 shows the incremental query regenerating and Algorithm 2 describes the incremental probing technique after a policy update in ingress switch.

Algorithm 1: Incremental Query Regeneration

```
Output: $NewQueryList$
foreach $newRule$ do
    $newQuery \leftarrow generateQuery(newRule)$;
    $QueryList.insert(newQuery,\ newRule.Index)$;
    foreach $Index > newRule.Index$ do
        $QueryList[Index] \leftarrow$
          $QueryList[Index] \cap (RuleList[Index] - newRule)$;
    end
end
foreach $removedRule$ do
    foreach $Index > removedRule.Index$ do
        $QueryList[Index] \leftarrow$
          $QueryList[Index] \cup (RuleList[Index] \cap removedRule)$;
    end
    $QueryList.remove(removedRule.Index)$;
end
```

B. incremental Anomaly Detection in Middle Switches

After policy's updates in the middle switches, the anomaly detection method should be applied, and as mentioned previously this process in time-intensive. For the sake of performance, only the queries that are related to the affected switches, are checked again. These queries can be processed from the ingress switches; however, this process might be time-consuming. In order to achieve a further performance gain, the Tracing function [1] is applied from the modified switch instead of the ingress switch. At the end, the new result of the Transfer function will be merged with the results of unchanged queries from the preceding iteration. By means of merging process, the new unmatched rule list will be generated. The new unmatched rules are checked by the anomaly detection method. Algorithm 3 presents an incremental algorithm for flow tables in middle switches after the rule update.

Algorithm 2: Incremental Anomaly Detection in Ingress Switch

```
   Input: $expath$ : $administrator\ expected\ path$
   Output: $report\ the\ rules\ and\ anomalies$
1  $unmatched\_rulelist \leftarrow \emptyset$;
2  foreach $ingress\_switch\ in\ ingress\_switches$ do
3      foreach $update\ in\ ingress\_switch$ do
4          $new\_queries \leftarrow query\_regeneration()$;
5          foreach $query\ in\ new\_queries$ do
6              $hops, routes \leftarrow$
                 Tracing_Function($ingress\_switch, query$);
7              $unmatched\_rulelist \leftarrow$
                 Update_UnmatchedRules($unmatched\_rulelist$,
                 $hops, routes$);
8              Find_InvalidRules($query, unmatched\_ rulelist$);
9              Find_IntraAnomaly($unmatched\_rulelist$);
10             Find_InterAnomaly($routes, expath$,
                 $unmatched\_rulelist$);
11         end
12     end
13 end
```

Algorithm 3: Incremental Anomaly Detection in Middle Switch

```
   Input: $expath$ : $administrator\ expected\ path$
          $paths$ : $routes\ before\ updates$
   Output: $report\ the\ rules\ and\ anomalies$
1  $unmatched\_rulelist \leftarrow \emptyset$;
2  foreach $update\ in\ middle\_switch$ do
3      foreach $hop\ in\ updated\_hops$ do
4          $queries \leftarrow$ get_queries_passfrom($hop, paths$);
5          foreach $query\ in\ queries$ do
6              $hops, routes \leftarrow$ Tracing_Function($hop, query$);
7              $unmatched\_rulelist \leftarrow$
                 Update_UnmatchedRules($unmatched\_rulelist$,
                 $hops, routes$);
8              Find_InvalidRules($query, unmatched\_ rulelist$);
9              Find_IntraAnomaly($unmatched\_rulelist$);
10             Find_InterAnomaly($routes, expath$,
                 $unmatched\_rulelist$);
11         end
12     end
13 end
```

V. Evaluation

In this section, we evaluate the performance and efficiency of the proposed method. The main purpose of devised method is to speed-up the anomaly detection process. Therefore, we will compare the execution time between of the static parallel and incremental approach. The algorithm is implemented in C++. The experiments run on the server with 48 Intel(R) Xeon(R) CPU 2.30GHz. In order to generate realistic OpenFlow rules, we use the Class Bench [15] tool. Each rule contains five main fields that include, Ingress Port, Source IP, Destination IP, Destination Port and Action. The method is evaluated with five datasets with seven middle-boxes. The middle-boxes in our datasets contain 500, 1000, 2000, 5000 and 10000 rules respectively. The experiment has three main phases, Query Generation, Probing Process, Anomaly Detection. The algorithm is run 30 times for each distinctive dataset. We report the average execution time with the confidence interval 95%. The results are represented in following subsections.

A. Query Generation Evaluation

According to the [1], the most time-intensive part of the algorithm is Query Generating. Algorithm 1 presents an incremental approach for increasing the execution speed. The execution time for different number of rules is shown in Table II. The Fig. 3(a) shows the comparison between the execution time of static parallel query generation and incremental approach for regenerating queries after inserting a new rule.

TABLE II: Query Regenerating Executing Time

Number of Rule	Execution Time (ms) Average	CI 95%
500	54.73	14.12
1,000	111.36	28.26
2,000	400.36	109.73
5,000	597.34	161.71
10,000	1,182.31	322.61

B. Probing Evaluation

In compliance with the Algorithm 2 and 3, the probing process should be run for each regenerated query and the modified one. On the other hand, if the new rule is added to the middle switch, the queries that pass through the updated switch should be checked again by the probing process. We choose some specific queries that yields the worse case where we guarantee that all switches of the topology in Fig. 2 will be checked by the algorithm. The execution time for different number of rules is shown in Table III.

TABLE III: incremental Probing Executing Time

Number of Rule	Execution Time (ms) Average	CI 95%
3500	0.043	0.0068
7,000	0.045	0.0045
14,000	0.058	0.0056
35,000	0.062	0.0039
70,000	0.061	0.0038

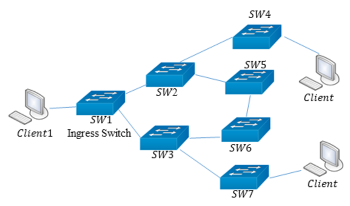
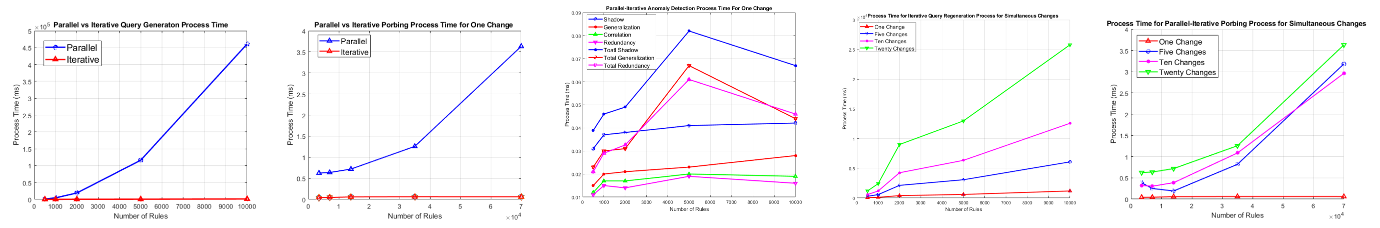


Fig. 2: Test Topology

The Fig. 3(b) shows the comparison between the execution time of static parallel probing process for 23 distinct queries and incremental probing approach after inserting a new rule.

C. Anomaly Detection Evaluation

The correctness of the Anomaly Detection algorithm is confirmed based on our generalized formalism reported in [1]. In order to assess the accuracy of our algorithm in terms of false positive and false negative, we test the algorithm with the twelve group of marked rules where each one of them represents a specific type of anomaly. As expected, there is no



(a) Incremental vs Static Parallel Query Generation Execution Time (b) Incremental vs Static Parallel Probing Execution Time (c) Incremental Anomaly Detection Execution Time (d) Incremental Query Regeneration Execution Time For Simultaneous Changes (e) Incremental Probing Execution Time For Simultaneous Changes

Fig. 3: Experiment Results

false negative or false positive in the implementation's output. Please note that the fact that our algorithm does not results into false positives or false negatives can be easily proven formally. The proof is omitted for the sake of brevity. The performance of the anomaly detection algorithm is evaluated by checking the eventual anomalies when inserting a new rule in flow tables with varying rule size. The superset, subset and partial anomaly detection process are the same as for the cases of simple generalization, simple shadow and simple correlation anomaly detection respectively. The experiment's results are presented in Fig. 3(c). Moreover, invalid and irrelevant anomaly detection procedures use the query generation and probing processes, which are presented in the previous subsections.

D. Simultaneous Changes

As mentioned in Section IV, the incremental algorithm is called for each specific time window. So, all rules' modifications that take place during the time window are considered as simultaneous changes. The window length might have a direct effect on the algorithm performance. The main parameter for defining the length of the time window is the number of modifications. Therefore, for evaluating the algorithm performance in different time windows, we design experiments with a different number of simultaneous changes. The results are presented in Fig. 3(d) and Fig. 3(e).

VI. Conclusion

SDN rules are usually updated continuously causing possible misconfiguration errors. Although several anomaly detection approaches exist in the literature, most of them are static and does not handle the case of frequent policy updates. In this paper, we propose a comprehensive incremental method to detect all potential anomalies subsequent to an update in the data plane. The incremental approach helps us to significantly improve the detection speed, which is very important for having an appropriate response to the frequent policy updates in SDN networks. The experiment results are very promising and show that this method can be used as an advisor to assist network administrators in order to maintain an anomaly free state. As a future work, we would like to develop a policy revise that can be used as a part of a real-time network troubleshooting toolbox.

References

[1] R. Aryan, A. Yazidi, P. E. Engelstad, and 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.

[2] 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.

[3] 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.

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

[5] P. Perešíni, M. Kuźniar, and 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.

[6] X. Wen, K. Bu, B. Yang, Y. Chen, L. E. Li, X. Chen, J. Yang, and X. Leng, "Rulescope: Inspecting forwarding faults for software-defined networking," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 4, pp. 2347–2360, 2017.

[7] 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.

[8] 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.

[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] 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.

[11] 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.

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

[13] 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.

[14] E. S. Al-Shaer and H. 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.

[15] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Transactions on Networking (TON)*, vol. 15, no. 3, pp. 499–511, 2007.