Received: 29 December 2018 | Revised: 27 June 2019 | Accepted: 13 May 2020

DOI: 10.1002/cpe.5907

WILEY

SPECIAL ISSUE PAPER

A parallel approach for detecting OpenFlow rule anomalies based on a general formalism

Ramtin Aryan[1,2] | Anis Yazidi[2] | Øivind Kure[1] | Paal Einar Engelstad[1,2]

[1]Department of Technology Systems, University of Oslo, Oslo, Norway

[2]Department of Computer Science, OsloMet – Oslo Metropolitan University, Oslo, Norway

Correspondence
Ramtin Aryan, Department of Technology Systems, University of Oslo, Oslo, Norway.
Email: Ramtina@ifi.uio.no

Summary

As the policies of a software-defined networking (SDN) network can be updated dynamically and often at a high pace, conflicts between policies can easily 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 article presents three main contributions. First, we devise an offline method for detecting unmatched OpenFlow rules, that is, rules that are never fired. In our taxonomy such anomalies can stem from either *invalid* or *irrelevant* unmatched rules. Second, we introduce a new set of definitions for the intraanomalies between rules in the same table, which might occur when using the *multiaction* feature of an OpenFlow rule. Third, our detection method has been enhanced to support parallel execution, which makes it a viable solution for troubleshooting large-scale networks. We provide some comprehensive experimental results based on both synthetic and real-life setup the synthetic set up is designed in such a way that the rule matching takes place in the last rules of the switch and thus putting more stress on the rule detection process. The parallel method is shown to outperform the single-threaded checking method by order of magnitude up to 21.

KEYWORDS

anomaly definition, anomaly detection, multithread, OpenFlow, parallelization, software-defined network, unmatched rules

1 | INTRODUCTION

The software-defined networking (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."[2] As the SDN controller is controlling the state of the network, it is possible to analyze network misconfigurations in a centralized manner. The OpenFlow protocol allows the controller to write rules directly to forwarding devices. In addition, SDN allows numerous applications and even multiple users to program the same physical network simultaneously.[3] Although SDN presents an alternative for facilitating network control by providing the ability for the network administrators to program the network data plane, the risk of misconfiguration still constitutes an omnipresent problem.

Various methods have been presented to deal with misconfiguration challenges. These methods fall under two main categories: log-based vs rule-based verification. Log-based methods detect policy violations based on mining the logs of the network devices.[4,5] Rule-based verification, on

A preliminary version of this article was presented at LCN 17, the 42th International Conference on Local Computer Networks, Singapore, in October 2017 [1].

the other hand, reveals policy violations by examining the semantics of the rules. 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] used propositional logic to detect policy violations between a new inserted rule and the combination of existing rules.

Research can also be categorized based on whether it mainly focuses on misconfigurations on a single device or between devices. We refer to this as intradevice anomalies and interdevice anomalies, respectively, or simply *intraanomalies* and *interanomalies* for short. For example, the works of Al-Shaer and Hamed[6] and Rezvani and Aryan[7] take an intradevice approach and focus on rule anomalies within a firewall (*intraanomalies*). On the other hand, VeriFlow[8] and the work of Kazemian et al[9] both belong to the intradevice category, as they are able to check policy violations among connected devices in a network (*interanomalies*).

In this article, we present a rule-based approach for detecting misconfigurations in SDN, which applies to both intra- and inter-device settings. It supports not only original single-action OpenFlow rules (ie, One action per rule), it also supports multiaction rules, that is, Rules that have more than one action.

We devise an offline method that is able to predict the action on a packet according to the applied rules by the controller. Furthermore, we develop a formal tracing method that can predict all the possible routes of any set of packets from a specific node in the network. The formal predicates are based on second-order logic.[10] Offline verification methods are applied before the deployment of a new network policy which might consist of thousands of rules. In those cases, usually, some latency in the order of minutes can be tolerated. Once a network policy has been deployed, it is updated in an incremental manner which raises the need for so-called online verification. The verification of the changes in the policy in the form of new installed rules or removed rules should take place in a nearly real-time manner. Therefore, online verification methods face stringent requirements in terms of processing time in order to cope with the new policy updates. The online verification operates by the principle of only checking the affected set of packets due to the policy update. In Reference 11, we show how we can extend the offline verification method proposed in Reference 1 to cover online verification. In this article, we focus on the offline method for modeling the static networks and verifying policies before any changes.

The formal prediction method has been used for detecting the rules that are never matched by any packets. Moreover, the new descriptions for the intra- and inter-anomalies between the unmatched rules and the rest of the rules are presented. Finally, our detection methods, which we reckon as OpenFlow policy checker (OPC), is described in detail.

The main contributions of this article are as follows:

- Using few insights, we show that the speed of the detection anomaly method can be significantly enhanced using parallelization and generation completely disjoint queries.
- A multithread design is implemented and evaluated as a demonstration of the parallelization capability of the anomaly detection method.
- The OPC has adequate performance even in the presence of a considerable number of overlapping rules.
- We resort to the Stanford University Backbone Network[12] as a real-life topology for analyzing the functionality and performance of the OPC.
- We show that the performance of the intraanomalies detection method is independent of the number of detected anomalies.
- In addition to the Standford realistic topology, we design a synthetic topology for simulating the Last Match scenario according to which is at least a subset of input query that parses bottom rules of middle-boxes.
- A new definition for the intraanomalies has been described for OpenFlow with the *multiaction* feature. This definition generalizes the state-of-the-art intraanomalies definition, which is only based on the single-action feature.
- We introduce the nomenclature *irrelevant* and *invalid* rule anomalies for the case of unmatched rules.
- 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, OPC, in contrast to Netplumber and VeriFlow, considers dependencies between rules in flow tables.[13]
- By contrast to the header space analysis (HSA),[14] our method is a priority-based method, which makes it compatible with the OpenFlow protocol. Comparison results show that our OPC method outperforms HSA considerably in terms of execution time.

In Reference 1, we presented a very initial stage of the work presented in this article where only the single-thread approach was tackled. Moreover, in Reference 1 we just presented two detection algorithms without further explaining the rest of the algorithms. The implementation of the single-thread design was tested in Reference 1 to merely obtain the preliminary set of results based on the synthetic dataset. This current work presents a considerable extension of Reference 1 that can be summarized in a nutshell by (1) designing a parallel approach for detection, (2) further development of the detection algorithms, (3) using a realistic experimental set up based on the Standford Topology and reporting more through experimental results.

The remainder of the article is organized as follows. In Section 2, we provide a comprehensive overview of the state-of-the-art. Section 3 discusses our formal tracing method. In Section 4, the definition of anomalies in OpenFlow rules is described, and their detection is explained. Finally, the evaluation results are presented in Section 5.

2 | 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,[9] who introduced a real-time policy checking tool based on HSA[14] called NetPlumber. By 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 real-time traffic monitoring.

Netplumber generates what is called a dependency graph between rules and keeps this graph updated on insertion or deletion of rules. NetPlumber is able to not only detect loops and other invariant violations but also check more sophisticated failures in policy such as: "Web traffic from A to B should never pass through waypoints C or D between 9 AM and 5 PM." NetPlumber[9] was tested on both the Stanford backbone and the SDN network of Google. It was reported that checking the conformity of updated rule to the general policy takes 50 to 500 microseconds.

HSA[9] tries to find automatically typical failures in both operational and experimental networks regardless of the running protocols. The protocol-agnostic framework is based on a formalism that is able to detect classical misconfiguration problems such as reachability failures, loops, and traffic leakage problems. Apart from detecting the aforementioned misconfiguration problems, HSA also permits to check that the network slices are perfectly isolated according to the network slicing policy. HSA was tested on the Stanford University backbone network and found all the forwarding loops in less than 10 minutes. The verified reachability constraints between two subnets in 13 seconds.

It is worth mentioning that there are some other pioneering works such as Xie et al,[15] which checks the reachability statically based on the analyses of IP connectivity and the firewall configuration. However, HSA looks at the entire packet header as a concatenation of bits with no associated meaning. Thus, it can be considered as a protocol-independent automated method for policy checking.

Although Netplumber proposes a real-time method for detecting all typical violations, it ignores interrule dependencies in flow tables. Both HSA and Netplumber verification processes are time-consuming and could not be suitable for networks at a high rate of links up and down. Therefore, for Netplumber, which focuses on the real-time environment, this is a significant weakness. Please note that Netplumber and HSA are only able to detect misconfigurations and not prevent them. Moreover, physical defect is not detected in these methods.

Mai et al[16] tackle the misconfiguration problem by formal analysis of data plane state rather than by diagnosing bugs in the control plane. This approach is able to not only detect the "invisible" bugs in routing configuration files but also unified the analysis regardless of the many implementations and protocols. The authors try to develop a tool to collect the forwarding information bases of network devices 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 SAT solver to perform analysis. Anteater was deployed in a campus network with 178 routers, 70 000 users and a combination of BGP and OSPF routing protocols. It was able to detect 23 bugs, including loops and stale access control rules. Similarly to Netplumber and HSA, Anteater is not able to detect hardware defect.

Khurshid et al propose VeriFlow,[8] a tool to check inserted policies from the controller to the forwarding devices in real-time. VeriFlow can be seen as a layer between the controller and network devices for dynamically detecting violations of network policy invariants. VeriFlow can detect anomalies in the scenario of inserted, modified or deleted rules, using an incremental algorithm. The tool is able to not only raising the alarm immediately after detecting the violations but also blocking a modification process that would lead to anomalies such as loops or black holes. VeriFlow has been evaluated on a Mininet environment with a NOX controller, and it was shown that the process for each rule insertion or deletion takes hundreds of microseconds. However, VeriFlow's verification phase has a negative impact on the network performance as it inflates TCP connection setup latency by a significant amount, around 15.5% on average. In addition, the method does not consider rules dependency inside flow tables. Furthermore, VeriFlow is unable to detect a physical failure.

Al-Shaer and Al-Haj[5] 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,[17] which partitions the network resources into smaller segments. FlowChecker is able to detect both intraswitch and interswitch misconfiguration in a path of OpenFlow forwarding devices across the same or different infrastructure. It uses the binary decision diagram (BDD) to encode the flow tables. Afterward, it tries to model the interconnected 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 a new policy on the network. FlowChecker extends the Config-Checker[18] method in order to include the QoS configuration verification. In a federated OpenFlow infrastructure, the authors propose to run FlowChecker

as a Master Controller that interfaces the different domain controllers to detect and resolve the inconsistencies across different federated domains.

El Atawy et al[4] propose a firewall testing technique based on policy-based segmentation of the traffic address space. By separating the packet's header (ie, protocol, source address/port, destination address/port) into different segments and assign a distinct weight to each segment which reflects the importance of the segment, the accuracy of the testing process increased. The operator can assign specific weight to each segment according to the test scenario and whether the segment represents a critical domain. However, the weight usually is computed based on different factors such as the number of rules intersecting in the segment, which reflects the probability of an error occurring in that segment. The method is shown to achieve a superior detection accuracy to the random sampling method as it has a higher tendency to focus on the testing of potential fault locations.

Golnabi et al[19] use three main techniques to provide an automated tool for analyzing and managing firewall policies. Frequency analyses based on both data mining and filtering-rule generalization techniques are used for reducing the number of rules. Furthermore, the authors propose a method that tries to detect dominant firewall rules as well as decaying rules. By virtue of this algorithm, network administrators not only can detect some hidden but not detectable anomalies but also are able to review the firewall rules automatically and possibly reorder them based on matching frequency or edit them to remove or aggregate some rules.

Config-Checker[18] 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. It is concerned about security aspects in firewalls. The novelty of the method is the creation of a 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. The authors try to model the semantics of access control policies using BDDs. In addition, symbolic model checking and computation tree logic have been used to probe all past and future states of the packet in the network. According to this modeling, it is possible to verify the reachability. The scalability of the Config-Checker was evaluated by running the method on a network with thousands of devices and millions of configuration rules.

Sherwood et al[17] 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 OpenFlow 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 a 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.[20] However, it has a latency and overhead on the control channel due to the use of an additional TLS connection.

Parkinson et al[21] proposed GraphBAD to both detect security anomalies and suggest mitigation plans. GraphBAD generates an undirected graph model from the security configurations and logs data. Afterward, the anomalies and anomalous subgraphs are identified via analyzing the graph-based model. Two synthetic data and KDD dataset are used for evaluating the proposed method.

Son et al devised a model checking system called "FloVer,"[22] a formal approach to prove the conformance of dynamically produced OpenFlow flow rules against nonbypass 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 the OpenFlow network's policies does not breach the network security and integrity of its policies.

Hinrichs et al[23] propose a NOX-based application, which uses a language-based method for designing, implementing, and testing a flow-based network policy language and enforcement infrastructure. Their approach named FSL is able to model and express basic network access controls, directionality in communication establishment (similar to NAT), network isolation (similar to VLANs), communication paths, and rate limits. It supports modular construction, distributed authorship, and efficient implementation. This solution supports external authentication sources for providing access control. A significant effort has been dedicated to the issue of policy conflict resolution. Nevertheless, research on designing secure SDN is limited. Probably, one of the most significant contributions in this regard is the implementation of an OpenFlow security application development framework called Fresco.[24] This framework is integrated with FortNox,[25] which is a security enforcement kernel. FRESCO's idea is to propose a rapid architecture and implementation of specific security modules, which are incorporated as an OpenFlow application.

Porras et al[25] use reusable modules based on the FortNox enforcement engine for detecting and mitigating network conflict and threats. Based on those modules, an inserted rule is analyzed to detect its effect on enabling or disabling prohibited/allowed existing rules. The new OpenFlow rule might be rejected or accepted depending. It prevents rules issues by lower priority applications from overriding rules generated by higher priority security applications.

This article deals with policy conflicts in OpenFlow switches. We propose an approach to detect typical problems such as loops, black holes, and policy violations based on a devised routing prediction method, and query-based auditing processes. The second-order logic is used for describing the query-based checking process.

3 | ROUTING PREDICTION BASED ON THE 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 input traffic could be helpful.

In this section, we 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.[13]

Moreover, all common policy misconfigurations in SDN, such as loops and black holes, can be detected via our proposed method. We adopt second-order logic for expressing the packet tracing process. Meanwhile, the input process in the middleboxes and the interdependency of rules have been represented via the Raining 2D-Box model. In this section, we explain the prediction method and all the functions for misconfiguration detection.

3.1 | Tracing function

The function "T" defines a recursive traceroute 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.

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

X denotes a node, which comprises a set of rules, $X:\{R_{1_x},R_{2_x},\dots\}$. 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
Rules ← Starting_Hop.Rules
foreach rule in Rules do
    if Query ∧ rule.Condition then
        Route ← rule.Action
        if rule.Action = Client Or rule.Action = Drop then
            return Route
        end
        Tracing_Function(rule.Action, Query, Route)
    end
end
```

3.2 | 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[26] that is shown in Figure 1. In this model, the input is checked sequentially against the higher priority rules as depicted in Figure 2.

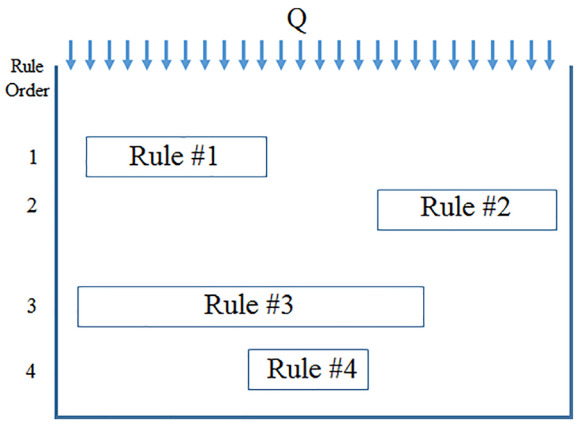


FIGURE 1 The rules' order and input query in Raining 2D-Box model

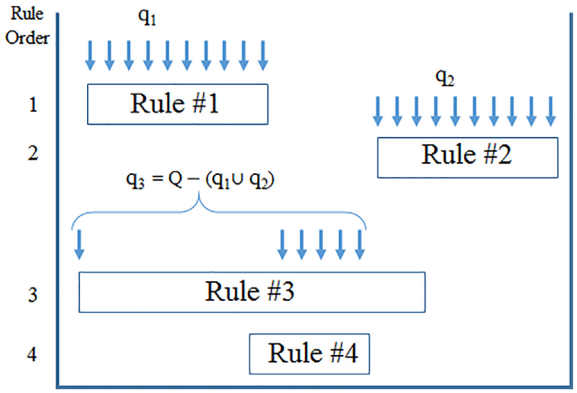


FIGURE 2 The input query, rules' intradependency and rules' order in Raining 2D-Box model

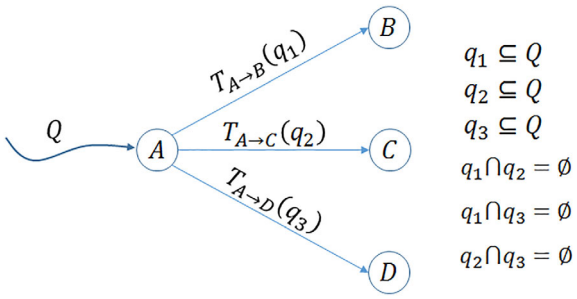


FIGURE 3 Simple directed graph and its transfer function

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 a formal definition of the transfer function. Figure 3 illustrates via a sample example the transfer operation in a directed graph. According to Figure 4 and Equation (2), the results of the transfer function can be described as follows:

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

$$q_3 \cup q_4 \cup q_8 = q_1$$

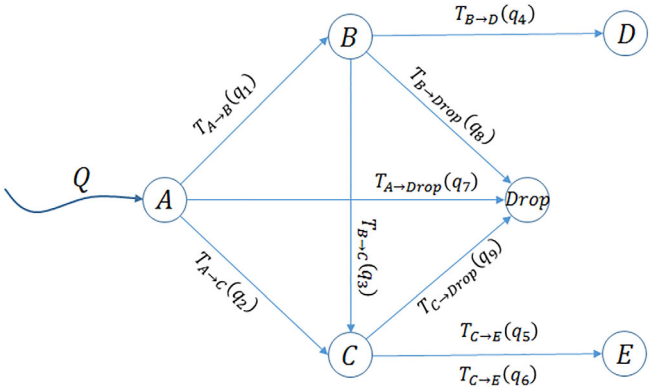
$$T_{A\to C}(q_2) : \{T_{C\to E}(q_6), T_{C\to Drop}(q_9))\}$$

$$T_{B\to C}(q_3) : \{T_{C\to E}(q_5), T_{C\to Drop}(q_9))\}$$

$$q_6 \cup q_5 \cup q_9 = q_2 \cup q_3$$

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

FIGURE 4 Sample directed graph and its transfer function



Therefore, based on Equations (1) and (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 Figure 4, 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 Figure 4, 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 algorithm.

3.3 | Reachability checking between endpoints

As mentioned above, reachability checking is one of the critical troubleshooting operations for a network administrator that is dealing with complex networks. According to the tracing function (Equation (1)) and the transfer function (Equation (2)), it is possible to check the reachability between to endpoints, that is, whether one specific host can connect to another specific host. The reachability checking method is defined based on 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, each of which comprises a set of rules.

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

Since the OPC is an offline method, we assume that whenever a packet is faced with table-miss[1], the tracing 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 Figure 5 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*.

[1]Whenever a table-miss takes place, the packet gets forwarded to the controller according to the OpenFlow protocol.

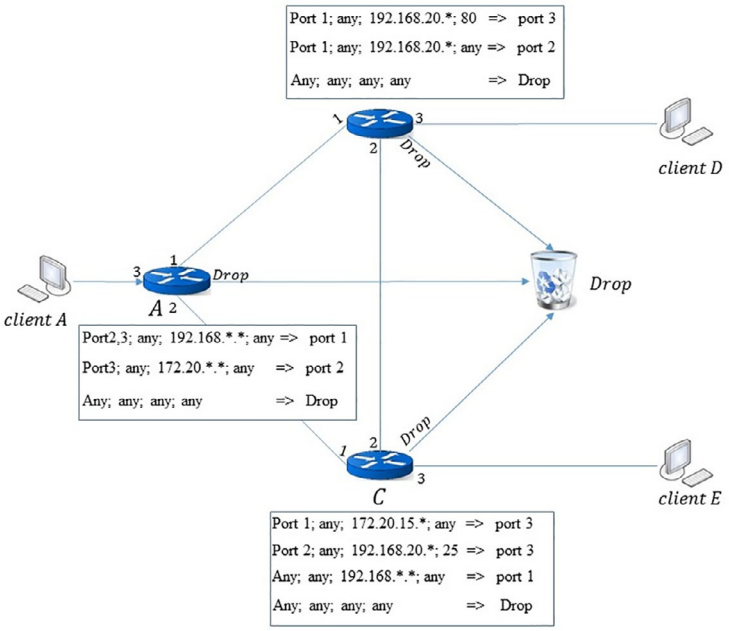


FIGURE 5 Sample network with routing tables

Algorithm 2. Reachability checking function

Input: Query, Starting_Hop, Destination_Hop
Output: Boolean Result
1 **foreach** *q in Query* **do**
2 Route ← Starting_Hop
3 Hops ← All hops in TRACING_FUNCTION (Starting_Hop, q, Route)
4 **if** *the last hop* ***in*** *Hops = Destination_Hop* **then**
5 **return** True
6 **end**
7 **end**
8 **return** False

3.4 | Reachability checking through a specific hop

In addition to checking the reachability between two nodes, A and B , a network administrator might also want to check whether the traffic from A to B passes through a given node C. The answer is useful for many debugging procedures. As stated in Equations (2) and (3), it is feasible to check the result of the query by second-order logic (Equation (4)). The predicate ψ has been declared for checking the reachability from a node X to a node Y by a query Q with the condition that the traffic passes through the node K. Each node X,Y, and K comprises a set of rules, such that, for node X, for instance: $X : \{R_{1_x}, R_{2_x}, \ldots\}$. Q represents a query, which could check the reachability of one or a set of packets.

$$\psi(X, Y, K, Q) : \exists q.q \in Q[T_{X\to Y}(q) \wedge (K \in T(X, q))]. \tag{4}$$

Based on Equation (4), The Conditional_Reachability_Function has been developed and shown in Algorithm 3. According to Figure 5 and Equation (4), the result of predicate ψ for the considered scenarios can be written as follows:

$$Q = \text{“}Port3, *.*.*.*, 192.168.20.5, 80\text{”}\ \psi(A, D, B, Q) = True$$
$$Q = \text{“}Port3, *.*.*.*, 192.168.20.5, 25\text{”}\ \psi(A, E, B, Q) = True$$
$$Q = \text{“}Port3, *.*.*.*, 172.20.15.5, 80\text{”}\ \psi(A, E, B, Q) = False$$
$$Q = \text{“}Port3, *.*.*.*, 172.20.15.5, 25\text{”}\ \psi(A, E, C, Q) = True$$
$$Q = \text{“}Port3, *.*.*.*, 192.168.20.5, 25\text{”}\ \psi(A, E, C, Q) = True$$
$$Q = \text{“}Port3, *.*.*.*, 192.168.20.5, 25\text{”}\ \psi(A, D, B, Q) = False$$

Algorithm 3. Conditional Reachability Function

```
Input: Query, Starting_Hop, Destination_Hop, Target_Hop
Output: Boolean Result
foreach q in Query do
    Route ← Starting_Hop
    Hops ← All hops in TRACING_FUNCTION (Starting_Hop, q, Route)
    if the last hop in Hops = Destination_Hop then
        if Target_Hop is in Hops then
            return True
        end
    end
end
return False
```

3.5 | Loop checking

Conforming to the second-order logic and Equation (2), the loop checking function could be declared as by Equation (5). The predicate L has been declared for checking the existence of a loop in all possible routes by a query Q from hop X. X represents a node, which comprises a set of rules, such as $X : \{R_{1_x}, R_{2_x}, \dots\}$. Q represents a query, which could do loop checking for a specific packet or for a set of packets.

$$L(X, Q) : \exists q.q \in Q \left[\exists Z.Z \in T(X, q) \left[\exists q'.q' \in Q \right.\right. \\ \left.\left. [(Z \in T(X, q')) \wedge (q' \neq q)]\right]\right]. \tag{5}$$

By virtue of Equation (5), the Loop_Checking_Function has been developed and is presented in Algorithm 4. According to Figure 5 and Equation (4), the result of predicate L for the considered scenarios is as follows:

$$Q = \text{“}Port3, *.*.*.*, 192.168.20.5, 80\text{”}\ L(A, Q) = False$$
$$Q = \text{“}Port1, *.*.*.*, 192.168.20.5, 25\text{”}\ L(B, Q) = False$$
$$Q = \text{“}Port3, *.*.*.*, 172.20.15.5, 80\text{”}\ L(A, Q) = False$$
$$Q = \text{“}Port3, *.*.*.*, 172.20.15.5, 23\text{”}\ L(A, Q) = True$$

Algorithm 4. Loop checking function

```
Input: Query, Starting_Hop
Output: Boolean Result
foreach q in Query do
    Route ← Starting_Hop
    Hops ← All hops in TRACING_FUNCTION (Starting_Hop, q, Route)
    foreach hop in Hops do
        foreach q′ in Query do
            Route′ ← Starting_Hop
            Hops′ ← All hops in TRACING_FUNCTION (Starting_Hop, q′, Route′)
            if q′ ≠ q && hop is in Hops′ then
                return True
            end
        end
    end
end
return False
```

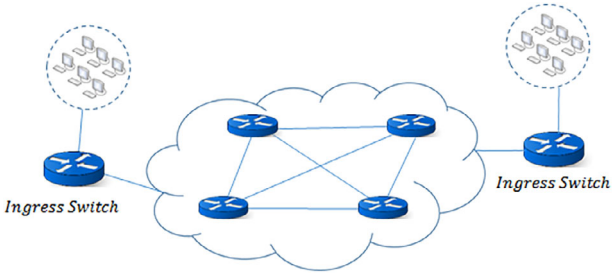


FIGURE 6 Ingress switch in network

4 | ANOMALY DETECTION ON SDN-SWITCHES

In this section, we present an offline anomaly detection approach, based on the formal methods defined in Section 3. The proposed approach 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.

4.1 | 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 the rest of the network. Figure 6 sketches a sample for 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[2](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 (6). 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.

$$\begin{aligned} Query_1 &: Rule_1 \\ Query_2 &: Rule_2 - Rule_1 \\ Query_3 &: Rule_3 - (Rule_1 \cup Rule_2) \\ &\vdots \\ Query_n &: Rule_n - \left(\bigcup_{i=1}^{n-1} Rule_i\right). \end{aligned} \tag{6}$$

While calling the transfer function (3.2), 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 the subject of the following subsection.

4.2 | OpenFlow rule anomaly

Sometimes one rule will never be matched by any possible queries, and there are several reasons for this type of anomaly. 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] define three more anomalies, namely, total shadowing, total generalization, and total redundancy. Moreover, interanomalies, which might occur in distributed firewalls, have been defined by Al-Shaer and Hamed[27] 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 intraanomalies here. However, as explained by Reference 28, OpenFlow-based rules might have more than one action, that is, multiaction. Thus, we shall propose a new expression for intraanomalies of rules that supports multiaction. To the best of our knowledge, this aspect has not been investigated in the literature before. In addition, another categorization has been represented for the interanomalies of rules. Therefore, the unmatching of rules can be a result of intra- or inter-anomalies, which will be defined in greater detail below.

[2]Without loss of generality, we suppose that the subtraction operation operates on the matching condition C_i.

4.2.1 | Intraanomaly for single-action and multiaction

An intraanomaly takes place between rules on the same device. According to References 6 and 7, these types of anomalies are categorized into seven groups. We shall use the bitwise format defined in Reference 7 in order to rewrite rules and packets. The formal specification of OpenFlow rule anomalies is put forward as follows.

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:

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

As per Equation (7), rule R_i is shadowed by the rule R_j for the group of actions, which are true in $(A_i \oplus A_j)$.

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:

$$R_{i_{priority}} < R_{j_{priority}}$$
$$R_i : (C_i, A_i), R_j : (C_j, A_j)$$
$$\exists R_i, R_j \in FlowTable$$
$$[\neg(C_i \Rightarrow C_j) \wedge \neg (C_j \Rightarrow C_i) \wedge (C_i \wedge C_j)]$$
$$\wedge (A_i \oplus A_j). \tag{8}$$

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

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:

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

According to Equation (9), rule R_i and rule R_j have generalization for the group of actions that which are true in $(A_i \oplus A_j)$.

Redundant anomaly

Rule R_i is redundant to Rule R_j if they have the 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:

$$R_i : (C_i, A_i), R_j : (C_j, A_j)$$
$$\exists R_i, R_j \in FlowTable\ [(C_i \Rightarrow C_j) \vee (C_j \Rightarrow C_i)]$$
$$\wedge (A_i \wedge A_j). \tag{10}$$

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

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 a different action from the previous rules. Formally, rule R_i is totally shadowed by rules $\{R_1 \ \dots \ R_k\}$ if the following condition holds:

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

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

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 $\{R_1 \ \dots \ R_k\}$ if the following condition holds:

$$\begin{gathered} R_{i_{priority}} < R_{1_{priority}}, \ \dots \ , R_{k_{priority}} \\ R_i : (C_i, A_i), R_1 : (C_1, A_1), \ \dots \ R_k : (C_k, A_k) \\ \exists R_i, R_1, \ \dots \ , R_k \in FlowTable \left(C_i \Rightarrow \left(\bigvee_{n=1}^{k} C_n \right) \right) \\ \wedge \left(\left(\bigvee_{n=1}^{k} A_k \right) \wedge A_i \right). \end{gathered} \tag{12}$$

As per Equation (12), rule R_i and rules in the set: $\{R_1 \ \dots \ R_k\}$ have total redundancy for the group of actions that are true in $((\bigvee_{n=1}^{k} A_k) \wedge A_i)$.

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 $\{R_1 \ \dots \ R_k\}$ if the following condition holds:

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

As described by Equation (13), rule R_i and rules in the set: $\{R_1 \ \dots \ R_k\}$ have total generalization for the group of actions that are true in $((\bigvee_{n=1}^{k} A_k) \oplus A_i)$.

4.2.2 | Interanomaly

According to the nomenclature proposed in Reference 27, at any point along the path of a given flow, a preceding switch is called an upstream hop whereas the 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 InterAnomaly takes place. Note that in this section, we assume that the flow tables are intraanomaly free. The Al-Shaer and Hamed[27] categorize the interanomalies in four groups. By contrast to Reference 27, this article defines four types of interanomalies in a different way, which are the root cause of unmatched rules.

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:

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

In Equations (14) to (16), *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).

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:

$$\begin{aligned} &R_i : (C_i, A_i), R_j : (C_j, A_j) \\ &\exists R_i \in SW_i, R_j \in SW_j \; Upstream\ (SW_j) \\ &\wedge (C_j \Rightarrow C_i) \wedge \neg\ \varphi\ (SW_j, SW_i, C_j). \end{aligned} \tag{15}$$

Partial rule anomaly

A partial rule anomaly occurs if just parts of packets, which 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:

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

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:

$$\begin{aligned} &R_i : (C_i, A_i), R_j : (C_j, A_j) \\ &\forall sw \in ingress, \left[\exists rule \in R \left[\nexists packets, (packets \wedge C_{rule})\right.\right. \\ &\left.\left. \wedge\ (T(sw, packet) \notin Ex\_Path)\right] \Leftrightarrow irrelevant(rule)\right]. \end{aligned} \tag{17}$$

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.

4.2.3 | 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 (18).

$$\forall q \in Q[\exists r \in R[\neg\ (q \wedge C_r)] \Leftrightarrow invalid(r)]. \tag{18}$$

As expressed by Equation (18), *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.

4.3 | Anomaly detection

In the previous subsection, unmatched rules' anomalies are defined and categorized. In this subsection, we will describe the detection method. According to our anomaly definition, there are three main groups of anomalies, namely, intraanomaly, interanomaly, and invalid rule anomaly. After the unmatched rule detection, the following steps will be performed to find which rule in which switch that causes the anomaly in question.

4.3.1 | Invalid rule anomaly detection

The unmatched rules are checked based on Equation (18) to detect invalid anomalies. The result determines the rules that might never be matched by all possible queries. Due to the fact that the input queries are generated based on the rules in ingress switches, our queries represents all possible traffic which can pass through the network from the clients. The invalid rule 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. The detection process is described in greater details in Algorithm 5. 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 to find any intersecting query, the rule is moved to the invalid rules list.

4.3.2 | Intraanomaly detection

The intraanomaly detection operation is executed after the removal of the invalid rules from the unmatched rule list. Each remaining member of the list will be checked with the other rules in the same flow tables to find the possible intraanomalies. The conflicting rules, together with the anomaly types, are reported to the network administrator for making a decision. In relation to Algorithm 6, for each unmatched rule, its corresponding flow table is fetched. Then, the target rule and its flow table are checked by the simple and total anomaly detection algorithms. The Simple_Anomaly_Detection_Function is described by Algorithm 7. According to the Algorithm 7 and Equations (7) to (9), 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 (10), the unmatched rule will be checked with the rules that have the same action. The Total_Anomaly_Detection_Function, which is shown in Algorithm 8 aims to detect the total anomalies based on Equations (11) to (13). In the first step, the algorithm collects the rules that have lower priority from the unmatched rule and have a partial intersection with it. Then, total generalization anomaly condition (Equation (13)) is checked for the part of collected rules that have a different action than the unmatched rule. In addition, the total redundancy anomaly condition (Equation (12)) 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 (11)) is checked for the part of collected rules that have different action from the unmatched rule. Subsequently, the rest of the rules, which have the same action with the unmatched rule, are checked for the total redundancy anomaly condition (Equation (12)).

Algorithm 5. Invalid rule detection function

```
   Input: Query_List, unmatchedRule_List
   Output: InvalidRules_List
 1 foreach rule in unmatchedRule_List do
 2 |  Matched = False
 3 |  foreach q in Query_List do
 4 |  |  if q ∧ rule.Condition then
 5 |  |  |  Matched = True
 6 |  |  |  break
 7 |  |  end
 8 |  end
 9 |  if !(Matched) then
10 |  |  InvalidRules_List ← rule
11 |  |  Remove rule from unmatchedRule_List
12 |  end
13 end
14 return InvalidRules_List
```

Algorithm 6. Intra-Anomaly detection function

```
   Input: unmatchedRule_List
   Output: Conflicting_Rules_List
1 foreach rule in unmatchedRule_List do
2 |  flow_table←fetch_flow_table (rule)
3 |  Conflicting_Rules_List←Simple_Anomaly_Detection_Function(rule, flow_table)
4 |  Conflicting_Rules_List←Total_Anomaly_Detection_Function (rule, flow_table)
5 end
6 return Conflicting_Rules_List
```

Algorithm 7. Simple anomaly detection function

```
Input: New_Ruel, Flow_Table
Output: Conflicting_Rules_List
foreach rule in Flow_Table do
    if Rule.priority>rule.priority then
        R1 = New_Rule, R2 = rule
    else
        R1 = rule, R2 = New_Rule
    end
    if R1.action ⊕ R2.action then
        if R2.condition ⇒ R1.condition then
            Conflicting_Rules_List← R1, R2,"Shadow Anomaly"
        else
            if R1.condition ⇒ R2.condition then
                Conflicting_Rules_List← R1, R2,"Generalization Anomaly"
            else
                if R1.condition ∧ R2.condition then
                    Conflicting_Rules_List← R1, R2,"Correlation Anomaly"
                end
            end
        end
    else
        if R2.condition ⇒ R1.condition then
            Conflicting_Rules_List← R1, R2,"Redundancy Anomaly"
        end
    end
end
return Conflicting_Rules_List
```

4.3.3 | Interanomaly detection

The interanomaly detection operation assumes that the flow tables are intraanomaly 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 (Algorithm 9), 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 interanomaly is reported. Whenever the paths are different, the Check_Rule_InterAnomaly_Function will be called. Finally, according to Equation (17), 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. This anomaly usually takes place whenever an administrator updates the policies and does not check the side effect of policies' modification. So, the new or modified policies lead the irrelevant rule which is never matched by any packet, which being matched before the update. The Check_Rule_InterAnomaly_Function aims to check and detect the anticipated anomalies between the unmatched rule and the rule that causes the conflict. According to Algorithm 10, 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 (14) is checked between two rules. If it is not verified, the superset anomaly condition, which is represented by Equation (15), will be inspected. Finally, as per Equation (16), the partial anomaly condition will be checked.

Algorithm 8. Total anomaly detection

```
Input: Ruel, Flow_Table
Output: Conflicting_Rules_List
totalR=False, R=Ø, totalG=False, G=Ø
foreach rule in unmatchedRule_List && rule.priority<Rule.priority do
    if !(rule.condtion⇒Rule.condition)&& !(Rule.condtion⇒rule.condition)&& (rule.condtion∧Rule.condition) then
        if Rule.action⊕rule.action then
            totalG=toalG∨rule.condition
            G←rule
            if Rule.condition⇒totalG.condition then
                Conflicting_Rules_List←Rule,G,"Total Generalization Anomaly"
            end
        else
            if !(Rule.action⊕rule.action) then
                totalR=toalR∨rule.condition
                R←rule
                if totalR.condition⇒Rule.condtion then
                    Conflicting_Rules_List←Rule,R,"Total Redundancy Anomaly"
                end
            end
        end
    end
end
totalR=False, R=Ø, totalS=False, S=Ø
foreach rule in unmatchedRule_List && rule.priority>Rule.priority do
    if !(rule.condtion⇒Rule.condition)&& !(Rule.condtion⇒rule.condition)&& (rule.condtion∧Rule.condition) then
        if Rule.action⊕rule.action then
            totalS=toalS∨rule.condition
            S←rule
            if Rule.condition⇒totalS.condition then
                Conflicting_Rules_List←Rule,S,"Total Shadowing Anomaly"
            end
        else
            if !(Rule.action⊕rule.action) then
                totalR=toalR∨rule.condition
                R←rule
                if totalR.condition⇒Rule.condtion then
                    Conflicting_Rules_List←Rule,R,"Total Redundancy Anomaly"
                end
            end
        end
    end
end
return Conflicting_Rules_List
```

Algorithm 9. Inter-Anomaly detection function

```
Input: unmatchedRule_List
Output: Conflicting_Rules_List
foreach Rule in unmatchedRule_List do
    Irrelevant_Rule=True
    foreach sw in ingress_swithes do
        paths←Transfer_Function(sw,Rule)
        foreach path in paths do
            ex_path←expected path by admin
            if Rule.hop∈ex_path then
                Irrelevant_Rule=False
            end
            if Path≠Ex_patch then
                Conflicting_Rules_List←Check_Rule_Inter-Anomaly_Function(path,ex_path,Rule)
            end
        end
    end
    if Irrelevant then
        Conflicting_Rules_List←Rule,"Irrelevant Rule"
    end
end
return Conflicting_Rules_List
```

Algorithm 10. Check rule inter-anomaly Function

```
Input: Path,Ex_path,Rule
Output: Conflicting_Rules_List
hop←last_common_hop(Path,Ex_path)
rule←hop.matched_rule
if !(Reachability_Checking_Function(hop,Rule.hop,Rule.Condition)) then
    if Rule.condition⇒rule.condition then
        Conflicting_Rules_List←Rule,rule,"Subset Anomaly"
    else
        if rule.condition⇒Rule.condition then
            Conflicting_Rules_List←Rule,rule,"Superset Anomaly"
        else
            if Rule.condition∧rule.condition then
                Conflicting_Rules_List←Rule,rule,"Partioal Subset Anomaly"
            end
        end
    end
end
return Conflicting_Rules_List
```

5 | EVALUATION

In this section, we evaluate our proposed method, OPC, which consists of four main phases: test query generating, probing process, intraanomaly detection, and interanomaly detection. The single thread and multithread approaches are implemented, and the results are compared. All methods are implemented in C++, and the experiments are run on the server with 48 Intel (R) Xeon (R) CPU 2.30 GHz.

Two network topologies are used in the evaluation: a mock setup with a fat-tree topology (Figure 11) and the Standford topology (Figure 15).[12] In order to generate realistic OpenFlow rules, we use the Class Bench[29] tool. The Stanford University Backbone Network[12] depicted in Figure 15 represents a real-life topology widely used in the literature. The algorithm is run 30 times for each distinctive rule dataset. We report the average execution time with a 95% confidence interval. Results are presented in the following subsections.

5.1 | Test query generation

In this subsection, the execution time of the test query generation process for different flow table sizes is evaluated. The "Test Query Generator Engine" is capable of generating all possible disjoint queries based on the flow tables of a specific ingress switch according to Equation (6).

As illustrated by Table 1 and Figure 7, the execution time increases dramatically whenever the ruleset size exceeds 5000 rules in the mock setup. However, multithread implementation yields different results. According to Table 2 and Figure 8, the query generation processing time does not

TABLE 1 Single-thread query generation processing time for the mock setup

Rules num.	Average (s)	CI (95%)
500	26.600	± 0.569
1000	96.539	± 1.163
2000	416.209	± 9.205
5000	2448.830	± 45.063
10 000	9736.508	± 12.815

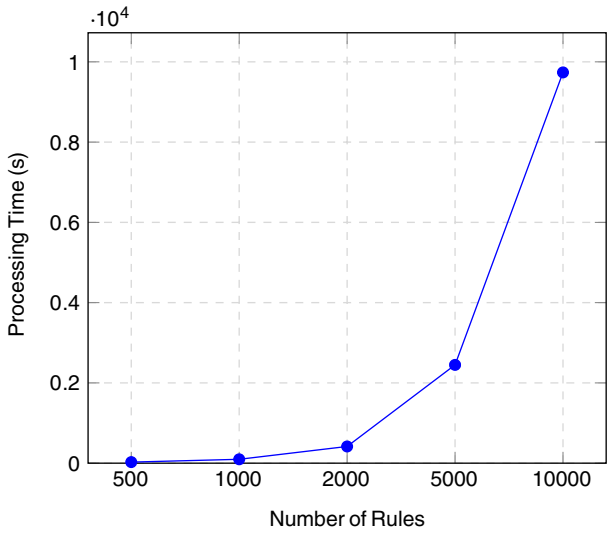


FIGURE 7 Query generating processing time with single-thread approach for the mock setup

TABLE 2 Multithread query generation processing time for the mock setup

Rules num.	Average (s)	CI (95%)
500	1.170	± 0.012
1000	4.699	± 0.054
2000	19.529	± 0.146
5000	116.78	± 0.799
10 000	459.879	± 2.767

FIGURE 8 Query generating processing time with multithread approach for the mock setup

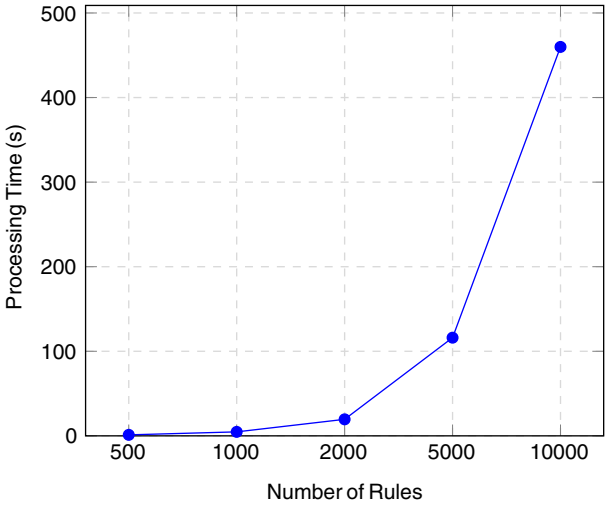


TABLE 3 Single-thread query generation processing time for the Stanford network

Switch	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9	SW_10	SW_11	SW_12	SW_13	SW_14	SW_15	SW_16
Rules num.	870	844	203	175	188	125	167	146	124	109	104	91	204	142	248	116
Average (s)	17 412.23	17 919.59	460.15	300.27	481.26	89.92	354.98	229.01	155.71	103.01	103.79	72.64	624.75	123.5	1462.26	108.89
CI (95%)	± 14.09	± 158.33	± 6.42	± 2.80	± 6.75	± 0.33	± 4.13	± 1.80	± 1.26	± 0.29	± 0.47	± 0.13	± 13.11	± 0.34	± 26.90	± 0.34

TABLE 4 Multithread query generation processing time for the Stanford network

Switch	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9	SW_10	SW_11	SW_12	SW_13	SW_14	SW_15	SW_16
Rules num.	870	844	203	175	188	125	167	146	124	109	104	91	204	142	248	116
Average (s)	27.9	27.06	3.05	2.1	3.72	1.05	2.67	1.84	1.63	1.2	1.24	0.97	4.51	1.27	7.57	1.3
CI (95%)	± 0.08	± 0.11	± 0.01	± 0.01	± 0.02	± 0.01	± 0.01	± 0.01	± 0.01	± 0.01	± 0.01	± 0.01	± 0.03	± 0.09	± 0.01	± 0.01

change dramatically by increasing the number of rules, and the process is exceedingly faster than a single-thread approach. As we can see from Tables 1 to 2 that the parallel method outperforms the single-threaded checking method, and was about 21 times faster when we ran it on 10 000 rules. In the Stanford network, query generating algorithm is applied for each switch. The processing time of single-thread and parallel approach are presented in Tables 3 and 4, respectively. As depicted in Figures 9 and 10, the parallel method was about 624 times faster than the single-thread method when ran it on the SW_1, which contains 870 rules.

Comparing the obtained results of the single-thread and multithread approaches shows drastic performance improvement. This is due to the fact that the query generation process for each rule in the ingress switch can be performed independently from the other rules.

5.2 | Probing process

The "Probing Process Engine" is capable of detecting unmatched rules based on the test queries in the snapshot of the flow tables. The queries are defined by the test query generator engine. For evaluating the probing process, a query, which contains ingress port, source IP, destination IP, and destination port, is used. The probing engine receives policy tables of middle-boxes and the test queries. The content of OpenFlow switch tables are fetched by the dump command,[30] 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. Then the tool converts the inputs into a binary content.

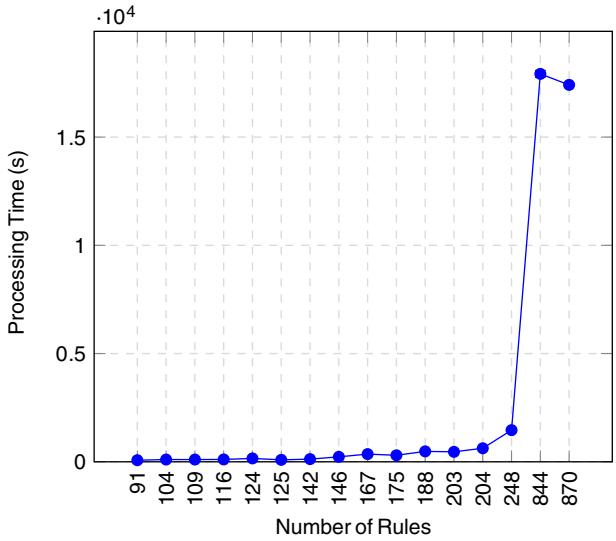


FIGURE 9 Query generating processing time with single-thread approach for Stanford network

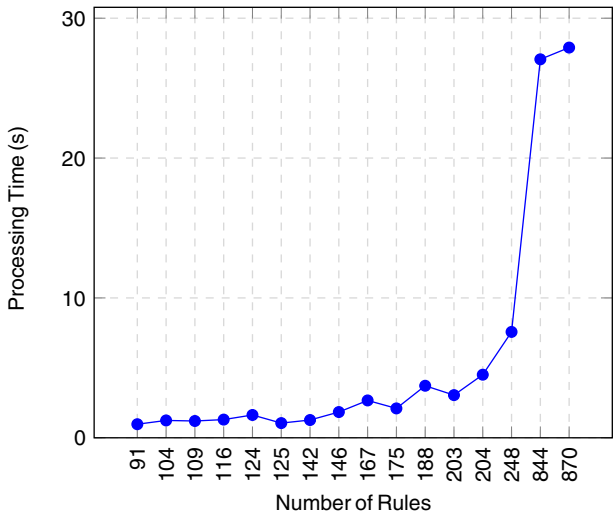


FIGURE 10 Query generating processing time with multithread approach for Stanford network

Finally, we call the "Transfer-Function" to parse and check the query destiny. After each iteration, the "Loop_Checking functions" is called for diagnosing loops. If a loop is detected, the process will be stopped. The probing process will also be terminated for a specific query if it received by the client or dropped by a rule in one of the middle-boxes.

According to this procedure, with the predefined network's flow tables and the test queries in the mock setup, 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.

In order to test the scalability of our approach, we create a synthetic dataset with a varying number of rules in each middle-box. In this evaluation, we use five datasets involving 45 switches each containing 500, 1000, 2000, 5000, and 10 000 rules, respectively. According to Figure 11, the switches are configured as a fat-tree topology. In order to generate a stress testing, we deliberately define position the rules that match the test queries at the bottom of each of the flow tables. We call this scenario the *Last Match* scenario. After the subtraction operation, the remaining set would be matched with the next rules. Clearly, there is at least a subset of query which parses all the middle-boxes. Therefore, all the flow tables are read, and for each input query in each middle-box, we have at least three disjoint output as new queries for the next switches.

The processing time for the probing procedure using the single-thread and the multithread approaches are presented in Tables 5 and 6, respectively. Table 7 shows the number of rules in each middle-box and the total number of rules for each dataset. Moreover, it provides the number of the transfer function call in both single-thread and multithread approaches.

As illustrated in Figures 12 and 13, the processing time for both approaches is roughly similar. In other words, the single-thread and multithread produce similar results when it comes to the Last Match scenario, and the queried rules are at the bottom. Thus, there is a little possibility for creating

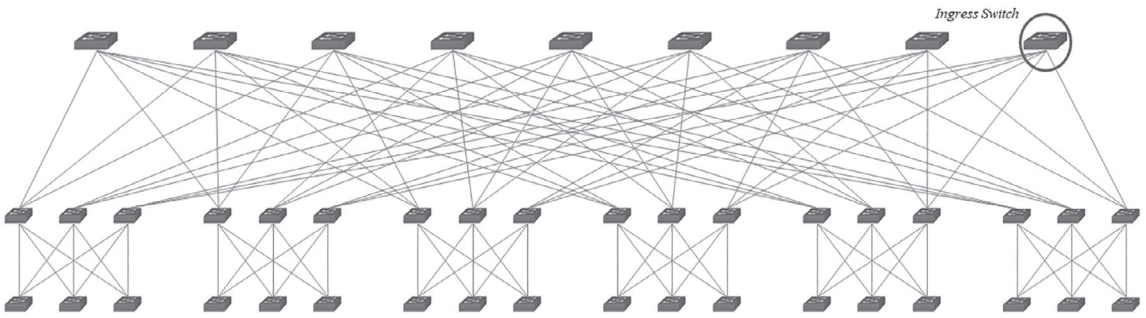


FIGURE 11 The mock topology

TABLE 5 Single-thread probing processing time for the mock setup

Rules num.	Average (s)	CI (95%)
22 500	23.678	± 0.23
45 000	46.985	± 0.44
90 000	87.857	± 1.4
225 000	201.714	± 3.7
450 000	384.428	± 8.8

TABLE 6 Multithread probing processing time for the mock setup

Rules num.	Average (s)	CI (95%)
22 500	22.873	± 0.013
45 000	39.751	± 0.06
90 000	81.077	± 0.168
225 000	180.057	± 0.215
450 000	338.234	± 0.642

TABLE 7 Probing process function call

Middle-box rules	500	1000	2000	5000	10 000
Total number of rules	22 500	45 000	90 000	225 000	450 000
Call transfer function	1.2×10^6	2.06×10^6	4.02×10^6	7.84×10^6	14.72×10^6

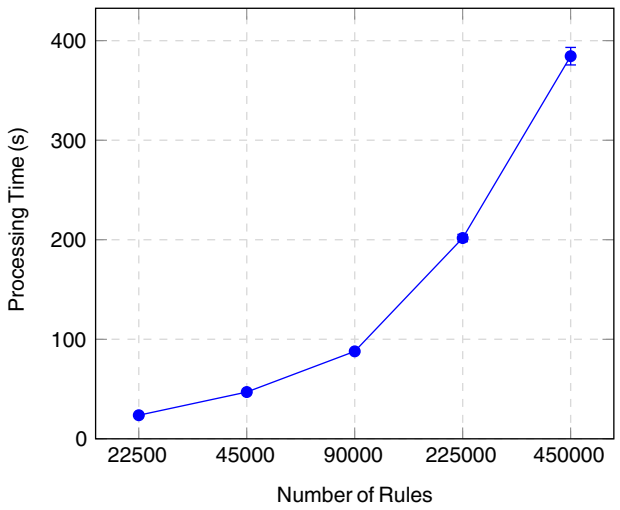


FIGURE 12 Probing processing time with single-thread approach for the mock setup

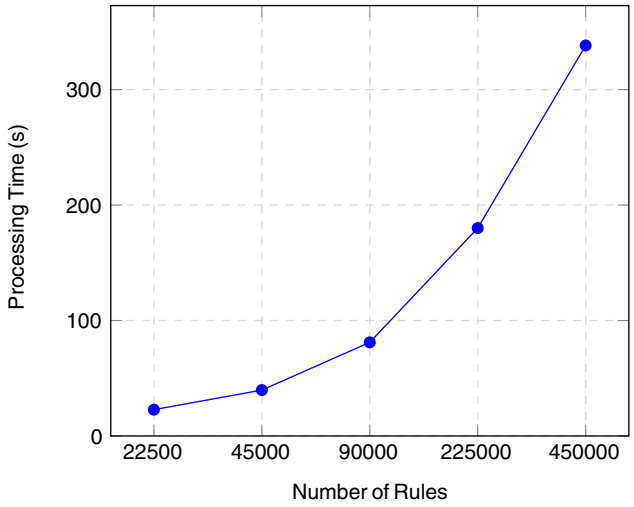


FIGURE 13 Probing processing time with multithread approach for the mock setup

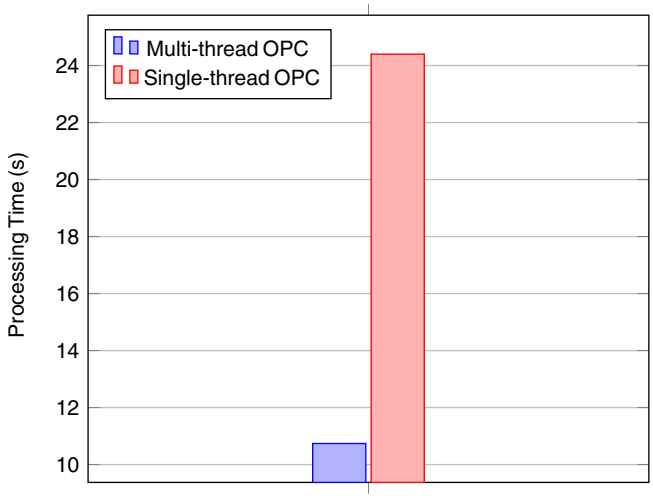


FIGURE 14 Average probing processing time for the Stanford network

several "branches" and checking them in parallel. A branching process refers to the case where an input query is matched by a rule with multiple actions in a flow table.

The probing algorithm is evaluated on the Stanford network as a real-world scenario. The mock setup tries to simulate the Last Match scenario as by design and given a query originating ingress, and there is at least a subset of this query that parses all the middle-boxes and produces a match in the last rule of each switch. However, in the Stanford topology, the branching can happen in each rule position and not necessarily starting at the last rules. It causes a new process generated for each new subquery. Therefore, this scenario can evaluate another aspect of the performance of the algorithm different from the Last Match scenario. As Figure 14 demonstrates, the processing time of multithread approach in average is faster than single-approach because there are more branches in this scenario which gives an opportunity to the algorithm to use its parallelizing capability.

Finally, we perform another performance check with introducing a modification in the rules the Stanford Topology. We rewrite the rules of each OpenFlow table in the Stanford Topology and converted them to totally disjoint rules. It means there is no overlap between any pair of rules in the same switch. Accordingly, the number of rules in each switch increases dramatically. On the other hand, subtraction is not required between disjoint rules and input queries. The assessment tries to find out the effect of subtraction function and size of the flow table on the performance of the probing algorithm Figure 15. The comparison of processing time average between the Stanford topology with and without disjoint rules is shown in Figure 16.

According to Figure 16, the size of the flow table has a negative effect on the performance of the probing process and reducing the overlap between rules cannot improve the performance considerably. This is due to the fact that the input queries should be checked with all the rules in a flow table sequentially. Therefore, the processing time for each branch is increased when the number of rules in each switch is considerably higher than the number of branches.

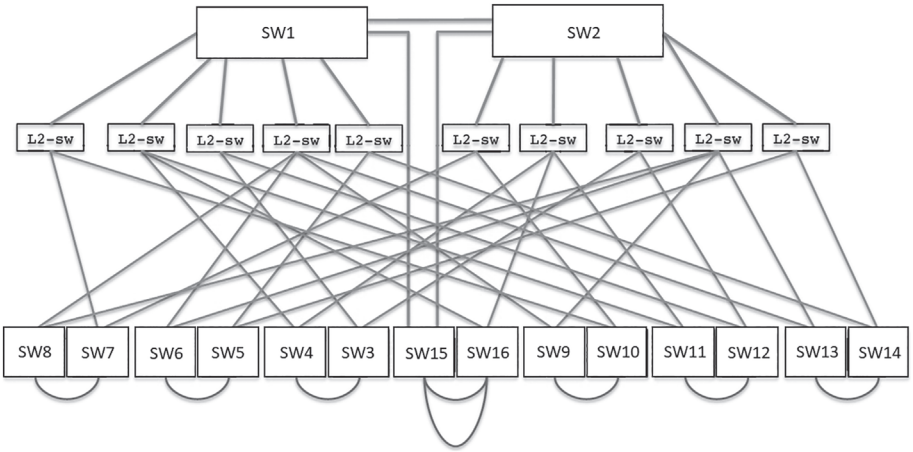


FIGURE 15 Stanford backbone topology

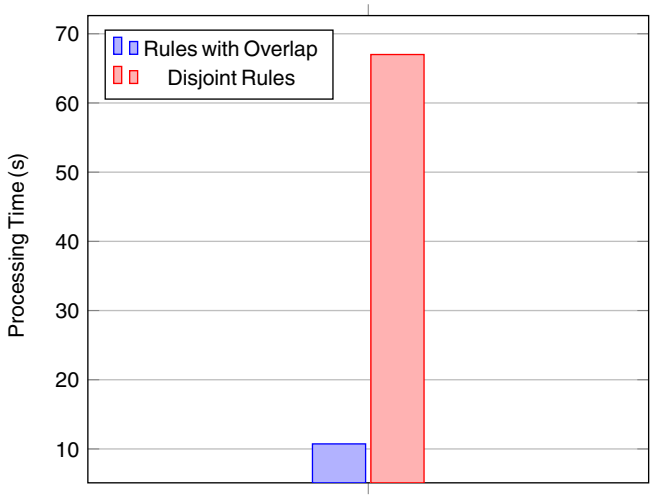


FIGURE 16 Rules with overlap vs disjoint rules average probing processing time for the Stanford network

As seen in Equations (14) to (17), the interanomaly detection resorts to the probing process. Therefore, the engine execution time follows the same pattern under different number of rules.

At this juncture, we compare the performance of the OPC with the HSA method. In order to allow fair comparison, we resort to Hassel_C[31] which is an optimized C-based implementation of the HSA method that supports multithreading. Furthermore, Hassel_C uses some algorithmic optimization techniques to boost the execution speed further and reduce the memory footprint. The performance of Hassel_C for reachability test and loop detection on the Stanford network is compared with both single-thread OPC and multithread OPC. The Hassel_C performs the loop detection test and reachability check by launching two separate queries.

We manually verify HSA, single-thread OPC and multithread yield precisely the same result for the reachability and loop detection tests.

We measure the time required for the loop detection and the reachability check. The comparison of the execution time of the HSA method and the OPC is presented in Figure 17. As demonstrated by Figure 17, while the single-thread OPC is more than seven times faster than HSA, the multithread OPC is more than 17 times faster compared with HSA. In fact, in this scenario, the processing time of the single-thread OPC, multithread OPC, HSA is, respectively, 24.40, 10.74, and 179.63 seconds. This result not only shows that the proposed transfer model has a proper parallelization capability but also demonstrates that the single-thread OPC is superior in terms of processing speed to HSA despite its multithread implementation.

In addition, we also evaluate the memory usage of the HSA and the OPC. According to Figure 18, the result shows that HSA takes up 6% memory at running time. The multithread OPC takes up 11%, and the single-thread OPC takes up 0.8% memory. Taking into account the significant gain in terms of the execution time of the multithread approach, this increase in terms of memory consumption is not

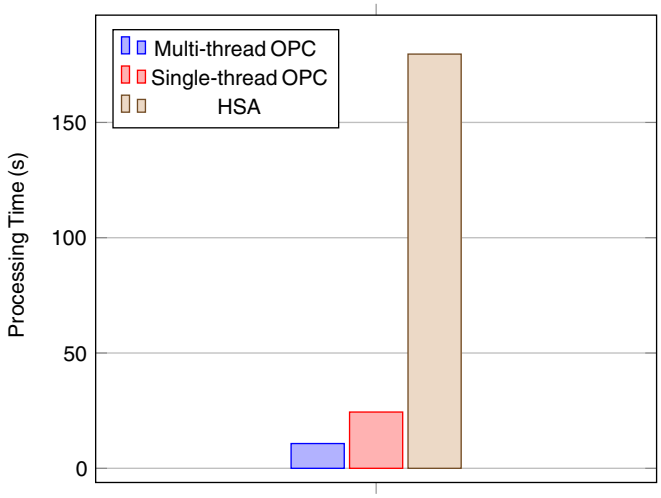


FIGURE 17 Comparison of processing time for probing for the Stanford network

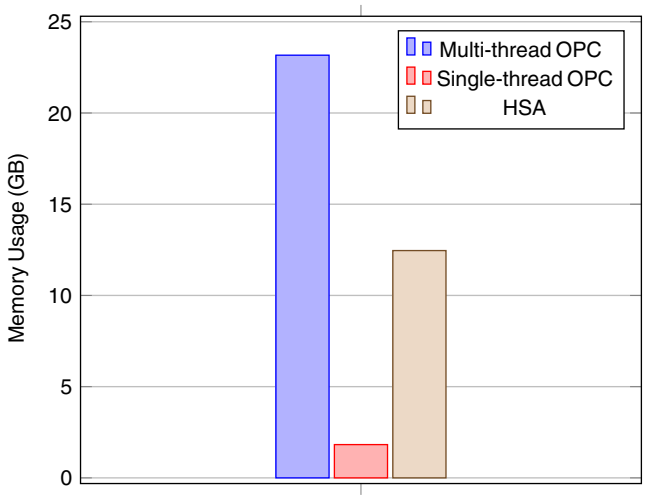


FIGURE 18 Comparison of memory usage for the Stanford network

an issue and can be omitted. Furthermore, the single-thread approach outperforms HSA both in terms of execution time and memory usage.

5.3 | Intraanomaly detection

The "Intraanomaly Detection Engine" is capable of checking the existence of intraanomalies between one unmatched rule and the rest of rules of the flow table. The engine receives one rule and its corresponding flow table and then tries to detect the simple and total anomalies among them.

In this subsection, the execution time of the intraanomaly detection is evaluated for different rule sets based on both single-thread and multithread approaches. By design, the execution time is independent of the type of the anomaly and numbers of detected anomalies. The processing time of intraanomaly detection via single-thread approach for mock setup is presented in Table 8 and Figure 19. As observed in Table 9 and Figure 20, the multithread detection method is remarkably faster than the single-thread one.

The intraanomaly detection algorithm was applied also for the Stanford network. The processing time of single-thread approach for intraanomaly detection is depicted in Table 10 and Figure 21. In addition, the processing time of parallel approach is presented in Table 10 and Figure 22. According to Figures 21 and 22, the multithread approach can improve the performance of intraanomaly detection (Table 11). By comparing the obtained results from Figures 19 to 22, it can be concluded that the parallelizing capability improves the performance since there is

TABLE 8 Single-thread intraanomaly detection processing time for the mock setup

Num. of rules	500	1000	2000	5000	10 000
Simple shadow (ms)	28.83	49.97	115.40	241.66	515.90
Simple generalization (ms)	26.95	45.53	107.62	225.50	449.07
Simple correlation (ms)	49.84	94.16	199.50	419.64	830.60
Simple redundancy (ms)	53.20	120.16	214.28	445.19	890.22
Total shadow (ms)	3.23	5.20	11.93	24.70	48.99
Total generalization (ms)	3.23	5.20	11.94	24.66	49.08
Total redundancy (ms)	0.78	1.26	3.42	7.20	14.36

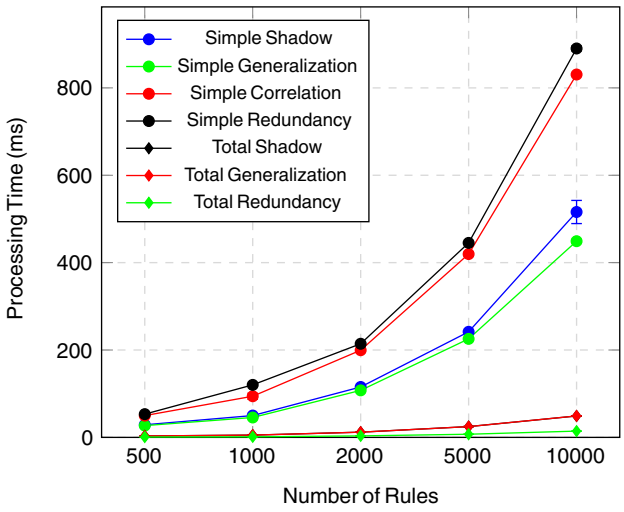


FIGURE 19 Intraanomaly detection processing time with single-thread approach

TABLE 9 Multithread intraanomaly detection processing time for the mock setup

Num. of rules	500	1000	2000	5000	10 000
Simple shadow (ms)	0.024	0.050	0.115	0.287	0.551
Simple generalization (ms)	0.023	0.052	0.102	0.274	0.545
Simple correlation (ms)	0.026	0.051	0.106	0.276	0.550
Simple redundancy (ms)	0.025	0.051	0.104	0.277	0.550
Total shadow (ms)	0.002	0.001	0.009	0.022	0.041
Total generalization (ms)	0.0008	0.005	0.002	0.006	0.013
Total redundancy (ms)	0.003	0.006	0.012	0.027	0.054

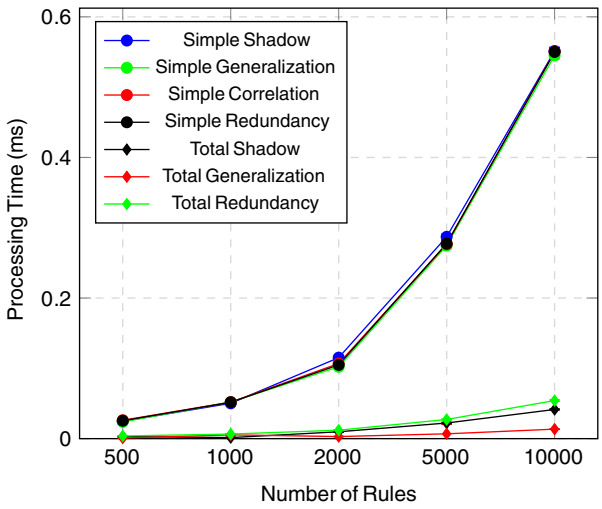


FIGURE 20 Intraanomaly detection processing time with multithread approach

TABLE 10 Single-thread intraanomaly detection processing time for the Stanford network

Num. of rules	91	104	109	116	124	125	142	146	167	175	188	203	204	248	844	870
Simple shadow (ms)	0.0044	0.005	0.0052	0.0055	0.0058	0.0062	0.0069	0.0069	0.0079	0.0082	0.009	0.0094	0.0098	0.0123	0.03772	0.03947
Simple generalization (ms)	0.0158	0.0181	0.019	0.0203	0.0215	0.0221	0.0251	0.0255	0.0296	0.0304	0.0333	0.0354	0.0362	0.0448	0.1516	0.1516
Simple correlation (ms)	0.0189	0.0212	0.0227	0.0243	0.028	0.0265	0.03	0.0334	0.0345	0.0365	0.0399	0.0415	0.0432	0.0528	0.1812	0.1795
Simple redundancy (ms)	0.0079	0.0059	0.0095	0.0101	0.013	0.0113	0.0129	0.0126	0.01266	0.0152	0.0191	0.0219	0.0185	0.0344	0.0762	0.1126
Total shadow (ms)	0.0075	0.0086	0.009	0.0096	0.0103	0.0103	0.0118	0.0121	0.0139	0.0149	0.0161	0.0174	0.0175	0.0212	0.0725	0.0724
Total generalization (ms)	0.0075	0.0086	0.009	0.0096	0.0103	0.0103	0.0117	0.0121	0.0139	0.0149	0.016	0.0173	0.0175	0.0212	0.0725	0.0724
Total redundancy (ms)	0.007694	0.008809	0.009244	0.009816	0.010542	0.010571	0.011994	0.012385	0.014164	0.015182	0.01626	0.017581	0.017772	0.021456	0.071195	0.072779

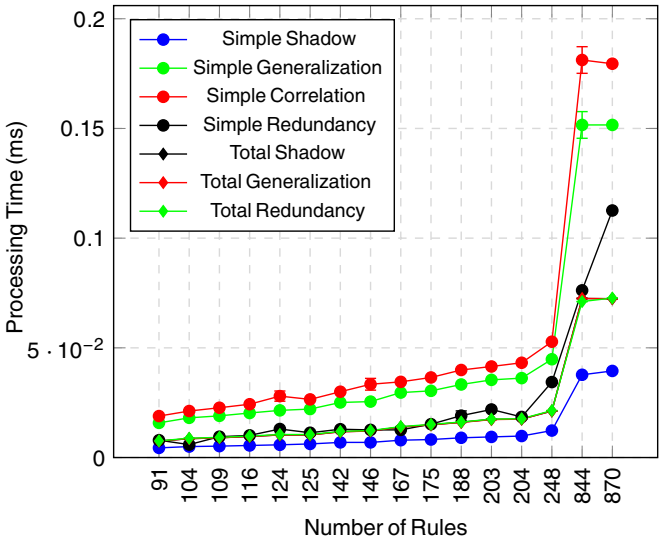


FIGURE 21 Intraanomaly detection processing time with single-thread approach for Stanford network

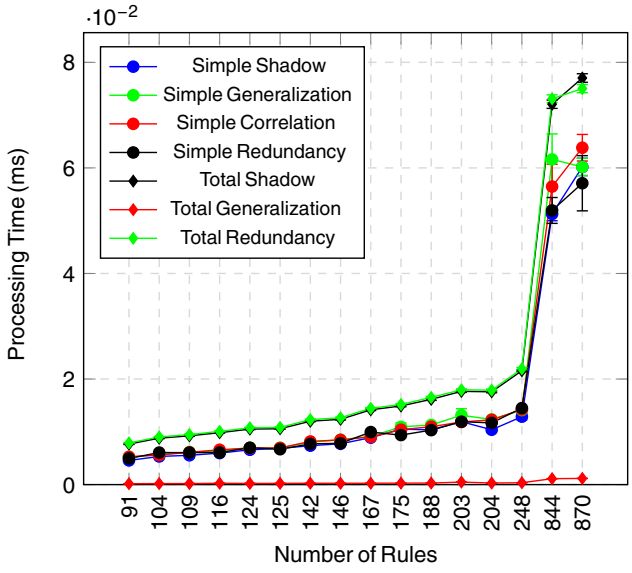


FIGURE 22 Intraanomaly detection processing time with multithread approach for Stanford network

no dependency in the anomaly detection process between any pair of rules (Algorithm 7, Algorithm 7). Moreover, the detection process of the total anomaly is faster due to the fact that the detection process is just applied on some specific rules and not on the whole set of rules as see in Algorithm 8.

6 | CONCLUSION

SDN rules are changed dramatically, which leads to error-prone policies. There have been some valuable studies on rule anomaly detection. However, those studies do not cope with multiaction OpenFlow rules. In this article, we provide comprehensive and generalized anomaly classification and detection methods for SDN that cover multiaction OpenFlow rules. Furthermore, we introduce the taxonomy: invalid and irrelevant anomalies for unmatched rules. An offline method is implemented based on the new definitions. Our detection method has been enhanced to support parallel execution, which makes it a viable solution for troubleshooting large-scale networks. The execution duration of each step of the method: (1) query generation, (2) probing, and (3) anomaly detection has been evaluated thoroughly under different scenarios, and the results are very promising. As future work, we would like to generalize our method for real-time policy checking where rules get inserted and deleted dynamically.

TABLE 11 Multithread intraanomaly detection processing time for the Stanford network

Num. of rules	91	104	109	116	124	125	142	146	167	175	188	203	204	248	844	870
Simple shadow (ms)	0.004579	0.005333	0.005551	0.005962	0.0066	0.006858	0.007376	0.007734	0.008893	0.010503	0.010378	0.011919	0.010386	0.01287	0.051266	0.060183
Simple generalization (ms)	0.005191	0.005688	0.006172	0.006398	0.006971	0.006972	0.008043	0.008502	0.009114	0.010941	0.011302	0.013137	0.012341	0.014122	0.061611	0.060183
Simple correlation (ms)	0.005276	0.005605	0.006109	0.006597	0.006933	0.006946	0.008178	0.008474	0.009037	0.010405	0.010965	0.011866	0.012273	0.01424	0.056464	0.063824
Simple redundancy (ms)	0.004954	0.006074	0.006058	0.00608	0.006981	0.006699	0.007716	0.007829	0.009957	0.009378	0.010326	0.011917	0.011693	0.014504	0.051941	0.057096
Total shadow (ms)	0.007677	0.008794	0.009239	0.009831	0.010517	0.010579	0.012005	0.012369	0.014167	0.014898	0.016153	0.017665	0.017578	0.021574	0.072065	0.077032
Total generalization (ms)	0.000172	0.000188	0.000196	0.000252	0.00022	0.000233	0.000255	0.000246	0.000256	0.000279	0.000291	0.000475	0.000295	0.00035	0.001123	0.001171
Total redundancy (ms)	0.007892	0.009032	0.009468	0.010094	0.010764	0.010838	0.012311	0.012658	0.014454	0.015182	0.016524	0.017956	0.017896	0.021942	0.073109	0.075031

ORCID

Ramtin Aryan https://orcid.org/0000-0002-9392-6704
Anis Yazidi https://orcid.org/0000-0001-7591-1659

REFERENCES

1. Aryan R, Yazidi A, Engelstad PE, Kure Ø. A general formalism for defining and detecting openflow rule anomalies. Paper presented at: Proceedings of the 42nd IEEE Conference on Local Computer Networks Institute of Electrical and Electronics Engineers (IEEE); 2017.
2. McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comput Commun Rev*. 2008;38(2):69.
3. Sherwood R, Gibb G, Yap KK, et al. Can the production network be the testbed? Paper presented at: Proceedings of the OSDI; vol. 10; 2010:1-6.
4. El-Atawy A, Ibrahim K, Hamed H, Al-Shaer E. Policy segmentation for intelligent firewall testing. Proceedings of the 2005 1st Work *Secure Network Protocols* NPSec, held conjunction with ICNP 2005 2005:67–72.
5. Al-Shaer E, Al-Haj S. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. Paper presented at: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration ACM; 2010:37-44.
6. Al-Shaer E, Hamed H. Design and implementation of firewall policy advisor tools. DePaul University, CTI, Tech Rep 2002.
7. Rezvani M, Aryan R. Analyzing and resolving anomalies in firewall security policies based on propositional logic. - 2009 IEEE 13th International Multitopic Conference on INMIC 2009; 2009.
8. Khurshid A, Zhou W, Caesar M, et al. Veriflow: verifying network-wide invariants in real time. Paper presented at: Proceedings of the 10th USENIX Symposium Networked System Des Implement (NSDI 13); 2013:15-27.
9. Kazemian P, Chang M, Zeng H, Varghese G, McKeown N, Whyte S. Real time network policy checking using header space analysis. Proceedings of the 10th USENIX Conference Networked System Des Implement; 2013:99-112.
10. Courcelle B. Graph structure and monadic second-order logic: Language theoretical aspects. Paper presented at: Proceedings of the International Colloquium on Automata, Languages, and Programming; 2008:1-13; Springer.
11. Aryan R, Yazidi A, Engelstad PE. An incremental approach for swift openflow anomaly detection. Paper presented at: Proceedings of the 2018 IEEE 43rd Conference on Local Computer Networks (LCN) IEEE; 2018:502-510.
12. Stanford Backbone. https://github.com/wuyangjack/standford-backbone.
13. Hu H, Han W, Ahn GJ, Zhao Z. FlowGuard: building robust firewalls for software-defined networks. Paper presented at: Proceedings of the 3rd Work Hot Top Software Define Network (HotSDN 2014); 2014:97-102.
14. Kazemian P, Varghese G, McKeown N. Header space analysis: static checking for networks. Paper presented at: Proceedings of the NSDI; vol. 12, 2012:113-126.
15. Xie GG, Zhan J, Maltz DA, et al. On static reachability analysis of IP networks. Paper presented at: Proceedings of the IEEE INFOCOM, vol. 3; 2005:2170-2183.
16. Mai H, Khurshid A, Agarwal R, et al. Debugging the data plane with anteater. *ACM SIGCOMM Comput Commun Rev*. 2011;41:290.
17. Sherwood R, Gibb G, Yap KK, et al. FlowVisor: a network virtualization layer. *Network*. 2009;1:15.
18. Al-Shaer E, Marrero W, El-Atawy A, ElBadawi K. Network configuration in a box: towards end-to-end verification of network reachability and security. Paper presented at: Proceedings of the 17th IEEE International Conference on Network Protocols, 2009. ICNP 2009 IEEE; 2009:123-132.
19. Golnabi K, Min RK, Khan L, Al-Shaer E. Analysis of firewall policy rules using data mining techniques. 2006 IEEE/IFIP **Network** Operations and **Management Symposium**NOMS 2006; 2006:305-315.
20. Flowvisor Project WebPage. https://openflow.stanford.edu/display/DOCS/Flowvisor. Accessed February 25, 2017.
21. Parkinson S, Vallati M, Crampton A, Sohrabi S. Graph BAD: a general technique for anomaly detection in security information and event management. *Concurr Comput Pract Exp*. 2018;30:e4433.
22. Son S, Shin S, Yegneswaran V, Porras P, Gu G. Model checking invariant security properties in OpenFlow. *IEEE Int Conf Commun*. 2013;1974-1979.
23. Hinrichs T, Gude N, Shenker S, Casado M, Mitchell J, Shenker S. Expressing and enforcing flow-based network security policies. *Univ Chicago Tech Rep*. 2008;9:1-20. http://people.cs.uchicago.edu/~thinrich/papers/hinrichs2008design.pdf.
24. Shin S, Porras P, Yegneswaran V, Fong M, Gu G. FRESCO: modular composable security services for software-defined networks. Paper presented at: Proceedings of the Network & Distributed System Security Symposium (NDSS 2013); 2013:1-16. http://www.csl.sri.com/users/vinod/papers/fresco.pdf%5Cnpapers3://publication/uuid/5A7E2F2B-FBAC-48CF-BDCE-0C4A1A4604FB.
25. Porras P, Shin S, Yegneswaran V, Fong M, Tyson M, Gu G. A security enforcement kernel for OpenFlow networks. Paper presented at: Proceedings of the 1st workshop on Hot Topics in Software Defined Networks (HotSDN 2012); 2012:121-126. http://dl.acm.org/citation.cfm?doid=2342441.2342466.
26. Chomsiri T, Pornavalai C. *Firewall Rules Analysis*. New York, NY: Security and Management Citeseer; 2006:213-219.
27. Al-Shaer ES, Hamed HH. Discovery of policy anomalies in distributed firewalls. Paper presented at: Proceedings of the INFOCOM 2004 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 4 IEEE; 2004:2605-2616.
28. OpenFlow Official WebPage. https://www.opennetworking.org/sdn-resources/openflow. Accessed February 25, 2017.
29. Taylor DE, Turner JS. Classbench: a packet classification benchmark. *IEEE/ACM Trans Netw (TON)*. 2007;15(3):499-511.
30. OVS Commands Reference Version 15, 2015. 1032 Elwell Court, Suite 105 Palo Alto, CA. 94303.
31. Hassel code. https://bitbucket.org/peymank/hassel-public/wiki/Home. Accessed February 02 2018.

How to cite this article: Aryan R, Yazidi A, Kure Ø, Einar Engelstad P. A parallel approach for detecting OpenFlow rule anomalies based on a general formalism. *Concurrency Computat Pract Exper*. 2020;e5907. https://doi.org/10.1002/cpe.5907