Automated and Optimized Formal Approach to Verify SDN Access-Control Misconfigurations

Amina Saâdaoui(✉), Nihel Ben Youssef Ben Souayeh, and Adel Bouhoula

Digital Security Research Lab, Sup'Com, University of Carthage, Tunis, Tunisia
{amina.saadaoui,nihel.benyoussef,adel.bouhoula}@supcom.tn

Abstract. Software-Defined Networking (SDN) brings a significant flexibility and visibility to networking, but at the same time creates new security challenges. SDN allows networks to keep pace with the speed of change by facilitating frequent modifications to the network configuration. However, these changes may introduce misconfigurations by writing inconsistent rules for Flow-tables. Misconfigurations can arise also between firewalls and Flow-tables in OpenFlow-based networks. Problems arising from these misconfigurations are common and have dramatic consequences for networks operations. Therefore, there is a need of automatic methods to detect and fix these misconfigurations. Given these issues, some methods have been proposed. Though these methods are useful for managing Flow-tables rules, they still have limitations in term of low granularity level and the lack of precise details of analyzed flow entries. To address these challenges, we present in this paper a formal approach that allows to discover Flow-tables misconfigurations using inference systems. The contributions of our work are the following: automatically identifying Flow-tables anomalies, using the Firewall to bring out real misconfigurations and proposing automatic method to deal with set-field action of flow entries.

These techniques have been implemented and we proved the correctness of our method and demonstrated its applicability and scalability. The first results we obtained are very promising.

Keywords: Flow entries · Flow table · SDN · Misconfigurations · FtDD · Inference system · Direct path · Firewall

1 Introduction

In SDN Network, devices can be programmed via different communication protocols, such as OpenFlow. In fact an openFlow network consists of a distributed collection of switches managed by a program running on a logically-centralized controller. Each switch has a flow table that stores a list of rules for processing packets. Each rule consists of a pattern (matching on packet header fields) and

H. Gao et al. (Eds.): TridentCom 2018, LNICST 270, pp. 96–112, 2019.
https://doi.org/10.1007/978-3-030-12971-2_6

actions (such as forwarding, dropping, modifying the packets, or sending them to the controller). The OpenFlow controller installs or uninstalls rules in the switches, reads traffic statistics, and responds to events. For each event, the controller program defines a handler, which may install rules or issue requests for traffic statistics. Therefore, Open flow and Software-Defined Networking (SDN) can simplify network management by offering programmers network-wide visibility and direct control over the underlying switches from a logically-centralized controller, but at the same time brings new security challenges by raising risks of software faults (or bugs), especially switches misconfigurations. Since companies rely only on the availability of their networks, such misconfigurations are costly. Due to the magnitude of this problem, our goal is to develop a method that allows to automatically identify configuration errors among the set of switches rules which should be well configured with respect to the firewall configuration. This task is challenging due to a number of reasons. First of all, an openflow switch generally comprises thousands of flow entries that are dependent and second flow entries do not always exactly match firewall rules. As an example, consider an enterprise network shown in Fig. 1. We have three switches their configurations are shown in Fig. 3. The firewall configuration that should be implemented is shown in Fig. 2 This example is considered throughout this paper to evaluate our approach to discover flow entries misconfigurations. We can note that the second flow entry fe_2, shown in Switch $S1$, is configured to forward traffic from the machine 172.27.2.3, to switch $S2$ and from this switch traffic will be forwarded to switch $S3$ using flow entry fe_2, then from the switch S_3 this traffic will be dropped using the flow entry fe_6, which is not conform to the requirements of the firewall configuration shown in Fig. 2 (rule r_2 will accept traffic from this source address). Although a misconfiguration is identified between these flow entries, most related studies [1,3,4] did not consider these configuration errors, also, most of these studies did not handle different actions of flow entries, precisely, the action set-Field that allow to modify header packet fields which can influence the path parsed by some packets and consequently the process of detection of misconfigurations. This task is more complex than it appears at first glance especially when a large number of switches and flow entries is deployed.

In this paper, we propose a new approach to discover misconfigurations in real-case openFlow switches configurations already designed, by considering all relations between all flow entries and all possible paths parsed by packets in a given Network. Our approach takes advantages of the interdependency of flow entries modeling their relations in a Flowtable decision diagram (FtDD). Our proposed method could be used also before updates occurred by the controller to verify if changes will induce further misconfigurations. This paper is organized as follows: Sect. 2 presents a summary of related work. Section 3 overviews the formal representation of flow entries, firewall configurations and details $FtDD$ structure. In Sect. 4, we present our inference systems to discover misconfigurations. In Sect. 5, we present first a study of the complexity of our inference systems, and then we address the implementation and evaluations of our tool. Finally, we present our conclusions and discuss our plans for future work.

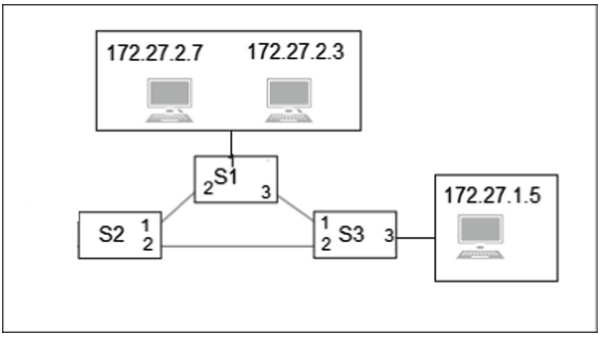

Fig. 1. Network topology

Rule N°	Source	Destination	Port	Action
1	172.27.2.7	172.27.1.5	80	drop
2	172.27.2.0/24	*	*	accept
3	172.27.1.5	*	*	accept
4	*	*	*	drop

Fig. 2. Firewall configuration

Switch1 Configuration

FE N°	Source	Destination	Port	Action
1	*	172.27.1.5	*	Fwd(S3)
2	172.27.2.3	*	*	Fwd(S2)
3	172.27.2.7	*	*	Fwd(S2)
4	172.2.3.7	*	*	Fwd(S3)
5	*	*	*	drop

Switch2 configuration

FE N°	Source	Destination	Port	Action
1	172.27.1.5	*	*	Set-Srce(172.27.1.5,172.27.2.7) && Fwd(S1)
2	172.27.2.0/24	*	*	Fwd(S3)
3	172.27.1.5	*	*	Fwd(S3)
4	172.27.3.7	*	*	Fwd(S3)
5	*	*	*	drop

Switch3 configuration

FE N°	Source	Destination	Port	Action
1	172.27.2.7	172.27.1.5	*	Fwd(port 3)
2	172.27.1.5	172.27.2.7	*	Fwd-Firewall
3	172.27.1.5	*	*	Fwd(S2)
4	172.27.2.3	172.27.1.5	*	drop
5	172.27.3.7	*	*	Fwd(S2)
6	*	*	*	drop

Fig. 3. Switches configurations

2 Related Work

A significant amount of research has addressed configurations analysis and modeling. For example, Some research has focused on firewall misconfiguration detection anad correction ([1,11,17,18]). Also, there was a considerable amount of work on detecting misconfiguration in routing ([2,7,10]). The concept of OpenFlow switch was introduced in [16] and used in different applications. The work

done on OpenFlow switches did not address the problems of switches misconfiguration detection and correction; instead, recently, there have been many verification tools proposed for SDN. Some tools debug controller software or applications, while others check the correctness of network policies.

Controller software or applications verification: In [4] authors propose a tool named NICE which automates the testing of OpenFlow Apps. In fact it allows to find bugs in real applications and to test the atomic execution of system events. But this tool does not guarantee the errors absence and does not allow to check safety properties. Ball et al. propose another tool in [3] named Verifcon that allows to verify the correctness of SDN applications on a large range of topologies and sequences of network events. The limitation of this work is that authors focus on safety properties without verifying the liveness properties of packets (packets must eventually reach their destinations) and also they assume that events are executed atomically ignoring out-of order rule installations.

Network policies verification: Frenetic [8] is a domain-specific language for OpenFlow that aims to eradicate a large class of programming faults. Using Frenetic requires the network programmer to learn extensions to Python to support the higher-layer abstractions. OFRewind [19] enables recording and replay of events for troubleshooting problems in production networks due to closed-source network devices. However, it does not automate the testing of OpenFlow controller programs. Kazemian et al. [12] proposed a method that allows to verify network properties like reachability, by using Header Space Analysis HAS but their work does not allow to check in real-time if network policy still not violated after rules update for example. Netplumber presented in [12] uses a set of policies and invariants to do real time checking. It leverages header space analysis and keeps a dependency graph between rules but it does not allow to model dynamic network behaviors. Hu et al. introduced in [11] Flowgard a new tool that allows to verify the network policy by providing methods to detect and correct firewall policy violations in OpenFlow based networks. FlowChecker [1] applies symbolic model checking techniques on a manually-constructed network model based on binary decision diagrams to detect misconfigurations in OpenFlow forwarding tables. In [13] authors present a tool *VeriFlow* used for verifying network correctness before the rules and logic are implemented in the network devices. The tool will check the changes made to the network for correctness or anomalies before allowing the changes to be deployed. But when large changes in the network happen, VeriFlow is unable to keep up and it is necessary to allow rules to be installed without verification. Instead, the verification process will run in parallel, at the same time as the rules are installed.

The objectives of our work are different. We aim first to automatically identify Flow-tables anomalies, using the Firewall to bring out real misconfigurations and finally, we propose automatic method to deal with set-field action of flow entries. Proving the correctness and completeness of proposed techniques is an unavoidable step. Nevertheless, most existing studies and algorithm ignore to prove these two properties. In our work, by using formal representation and inference systems we proved their completeness and correctness.

3 Formal Specification

Our objective is to discover each misconfiguration with the minimum number of operations. Therefore, we need a formal specification to deal with this problem and also to prove the correctness and completeness of our work. In what follow, we define, formally, some key notions.

3.1 Open Flow Switch Flow Entries

An OpenFlow Switch configuration consists of a flow table, which perform packet lookups and forwarding, and an OpenFlow channel to an external controller. The switch communicates with the controller and the controller manages the switch via the OpenFlow protocol. A flow table contains a set of flow entries of the form $FL = \{fe_i => a_i; 1 =< i <= n\}$; each flow entry consists of match fields fe_i, and a set of actions to apply to matching packets $ai = \{FORWARD, CONTROLLER, set(field1, field2)\ and\ FORWARD, drop\}$, where the action $CONTROLLER$ allows to forward packets to the controller which will filter them using the firewall configuration.

3.2 Firewall Configuration

We consider a finite domain $\mathcal{P}$ containing all the headers of packets possibly incoming to or outgoing from a network.

A simple firewall configuration is a finite sequence of filtering rules of the form $FR = (r_i \Rightarrow A_i)_{0<i<N+1}$. These rules are tried in order, up to the first matching one. A filtering rule consists of a precondition r_i which is a region of the packet's space, usually, consisting of source address, destination address, protocol and destination port. Each right member A_i of a rule of FR is an action defining the behavior of the firewall on filtered packets: $A_i \in \{accept, deny\}$.

3.3 *FtDD* (Flow Table Decision Diagram) of a Path in a Distributed Environment

A flow tables decision diagram of our network, is constructed using the collection of flow entries of different flow tables of different switches in our network switches. Therefore, the $FtDD$ of our network could be represented as follows: $FtDD = \{dp_j; 1 <= j <= m\}$, which is an acyclic and directed graph that has the following properties: There is exactly one node in $FtDD$ that has no incoming edges. This node is called the root of $FtDD_i$. The nodes in $FtDD$ that have no outgoing edges are called terminal nodes. $FtDD_i$ is the union of direct paths dpi. The algorithm used to construct an $FtDD$ is detailed in [9,15]. Each direct path

is represented as follows: $FtDD = \bigcup_{j(i:1 \to m)} dp_i.\ dp_j = dp_j.srce \wedge dp_j.protocol \wedge dp_j.dest \wedge dp_j.flowEntries \wedge dp_j.action$. Where:

- $dp_j.srce$ is the range of source address represents by the direct path dp_j.
- $dp_j.dest$ is the range of destination address represents by the direct path dp_j.
- $dp_j.protocol$ is the range of protocols represented by the direct path dp_j.
- $dp_j.flowEntries$ is the set of flow entries from the flow table configuration that match the domain of packets represented by this direct path. But we have to precise for each rule the flow table that belongs to it.
- $dp_j.action$= = the action of this direct path dp_j. The action of each direct path depends on the actions of each flow entry handled by this direct path from each switches in this path, so we have:
  - $dpj.action = accept$ if all flow entries applied the action forward from the source to the destination.
  - $dpj.action = drop$ if at least one rule applies the action drop to the packets handled by this direct path.
  - $dpj.action = set - Field(field1, field2)\ and\ Fwd(Sk)$ if in this direct path we have a flow entry that apply this action.
  - $dpj.action = Loop$, if the flow handled by this direct path is returned to a switch already exists in the set $dpj.flowEntries$.
  - $dpj.action = CONTROLLER$ if at least one rule applies this action to the packets matched by this direct path, packets forwarded to the controller will be handled by the firewall.

Our current work allows to automatically discover misconfigurations in all switches of our network by considering all relations between all flow entries and by considering also all parsed paths by using $FtDD$. In the next section we discuss our approach to deal with this problematic.

4 Inference Systems

In this work, our goal is to propose an automatic method that supports OpenFlow controller by effectively managing flow-tables entries in dynamic OpenFlow-based networks. To achieve our goal and address this challenge, we seek a solution based of inference systems.

4.1 Inference System for Constructing *FtDD*

The first step is to define a set In-switches composed by couples (S_{in}, I) switches from which the traffic flow first. Where I is source addresses that are linked to the switch S_{in}. The verification in our work is based on firewall requirements; therefore, we use the firewall rules and the network topology to define this set I. Our goal is to construct the $FtDD$. To achieve this goal we propose in Fig. 4 an inference system that presents steps to construct this $FtDD$.

$$Init\ \frac{}{\varnothing, S_{in}, \varnothing, \varnothing}$$

$$Start\ \frac{FtDD, \{fe\} \cup S_{in}, \varnothing, \varnothing}{construct_{FtDD}(FtDD, \{fe\} \cap I), S_{in}, FE_m, \varnothing}\ \begin{array}{l} if(\{fe\} \cap I \neq \varnothing \text{ and } fe.action = Fwd(S_j)) \\ where\ FE_m = \{(fe_m, dp)\ where\ (\{fe_m\} \in S_j \text{ and } \\ \qquad dp = \{fe_m\} \cap \{fe\} \cap I)\ if\ \{fe_m\} \cap \{fe\} \cap I \neq \varnothing\} \end{array}$$

$$Pass\ \frac{FtDD, \{fe\} \cup S_{in}, \varnothing, \varnothing,}{FtDD, S_{in}, \varnothing, \varnothing}\ if\ no\ other\ rule\ applies$$

$$Apply\ \frac{FtDD, S_{in}, (fe_m, dp) \cup FE_m, F}{construct_{FtDD}(FtDD, dp), S_{in}, FE'_m, F'}\ \begin{array}{l} if(dom(dp) \setminus F \neq \varnothing) \\ where \begin{cases} F' = dom(fe_m) \cup F\ if(verify\_Sw(fe_m, F)) \wedge F' = \varnothing\ otherwise \wedge \\ \quad FE'_m = FE_m \cup \{(fe_n, fe_n \cap dp)\ where\ condition\_add\}\} \end{cases} \end{array}$$

$$Pass_{Apply}\ \frac{FtDD, S_{in}, (fe_m, dp) \cup FE_m, F}{construct_{FtDD}(FtDD, dp), S_{in}, FE_m, F}\ if\ \ apply\ \ is\ \ not\ \ applied$$

$$Stop\ \frac{FtDD, \varnothing, \varnothing, \varnothing}{FtDD}$$

Fig. 4. Inference system for constructing FtDD

The rules of this inference system apply to quadruple $(ftdd, S_{in}, FE_m, F)$ where $ftdd$ is the Flow entries decision diagram of the couple (S_{in}, I), FE_m is a temporary variable contains a set of flow entries from different switches in our network that we should parse to get the real path from which packets from sources in the set I passed. F is a temporary variable contains the set of packet matched by rules already parsed. The inference rule *start* allows to parse rules from the switch S_{in} that match the set I, this inference rule allows also to define the set FE_m if the action of the parsed flow entry fe is forward to another switch S_j, therefore this set contains rules from the switch S_j that match the set of packets matched by previous traffic. The rule apply allow to route all traffic according to rules matched and actions $FORWARD$. So the idea implemented by this inference system is as follows: For each flow entry from the switch S_{in}, we verify if its action is to forward to another switch, in this case, we parse flow entries of the new switch until we obtain a flow entry with an action drop, $CONTROLLER$ or a forward to another switch already parsed. Therefore, the condition to add a flow entry to the set of rules to be parsed is described, as follows:

The rule *Stop* is applied when we parse and update all the paths of the set S_{in} (Fig. 5).

$$condition\_add = \begin{cases} fe_n.action = Fwd(S_k) \wedge fe_n \in S_k \wedge \\ (fe_n \cap dp \neq \varnothing) \wedge S_k \notin switches(dp) \end{cases}$$

Fig. 5. ConditionAdd

The function $Construct\_FDD$ is the function used to construct FDD depicted in previous work [9,15]. The flow entries decision diagram of all sets S_{in} is defined as follows: $FtDD = \bigcup ftdd$.

For example if we consider the network topology shown in Fig. 1, we should first start by defining possible inputs I_{in}. We have three sets of possible input addresses:

- $I1 = \{172.27.2.7, 172.27.2.3\}$ which is linked to switch S_1.
- $I2 = \{172.27.1.5\}$ which is linked to switch S_3.
- $I3 = \{*/I1UI2\}$ which is the set of possible input address sources that could income to switch S_2.

We have three sets of possible input address sources. By applying inference system shown in Fig. 4. We will obtain $FtDD$ shown in Figs. 6, 7 and 8 respectively.

In order to prove the correctness and completeness of our approach, we start by the following theorem:

- **Theorem** if $(\emptyset, S_{in}, \emptyset, \emptyset) \vdash^* FtDD$ then, $FtDD$ is correct and complete.
- **Proof** if $(\emptyset, S_{in}, \emptyset, \emptyset) \vdash^* FtDD$ then, $\forall p \in dom(I)$, $\exists dp_i \in FtDD$ where $p \in dom(dp_i)$, because at each Flow table, there is a default flow entry that match all possible packets. Therefore, a packet will match at least on flow entry. Therefore, $FtDD$ is complete. If $(\emptyset, S_{in}, \emptyset, \emptyset) \vdash^* FtDD$, then $\forall p \in dp_i$, $dp_i.FlowEntries$ contains the path parsed by the packet p. In fact, at each step, and if the flow entry match the packet p, and this flow entry is not masked by previous traffic (the set F in our inference system), then one of the inference rules *Apply* or *Start* is applied, and these inference rules will apply

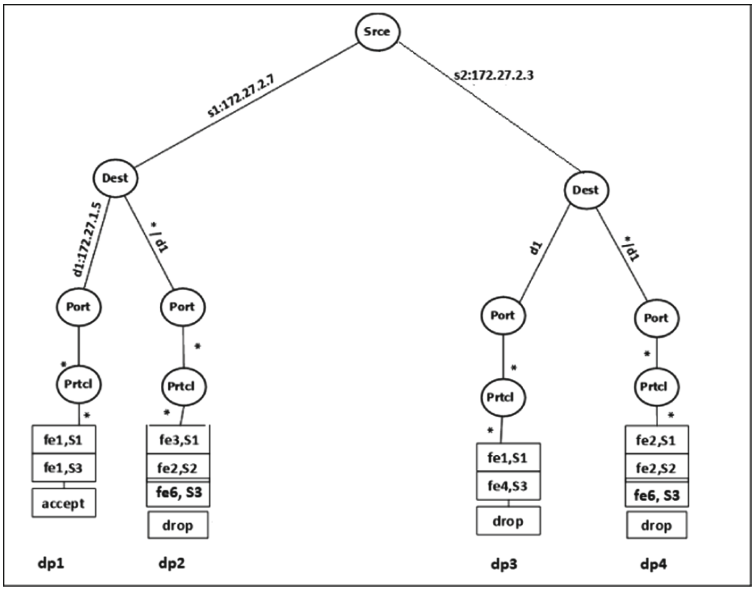

Fig. 6. FtDD of I1

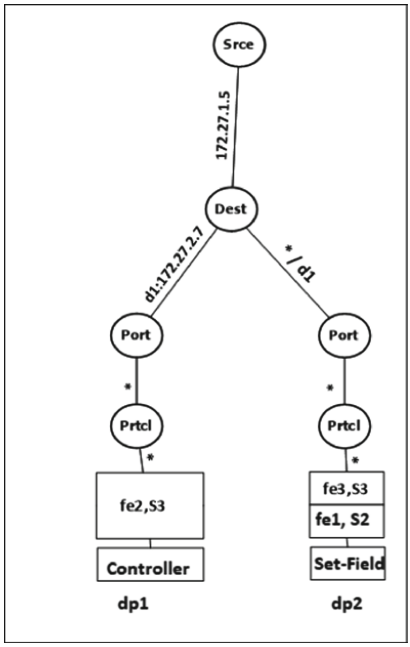

Fig. 7. FtDD of I2

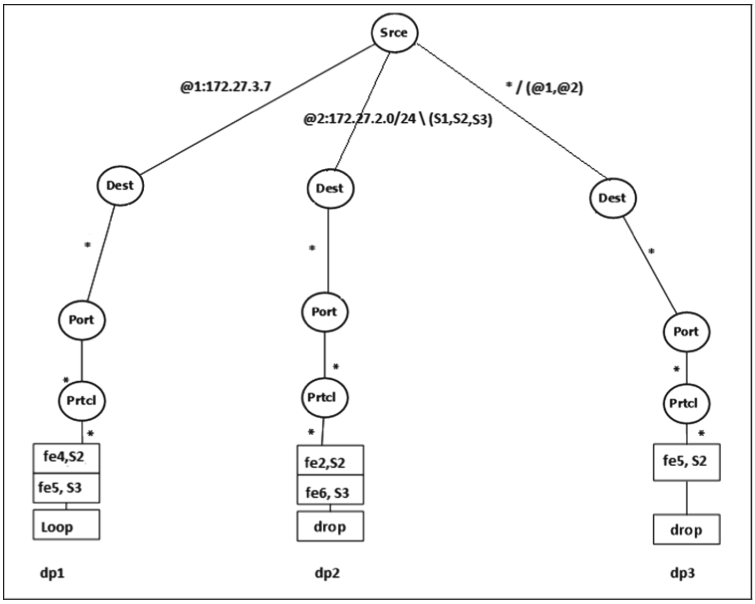

Fig. 8. FtDD of I3

the function $Construct_F DD$ which will add the flow entry applied to the direct path $dp_i.FlowEntries$. Therefore, $dp_i.FlowEntries$ contains exactly all flow entries parsed by the packet p. Then, $dp_i.FlowEntries$ is equivalent to the SDN path of the packet p. Therefore, our $FtDD$ is correct.

4.2 Inference System for Dealing with Set-Field Actions

For a flow entry, we must consider various Set-Field actions, which can rewrite the values of respective header fields in packets that can affect the process of verification. Therefore, before constructing $FtDD$ we have to analyze the impact of these modifications on the flow entries.

The inference system shown in Fig. 9 allows to find and assign effective actions to direct paths that have the action set-field.

In our work we are interested in discovering switches misconfigurations; therefore, knowing the effective action applied on each direct path is an unavoidable step. Our inference system is applied on four variables (shown in Fig. 9), The first one is the set $DP - Set$ which contains all direct paths in our $FtDD$ where actions of these direct paths is equal to "Set-Field(Field1, Field2) and Forward(Sk)": $DP - et = \{dp \in FtDD,\ dp.action = Set - Field(Field1, Field2) \wedge Forward(Sk)\}$. We should find the real action applied by these direct paths. The second one is our $FtDD$ constructed using the inference system defined in the previous section. The Third component dp_match contains all direct paths from $FtDD$ that match the same packets as a given direct path. The main inference rule in this inference system is $update\_FtDD$, it allows to update $Ftdd$ by assigning the effective action applied a given direct path. In fact, for each direct path from the set DP-Set we try to find this action by verifying if direct path that match the modified direct path (i.e., we modify field1 by field2) and have the switch S_k in their path (dp.flowEntries contains a flow entry from the switch S_k) have all the same action, if it is the case we assign this effective action to the direct path otherwise we consider the action as **UNDEFINED** (This indicative will help us to find misconfigurations in the next steps of our work). We have to precise that the new direct paths of our set Dp-match could contain other

$$Init\ \frac{}{DP_{Set}, FtDD, \varnothing, \varnothing}$$

$$Parse\ \frac{dp_{set} \cup DP_{Set}, FtDD, \varnothing, \varnothing}{DP_{Set}, FtDD, dp_{set}, DP_{match}} \quad where\ DP_{match} = \{dp \in FtDD\ where\ dp \cap modify - Field(dp_{set}) \neq \varnothing \wedge S_k \in dp.flowEntries\}$$

$$Update - FtDD\ \frac{DP_{Set}, FtDD, dp_{set}, DP_{match}}{DP_{Set}, FtDD', \varnothing, \varnothing} \quad if(\forall dp \in DP_{match}\ \ dp.action \neq set - Field(f1, f2))\ where\ FtDD' = set - action(FtDD, dp_{set}, Action(DP_{match}))$$

$$Pass\ \frac{DP_{Set}, FtDD, dp_{Set}, DP_{match}}{DP_{Set}, FtDD, \varnothing, \varnothing} \quad if\ no\ other\ rule\ applies$$

$$Success\ \frac{\varnothing, FtDD, \varnothing, \varnothing}{FtDD} \quad if(\forall dp \in FtDD, dp.action \neq undefined)$$

$$Failure\ \frac{\varnothing, FtDD, \varnothing, \varnothing}{Failure} \quad if\ success\ is\ not\ applied$$

Fig. 9. Inference system for dealing with set rules

direct paths that have the action set-field, therefore in this case we will re-add the direct path dp-set to the set DP-Set and we will find all applied actions recursively. The rule Success will be applied if after updating $FtDD$ all actions are defined and the inference rule Failure will be applied otherwise.

We used two functions in this inference system:

- $modify - Field(dp - set)$: This function allows to modify fields of the direct path dp-set by replacing $field1$ by $field2$.
- $Action(DP{-}match)$: This function returns the action applied by direct paths in the set $Dp{-}match$, if all the direct paths apply the same action, otherwise, it returns **UNDEFINED**.
- $Set - action(FtDD, dp - set, act)$: This function allows to update $FtDD$ by assigning the action act to the direct path $dp - set$.

If we consider our example presented in the Sect. 1, we should find actions of different direct paths that have the action $(set - Field, Fwd(Sk))$. In our case we have one direct path:

- $dp2$ in $FtDD2$ shown in Fig. 7: By applying our inference system, we should find different direct paths in $FtDD$ that match the modified direct path $dp2$ (i.e., by replacing 172.27.1.5 by 172.27.2.7 in the field source address) and have a flow entry applied by S_3 in their field $dp.FlowEntries$. In our case we have direct path $dp2$ from $FtDD1$ and the action of this direct path is drop therefore we update our $FtDD$ by assigning the action $drop$ to the direct path $dp2$ from $FtDD2$.

We write $C \vdash_{SetField} C'$: C' is obtained from C by application of one of the inference rules of Fig. 9.

4.3 Inference System for Discovering Access-Control Misconfigurations

We have two types of misconfigurations: Total and partial misconfigurations:

- TMC: A direct path $dp_i \in FtDD$ is totally misconfigured iff it the packets mapped by this path apply a different action as applied in the security policy FC on these packets.
- PMC: A direct path $dp_i \in FtDD$ is partially misconfigured iff **some** packets mapped by this path apply a different action as applied in the security policy SP on these packets.

In Fig. 10, we propose an inference system to discover total and partial misconfigurations. Inference rules are applied on quadruple $(FtDD, TMC, PMC, dp_v)$, where $FtDD$ is the set of all flow entries decision diagrams of all paths in our network. TMC and PMC are the sets of total and partial misconfigurations respectively. dp_v is the direct path to be verified.

The inference rule parse allows to define the direct path to be verified. In most cases it is the direct path dp_i but in some cases when the dp_i.path contains a flow

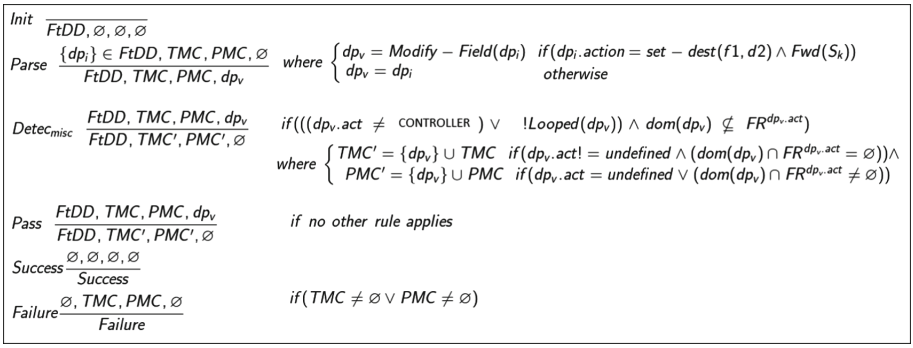

Fig. 10. Inference system for discovering misconfigurations

entry that have the action $set - Field$ where field is a destination address, the direct path to be verified is the direct path modified by replacing the destination address with the new one.

The main inference rule in this system is $Detect\_misc$, it deals with each direct path and compares the domain of this direct path with the set of packets of the firewall configuration that applies the same action as this direct path. If it is partially or not included by this set then we have a partial or a total misconfiguration. And if the action of the direct path is undefined then we consider this direct path partially misconfigured. The $Success$ rule is applied when we parse all direct paths of all $FtDD$ in our network without identifying a misconfiguration (total or partial). Failure is applied when at least one configuration error is identified.

For example, if we consider the network topology shown in Fig. 1 and once ensured that all direct paths have an assigned action, we proceed to the discovering of misconfigurations using our inference system. We parse all paths of $FtDD$, for each path we verify if we have an effective misconfiguration, as we explained in the previous section we have three sets of possible input addresses:

- $I1 = \{172.27.2.7, 172.27.2.3\}$ which is linked to switch S_1.
- $I2 = \{172.27.1.5\}$ which is linked to switch S_3.
- $I3 = \{*/I1UI2\}$ which is the set of possible input address sources that could income to switch S_2.

And for each $I_i \in I_{in}$, we have an $FtDD$ shown in Figs. 6, 7 and 8 respectively.

For $FtDD1$: For packets incoming from the set I_1, we have three total misconfigurations, in direct paths dp_2, dp_3 and dp_4, they apply the action $drop$, while the firewall configuration applies the action $accept$ to packets mapped by the domain of these three direct paths. We have also a partial misconfiguration, in dp_1, in fact, according to the firewall configuration shown in Fig. 2, packets that match $dp1$ and have a port number equal to 80 should be rejected, but dp_1 accepts all packets, even packets that match this port number, Therefore we should detect a PMC in this direct path.

For $FtDD2$: In this $FtDD$, There is no misconfiguration in dp_1, the action applied is $CONTROLLER$, packets will be forwarded to the firewall which is perforce with respect to the firewall. And for the direct path dp_2, there is a total misconfiguration, in fact the action assigned to the direct path dp_2 using the inference system shown in Fig. 9 is drop and the action applied by the firewall to these packets is *accept*, therefore we have one total misconfiguration in this direct path.

For $FtDD3$: For the direct path dp_1 we cannot make a decision because packet parsed by this direct path will be forwarded from S_2 to S_3 which will by his turn forward it again to S_2, therefore this direct path contains a $LOOP$ and no final decision is made. We have a TMC in dp_2, all packets matched by this direct path have a different action as applied in the firewall.

In order to prove the correctness of our approach, we start by the following definition:

- **Definition** $FtDD$ is called *misconfiguration-free* if and only if $\forall dp \in FtDD$, dp verifies these two conditions:
- (1) $dp.action \neq UNDEFINED$.
- (2) $dom(dp) \subseteq FW^{dp.action}$.
- **Theorem** if $FtDD$ is *misconfiguration-free* and $(DP_{Set}, FtDD, \emptyset, \emptyset) \vdash^*_{SetField} Success$ then, $(FtDD, \emptyset, \emptyset, \emptyset) \vdash^*_{detectMisc} Success$.
- **Proof** $FtDD$ is misconfiguration-free, then $\forall dp \in FtDD$, dp applies the same action as defined in FC, $dom(dp) \subseteq FW^{dp.action}$. It follows that at each step we apply first the inference rule $Parse$ to define the direct path to be verified dp_v, then for this direct path we try to apply the inference rule $Detect\_misc$, or $dom(dp) \subseteq FW^{dp.action}$ and $(DP_{Set}, FtDD, \emptyset, \emptyset) \vdash^*_{SetField} Success$ it means that $\forall dp \in FtDD$, $dp_v.action \neq undefined$, therefore, the precondition of the inference rule is not verified. It follows that in all steps $Pass$ inference rule is applied, i.e., $TMC = \emptyset$ and $PMC = \emptyset$, therefore $(FtDD, \emptyset, \emptyset, \emptyset) \vdash^*_{detectMisc} Success$.

4.4 Inference System for Extracting Accepted Denied

In Fig. 11, we propose an Inference system that presents necessary and sufficient steps for extracting accepted and denied packets from a firewall configuration FR. We extract the accepted and denied packets before and after removing each rule from the firewall configuration, two cases can be faced:

- Case1: FR^{accept} (before removing r_i) is equal to FR^{accept} (after removing r_i) and FR^{deny} (before removing r_i) is equal to FR^{deny} (after removing r_i): In this case, we can remove r_i safely without altering the firewall behavior.
- Case2: FR^{accept} (before removing r_i) is different from FR^{accept} (after removing r_i) and/or FR^{deny} (before removing r_i) is different from FR^{deny} (after removing r_i): in this case we should maintain r_i in the configuration file.

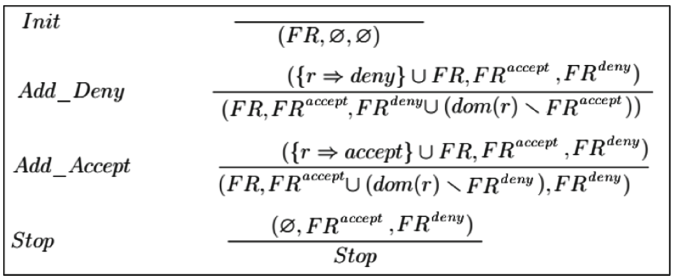

Fig. 11. Inference system for extracting accepted and denied packets

5 Implementation and Experimental Results

5.1 Implementation

We used all-in-one pre-built virtual machine, built by SDN Hub [5]. Wich is a Ubuntu image that has a number of SDN software and tools installed, like: SDN Controllers: OpenDaylight with support for Openflow 1.2, 1.3 and 1.4, and LINC switch. Mininet to create and run example topologies. This pre-built virtual machine contains also a JDK 1.8 and Eclipse, which allows us to easily integrate our solution.

The topology as shown in Fig. 1 is built from a Python program which uses the topology files to build the topology in the controller. The following command allows to build the configuration from the file TOPOTEST:

– $ubuntu@sdnhubvm: /mininet/examples\$\ sudo\ mn\ --custom\ topotest.py\ --topo\ toptest.$

For example to add the flow entry fe1 to the switch S1 we use the following command:

– $sh\ ovs-ofctl\ add-flow\ s1\ priority = 500, nw\_dst = 172.27.1.5, actions = 3.$

We implemented the techniques and inference systems described earlier in a software tool, using a Boolean satisfiability (SAT) based approach. This approach reduces the verification problem into Boolean formula and checks its satisfiability. In our case, in order to verify if a direct path is partially or totally misconfigured, we verify if the domain of the direct path reduced into Boolean formula is included or not in the domain of the firewall configuration reduced into two sub-domains FR^{deny} and FR^{accept} as explained in Sect. 4. So, our formalism for specifying the flow entries and the firewall configuration is a Boolean-based specification language. We have chosen also the Java developing language. On the other hand, the verification of the satisfiability of Boolean expressions is performed using Limboole [14]. This tool allows to check satisfiability respectively tautology on arbitrary structural formulas and not just satisfiability for formulas in conjunctive normal form (CNF), and can handle large set of non-quantified Boolean clauses in reasonably good time.

5.2 Complexity

For n rules in each flow table, there can be a maximum of $2n-1$ outgoing edges for a node. Therefore, the maximum number of paths in a constructed FtDD is $(2n-1)^d$, where d is the number of fields in each flow entry. After the construction of $FtDD$ the discovering of misconfigurations process, explained in Sect. 4, is done on direct paths elements $dp_i.FlowEntries$ and $dp_i.action$. Therefore, for this inference system, the complexity (without counting the elementary functions) is equivalent to the complexity of operations in an ordered list and equal in this case to the complexity of parsing a list which is equal to $O(m)$ (where m is the size of a set). Thus, in our case, the complexity of this inference system is equal to $O(n^d)$, where d is the number of inspected fields. Given that d is typically small (generally we have 4 or 5 fields) our inference systems have a reasonable response time in practice. The next section confirms the above remarks.

5.3 Experimental Results

We have also conducted a set of experiments to measure the performance of our inference systems. The experiments were run on an Intel Dual core 1.6 GHz with 2 Gbyte of RAM. It is supposed that we have $IPv4$ addresses with net-masks and port numbers of 16 bit unsigned integer with range support. Figure 12 summarize our results. We consider time treatment factor that we review by varying the number of switches and flow entries. In overall terms, we consider the average processing time, in seconds, of the main procedures of FtDD construction, dealing with set Set-Field direct paths (dp_i that have action equals to $set-Field$) and $FtDD$ misconfigurations detection. At the end, our tool proved a stable performance showing acceptable processing time to the treatment of complex combination of filtering flow entries.

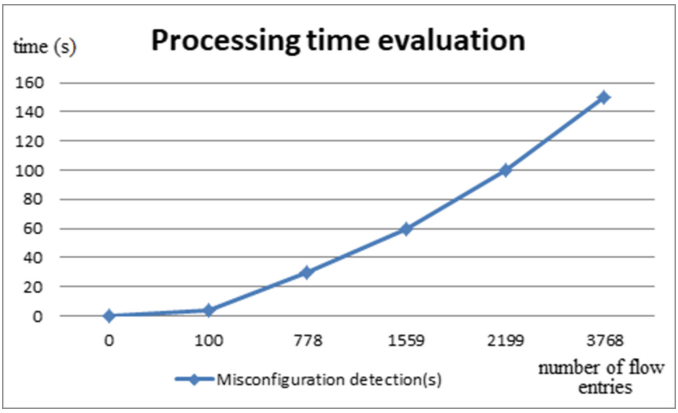

Fig. 12. Processing time evaluation

6 Conclusion

We presented in this paper a set of inference systems to automatically analyze, detect OpenFlow switches misconfigurations. More precisely, our proposal is an offline tool intended for discovering these misconfigurations by using a formal method and a data structure (FtDD), this tool can be used periodically or before updates on Flow tables occurred by the controller to verify if changes will induce further misconfigurations. The advantages of our proposal are the following: First, The detection approach is optimal, using the minimum number of operations. Second, we considered all flow entries of different switches, all paths, all actions of our switches. Third, we analyze also all modifications that can occur on packets if actions set-Field are used, which is not considered by all previous work. Fourth, we proved the correctness and completeness of our approach. While the current approach primarily focuses on discovering switches misconfigurations, in our future work, we plan to automatically resolve these misconfigurations. We are also interested in developing a tool that allows to perform automatically all proposed techniques and test this tool on Cisco Open Network Environment for Government [6] which is a comprehensive solution designed to help government network infrastructures become more open, programmable, and application-aware.

References

1. Al-Shaer, E., Al-Haj, S.: Flowchecker: configuration analysis and verification of federated openflow infrastructures. In: 3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig 2010, Chicago, IL, USA, 4 October 2010, pp. 37–44 (2010)
2. Alimi, R., Wang, Y., Yang, Y.R.: Shadow configuration as a network management primitive. In: Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, 17–22 August 2008, pp. 111–122 (2008)
3. Ball, T.: Vericon: towards verifying controller programs in software-defined networks. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, 09–11 June 2014, pp. 282–293 (2014)
4. Canini, M., Venzano, D., Peresíni, P., Kostic, D., Rexford, J.: A NICE way to test openflow applications. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, 25–27 April 2012, pp. 127–140 (2012)
5. All-in-one sdn app development starter vm (2018)
6. Cisco open network environment for government (2018)
7. Feamster, N., Balakrishnan, H.: Detecting BGP configuration faults with static analysis (awarded best paper). In: Proceedings of 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), Boston, Massachusetts, USA, 2–4 May 2005 (2005)
8. Foster, B., et al.: Frenetic: a network programming language. In: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, 19–21 September 2011, pp. 279–291 (2011)

9. Gouda, M.G., Liu, A.X.: Structured firewall design. Comput. Netw. **51**(4), 1106–1120 (2007)
10. Griffin, T., Wilfong, G.T.: On the correctness of IBGP configuration. In: Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 19–23 August 2002, Pittsburgh, PA, USA, pp. 17–29 (2002)
11. Hu, H., Han, W., Ahn, G.-J., Zhao, Z.: FLOWGUARD: building robust firewalls for software-defined networks. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN 2014, Chicago, Illinois, USA, 22 August 2014, pp. 97–102 (2014)
12. Kazemian, P., Chan, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: NSDI, pp. 99–111. USENIX Association (2013)
13. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Brighten Godfrey, P.: Veriflow: verifying network-wide invariants in real time. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, 2–5 April 2013, pp. 15–27 (2013)
14. Limboole sat solver (2018)
15. Liu, A.X., Gouda, M.G.: Diverse firewall design. IEEE Trans. Parallel Distrib. Syst. (TPDS) **19**(8), 1237–1251 (2008)
16. McKeown, N., et al.: Openflow: enabling innovation in campus networks. Comput. Commun. Rev. **38**(2), 69–74 (2008)
17. Saadaoui, A., Ben Youssef Ben Souayeh, N., Bouhoula, A.: Formal approach for managing firewall misconfigurations. In: IEEE 8th International Conference on Research Challenges in Information Science, RCIS 2014, Marrakech, Morocco, 28–30 May 2014, pp. 1–10 (2014)
18. Saâdaoui, A., Ben Youssef Ben Souayeh, N., Bouhoula, A.: FARE: fdd-based firewall anomalies resolution tool. J. Comput. Sci. **23**, 181–191 (2017)
19. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A.: Ofrewind: enabling record and replay troubleshooting for networks. In: USENIX Annual Technical Conference. USENIX Association (2011)