A Comprehensive Solution for the Analysis, Validation and Optimization of SDN Data-Plane Configurations

Wejdene Saied, Faouzi Jaidi and Adel Bouhoula
Digital Security Research Unit, (Sup'Com)
University of Carthage, Tunisia
{wejdene.saied, faouzi.jaidi, adel.bouhoula}@supcom.tn

***Abstract*—Software Defined Networking (SDN), as an emerging paradigm, offers a centralized control platform by disassociating the forwarding process of network packets (data plane) from the routing process (control plane). However, the distributed state of the Openflow rules across various flow tables and the involvement of multiple independent rules writers may lead to problems of inconsistencies and conflicts within configurations at the infrastructure level. To tackle these issues, we propose, in this paper, an offline approach to fix violations at data plane side and a fine-grained control of SDN switches flow tables. Our solution considers Flow entries Decision Diagram (FeDD) as data structure and relies on formal techniques for analyzing the policy defects and resolving misconfigurations. It allows ensuring that the operator's policies are correctly applied in an optimal way. The implemented prototype, on top of OpendayLight, of our solution and experimentations, based on a real network configurations topology, demonstrate the scalability and applicability of our approach.**

***Index Terms*—Software Defined Networking, Security Policy, Invariants Detection, Flow entries Decision Diagram, SDN Flow Tables Analysis.**

I. Introduction

Most operators check configurations device by device with an ad-hoc way in order to debug traditional network faults. However, network existing tools are unable to automatically detect, locate and repair the root causes. To solve these challenges, Software Defined Networking (SDN) proposes the decoupling of data and control planes in network equipments. This enables independent development of their equipments and a centralized control platform where operators can statically verify network policies. However, many errors caused by switch software bugs and external modification [6, 15] bring forth new security challenges. We focus our study on recent approaches to verify the correctness of network configuration at data plane side (i.e proposals allowing to detect and correct misconfigurations at the Openflow switch level). Most of existent (related) works generate probe packets to check the existence of rules at switches without verifying additional network properties (e.g., access control). These security properties are dependent on paths under frequent network updates or reconfigurations. Other works [12, 16, 18], are only able to simply raise alarms to indicate some violations to users, but cannot provide an automatic violation resolution.

978-3-903176-31-7 © 2020 IFIP

To make our discussion concrete, we consider an example of network topology, shown in Fig.1, with three switches, three hosts, and a simple firewall application used to deploy the security policy. We assume in our study that the SDN controller program is correct and the firewall rules are consistent. One such challenge is introduced by the feature of packet modification bypassing a firewall. In the following scenario, we demonstrate a detection of indirect access violation due to modification of field values: a packet from the host 191.55.3.4 arrives at switch S1, it matches the first rule that sends it to switch S3 after replacing its source IP address with the new 191.55.3.9. Then, the switch S3 drops this packet after applying its matching first rule. However, this flow must be forwarded to the destination host 191.55.7.2 according to the second firewall rule. To overcome the limitations of existing

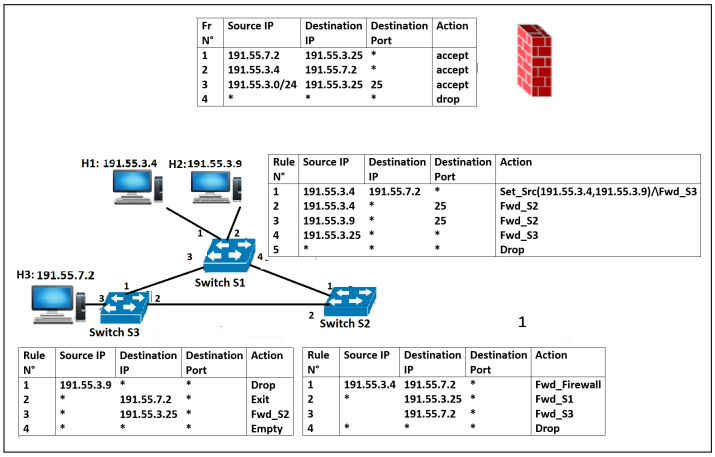

Fig. 1. Network Topology Case of Study.

approaches, we proposed to use in [5] a graph-based model, called Flow entries Decision Diagrams (FeDD), to schematize the relations between intra and inter switch filtering rules. This model allows detecting some network properties violations like blackholes and loops forwarding. However, this prior work is insufficient to meet the requirements of today's operators: a tool that ensures the operator's configurations will correctly reflect on packet forwarding paths. More, the defined model

deals only with defects detection and does not focus on defects analysis and resolution. Given these limitations, we think a missing part in our previous work that can enhance the security at SDN data plane configurations. We focus, in the current paper, on a complementary solution, that enhances the previous model, allowing: (i) the identification of additional invariants like partial and entire access violations with regards to the firewall security policy; and (ii) the correction of all the detected defects and faulty rules while ensuring the switch configurations accuracy and correctness in an optimal way.

Therefore, the major contributions of this paper are summarized as follows:

1) We propose a method to investigate access violation kinds from FeDD analysis: entire and partial violations by referring to the firewall application to bring out concrete switches misconfigurations.
2) We propose fine-grained resolution mechanisms to correct each discovered security invariant in the first step. This process should be accurate, correct and effective by applying different controls (such as modifying some fields of faulty rules, removing some rules, etc.) while respecting the compliance of switch configurations regarding the firewall security policy and without increasing the configurations complexity.
3) To ensure a high level of surety, we formally specified (via a set of inference systems) and proved the correctness and completeness of our proposal.
4) We conducted several experimental results and evaluations that highlight the efficiency, effectiveness and scalability of our approach.

This paper is organized as follows: Section 2 presents a summary of related works. Section 3 overviews background technologies and security challenges. In Section 4, we detail our approach . In Section 5, we address the implementation and evaluation of our solution. Finally, we present our conclusions and discuss our plans for future work.

II. Related Works

Previous efforts on automatic network debugging addressed the correctness of network configurations [7, 11, 6, 14, 20]. However, despite the fact that the SDN controller program and the configurations are correct, the data plane may show misconfigurations due to switch software bugs [22, 24] or malicious attacks [4]. Existing verification tools can only ensure network correctness at the controller side, but cannot guarantee the correctness of rules at flow tables or data plane behaviors. The data plane verification tools are classified into three categories as follows [3]:

Network policies verification : Anteater [14] is a tool that analyzes the data plane state of network devices by encoding switch configurations as boolean satisfiability problems (SAT) instances. Veriflow [13] can perform reachability checking in real time. FlowChecker [20] identifies intra-switch misconfigurations within a single flow table. NetPlumber [12] checks incrementally the compliance of state changes and use Header Space Analysis (HSA) to capture all possible data paths via the plumbing graph. Hu et al. [11] propose the FlowGuard tool for building SDN firewalls, but, it cannot monitor dynamic packet modifications. Authors in [9] further extended the work of FlowChecker for adjusting the structure of multiple flow tables by treating the table as the location of the state instead of the device to check the flow table pipeline misconfiguration. However, the result only returns a single counterexample for the violation, which is hard to be used to analyze the reason for failures. Li et al. introduced the field transition rules into VeriFlow for defending covert channel attacks [8]. However, the header change rules still cannot take action in the forwarding graphs for verifying the reachability. A recent tool, called FlowMon [7], addresses challenges created by the inter-reaction of flow path and firewall authorization space. However, FlowMon cannot detect indirect violations caused by rule dependencies. Therefore, these tools cannot detect inter-federated switches inconsistencies, or packet forwarding. More, the packet transformations are not efficiently handled and many invariants like access violations cannot be checked incrementally.

Controller software verification : Canini et al. [23] present the NICE tool which checks the correctness of SDN controller but it cannot guarantee the absence of errors. More, no correction approach or update inter-switches is proposed after bugs detection. Besides, only the basic invariants are detected. OFRewind [24] enables recording and replaying of troubleshooting for the network. However, it does not automate the testing of Openflow controller programs. Authors, in [1], propose a method for automatic verification of packet reachability by automatically generating logical formulas for reachability verification. However, it cannot handle other more complex policies such as access violations and loop forwarding. Authors, in [2], adopt the concept of atomic predicates and the parallel process computational framework Spark to verify data plane properties. However, they don't propose the resolution mechanism of these defects.

Packet trajectory tracers and data plane testing tools : ATPG [17] generates automatically test packets by injecting network probes. However, it cannot localize the faulty rule. More, this tool does not dictate how these probes should be constructed. ATPG is limited to detect only liveness properties. The highlight of VeriDP [21] is that in order to detect the flow table inconsistencies, it uses the Bloom-filter-based tagging method. However, this approach doesn't incorporate all Set-Actions in the flow tables. These approaches do not include all types of actions and can detect only some basic invariants. In our prior work [5], we propose a new approach for a deep and automated data plane analysis with consideration of flow rules dependencies. However, it is limited to discover some reachability issues such as forwarding loops and blackholes. In addition, we do not propose any method to correct the faulty rules after localizing misconfigurations.

At the end, the major limitations of these works consist in simply preventing users from possible anomalies, but it cannot provide a fine-grained violation resolution. Also, they ignore rule dependencies and some invariants within security

constraints, such as firewall policies, for compliance checking. Unlike recent work that provides a manual invariant resolution process that can trigger possible anomalies, our approach allows the administrator to automatically correct detected defects while ensuring that SDN data plane is continuously compliant with the security policy deployed in the firewall application.

III. Background

In what follows, we formally define some key notions to explain our approach.

A. Security Policy

A security policy *SP* represents a collection of all packets either allowed or denied by the firewall rules. We consider two sets, SP_d and SP_a where SP_a consists of packets accepted to pass through the set of directives *SP* and SP_d is the subset of denied packets. In this paper, we suppose that *SP* is consistent,i.e.$SP_d \cap SP_a = \emptyset$.

B. Flow Policy

Openflow-enabled devices support the abstraction of a flow table, which is manipulated by the Openflow controller. When a packet arrives at the OpenFlow switch from an input port, it is matched against the flow table to determine if there is a matching flow entry. Formally, a flow entry is $Ft = \{r_i; 1 \leq i \leq n\} = \{f_i \Rightarrow action_i; 1 \leq i \leq n\}$ where $f_i =< @sourceIP, @destinationIP, portDest >$ and $action_i$=$\{Set\_Field \wedge Forward, Forward, Drop, Empty, Controller\}$. The action *Controller* forwards packets to the controller which in turn will filter according to the security policy.

C. FeDD Description

In this paper, we referred to the data structure used for multiple switches, called Flow entries Decision Diagrams (FeDDs) and built from a set of rules in the switch configurations. A FeDD is an acyclic and directed graph that has exactly one node, called the root. A directed path from the root to a terminal node is called a decision path dp_i. The algorithm used to construct a FeDD is detailed in [5]. A decision path dp_i is depicted as follows:
$dp_i = (dp_i.S) \wedge (dp_i.D) \wedge (dp_i.P) \wedge (dp_i.Sid.r) \wedge (dp_i.r.id) \wedge (dp_i.r.actions)$ where:

- dp_i.S, dp_i.D, dp_i.P are the domain of 3-tuple fields (Source IP address, Destination IP address and Port destination) matched by the direct path dp_i.
- dp_i.Sid.r is the identifier of the current switch that owns the rule matching the domain of packets in the dp_i.
- dp_i.r.id identifies the rule overlapped with the packets domains represented by this dp_i.
- dp_i.r.A is the action of each direct path that depends on the actions of each flow entry handled by this direct path from every switch in this path. It can be *Exit, Drop, fwd_nextSw, Empty or Controller*.

All FeDD based models convert the switch flow tables into a flow entries decision diagram. Therefore, FeDD of our network is : $FeDD = \cup_i FeDD_i = \cup_{ik:1..n} dp_k$

D. Security Challenges

Openflow switch misconfigurations have a direct impact on the security and the efficiency of the network. To highlight this situation, we introduce the following invariants:

Loop freedom: it means that there should not exist any packet injected into the network, that it would cause a forwarding loop. The loop invariant can be identified by checking the flow history to determine if the flow has passed through the current switch before. For example, according to our topology shown in Fig.1, $PreviousPaths = \{S2 \rightarrow S1 \rightarrow S3 \rightarrow S2\}$. Therefore, the faulty switch is S2.

Access violation : Openflow allows various Set-actions, which can rewrite the values of header fields in packets. This challenge can influence the path parsed by some packets. More, flow rules may overlap each other in the same switch or between switches which cause indirect network breaches. Depending on the complexity of an overlap found in violated space, we distinguish between two types of access violations:

- Entire Violation: if the fields domain of the decision path covers the whole space (denied or accepted) of the security policy.
- Partial Violation: if the fields domain of the decision path partially covers the space of the security policy.

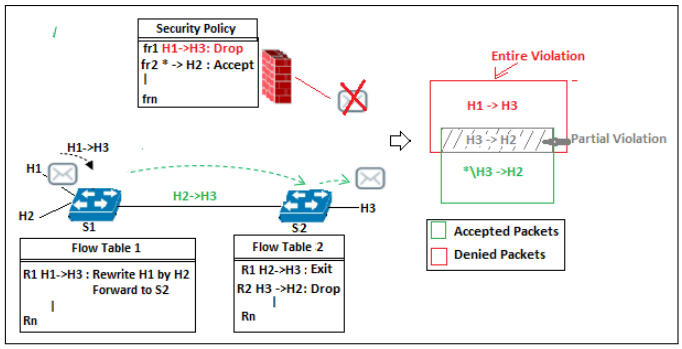

Fig. 2. Indirect Access Violation due to field modification.

For a concrete example, consider the toy network in Fig.2. In this scenario, the H1 wants to send a packet to the H3 machine. When the packet arrives at the switch entry, it matches its first rule that changes its source address to that of H2 and then it sends it to the S2 switch, which in turn allows this packet to exit the network. This contradicts the first firewall rule (fr1) that drops all traffic from H1 to H3 : it leads to an entire violation. In addition, the second firewall rule (fr2) allows all traffic sent to the H2 host. But the R2 rule of switch S2, drops the packet sent from H3 to the H2 host. This results in a partial violation.

Blackholes and Controller: in order to pinpoint the "Blackhole" and "SendTo controller" invariants, the SDN switch configuration considers the default-action (*Drop, Empty or SendTo controller*) as an action used for packets that don't match any existing flows. As depicted in Fig.1, the default action deployed in the two switch configurations S1 and S2 is "Drop" and in the S3 flow table is "Empty". Hence, S3 engenders a blackhole.

IV. Our Approach

A. Principle

The principle of our approach, depicted in Fig.3, is based on three main phases: ***Phase 1:*** Security Policy Space Analysis– given the security policy deployed in the firewall application as input, we extract accepted and denied packets which is shown in Fig.3 respectively as two spaces SP_{accept} and SP_{deny}.

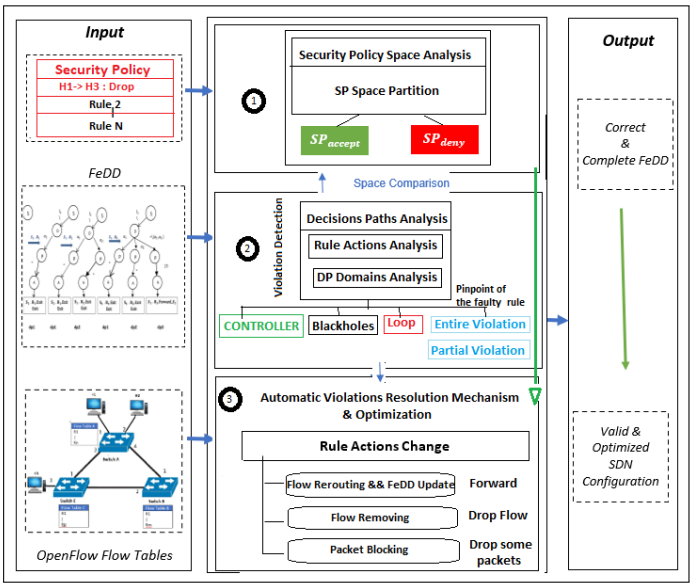

Fig. 3. Approach Architecture.

Phase 2: Security Analysis & Violation Detection– in this step, we analyze each decision path in our FeDD in order to detect possible defects in the configuration. We identify basic anomalies as well as network security policy access violations. This detection is based on the verification of a set of invariants according to the Security Policy (SP). The major advantage of this detection is to specify the rule that caused the error, unlike other related works which reject the flow or identify only the set of faulty switches. This step will help us directly and quickly correcting the wrong decision without analyzing all the paths in our FeDD.

Phase 3: Automatic Violations Resolution & Refinement– in this step, we define a set of resolution methods for each detected invariant with respect to the following technical requirements: accuracy, flexibility and scalability.

B. Security Analysis & Defects Detection

In this section, we introduce our algorithm, depicted in Fig.4, for discovering various invariants from our FeDD. To achieve our goal, we start with the following definition:

Definition 1. FeDD is called misconfiguration-free if and only if $\exists dp_i \in FeDD$ that verifies one of the following conditions:
- Loop (LP): a direct path $dp_i \in FeDD$ invokes forwarding of loops if the previousPaths stores twice the same switch traversed by this dp_i.
- Blackhole (BLK) : a direct path $dp_i \in FeDD$ depicted a blackhole if the packet matched the default action Empty as configured in the switch.
- Entire Violation (EnV) : a direct path $dp_i \in FeDD$ is **totally** violated if **all** the packets tracked by this path apply a different action as applied in the security policy SP. Hence *EnV* is identified by applying the algorithm shown in Fig.4 (L1-L3) . Formally: $(domain(dp_i) \subseteq SP_{!dp_i.r_{vi}.action})$
- Partial Violation (PaV) : a direct path $dp_i \in FeDD$ is **partially** violated if **some** packets tracked by this path apply a different action as applied in the security policy SP. Hence *PaV* is identified by applying the algorithm shown in Fig.4 (L4-L6). Formally: $(domain(dp_i) \not\subset SP_{dp_i.r_{vi}.action}) \wedge (domain(dp_i) \cap SP_{dp_i.r_{vi}.action} \neq \emptyset)$.

In fact, it compares the domain of the direct path with the set of packets of the security policy having two different actions, if it is totally included by it, then we have the entire Violation *EnV* and if it is partially included by it, then we have a partial Violation *PaV*.

Input : FeDD, a set of decisions paths dp

SP_a , a set of accepted packets from SP

SP_d , a set of denied packets from SP

Output : EnV, a set of dp detecting an entire violation

PaV, a set of dp detecting a partial violation

```
L1 foreach dp ∈ FeDD do
L2   if (r. actions ≠ fr. actions) ∧ (Dom(dp) ⊆ SP_r.actions)
L3     dp.append("EnV ") ;
L4   else
L5     if (Dom(dp) ⊈ SP_r.actions ) ∧ (Dom(dp) ∩ SP_r.actions ≠ ∅)
L6       dp.append("PaV ") ;
L7 FeDD ← FeDD\{dp};
L8 return EnV, PaV ;
```

Fig. 4. Algorithm: Discovering Access Violations from FeDD.

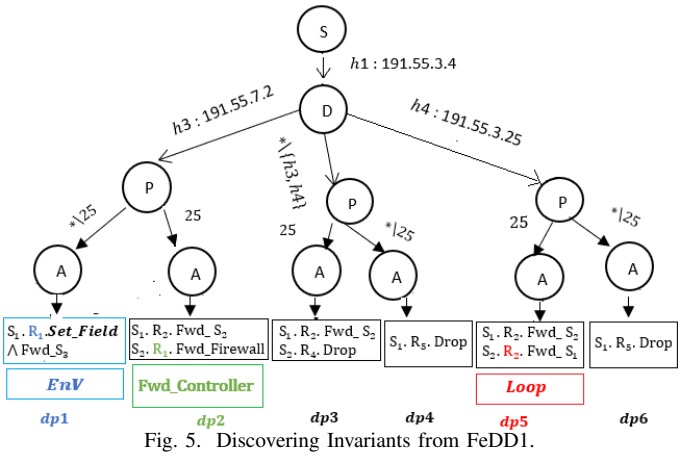

Fig. 5. Discovering Invariants from FeDD1.

For example, in our case study and according to Fig.1, we have three sets of possible input addresses (h1:191.55.3.4,

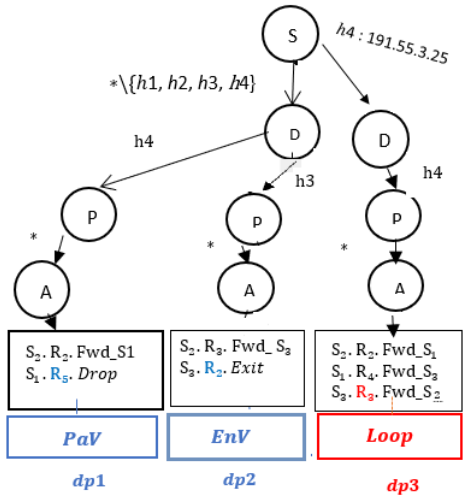

Fig. 6. Discovering Invariants from FeDD2.

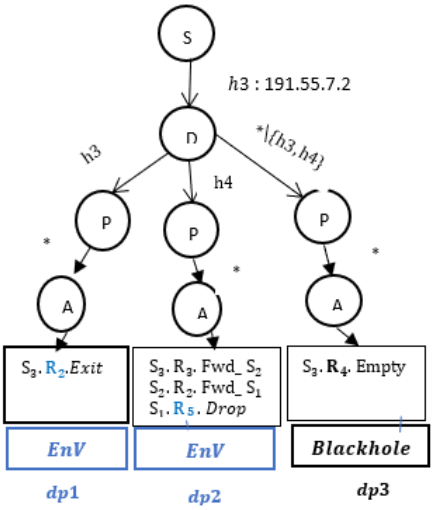

Fig. 7. Discovering Invariants from FeDD3.

h2:191.55.3.9 and h3:191.55.7.2), and by applying our algorithm, shown in Fig.4, we obtain all invariants discovered from FeDD depicted in Fig.5, Fig.6 and Fig.7. *FeDD2 of switch S2*: we identify a loop at dp3 caused by rule R3 of switch S3 because it forwards a packet to switch S2 which is already visited. We also have one entire violation EnV at dp2, according to the firewall rule fr4. We identify at dp1 one partial violation PaV because some packets of this dp1 will be accepted (191.55.3.0, 191.55.3.25, 25) according to the firewall rule fr3.

C. Invariants Correction & Refinement

Our objective is to determine which correction method should be used for each detected invariant.

1) Resolving loop forwarding: our approach to resolve loop forwarding from our FeDD is shwon in Fig.8. At first, we extract all possible paths from our topology to track a packet from source to target destinations. For example, $(source, target) = (191.55.3.4, 191.55.7.2) = (S1, S2, S3) \vee (S1, S3)$. Then, for each $dp_i \in$ LP, we compare the domain of this dp_i with allowed or denied spaces from the space SP set. Thus, we have two cases:

- Case 1: $domain(dp_i) \in SP_a$; in this case, we have two situations: (i) the destination of this dp_i is not linked to the switch caused a loop : in this situation, we forward the packet to the next switch. In Fig.9, we demonstrate our loop resolution process at dp5 of Fig.5. In fact, given a network topology, we identify the paths followed by the packet from the source address (dp.SourceIP) to the target destination (dp.destIP). Then, we retrieve the following switch identifiers from these paths. When, we reach the last switch (terminal), we assign the action "Exit" to the corresponding rule (L11-L12). (ii) Otherwise, we just modify the rule action to "Exit" as depicted in Fig.8 (L3-L5);

Input : LP : the set of decision paths dp where a loop is detected

Output : dp well corrected.

```
L1  Foreach  dp ∈ LP  do
L2    if  dom(dp) ∈ SP_a  then
        /* case1 : destination IP address of this dp is linked to the faulty switch.*/
L3      if dom(dp.DestIP).attached(dp.Sid) then
L4        r.setAction("Exit") ;
L5        dp.append("Exit") ;
L6      else
L7        nextPort = dom(dp).findDestPort(Sid) ;
L8        r.setAction("port_"+nextPort) ;
L9        flowAux.updateFlow(dom(dp)∩dom(r)) ;
L10       nextSw= dom(dp).extractSid(r.actions) ;
              /* check if nextSw is last switch*/
L11         if dom(dp).islastSwitch(dp.nextSw) then
L12           r.setAction("Exit") ;
L13         else
L14           exploreRecursively (flowAux, nextSw, previousPaths, dp ()) ;
L15   else
L16   r.setAction("Drop") ;
L17   dp.append("Drop") ;
```

Fig. 8. Algorithm for resolving loop defects.

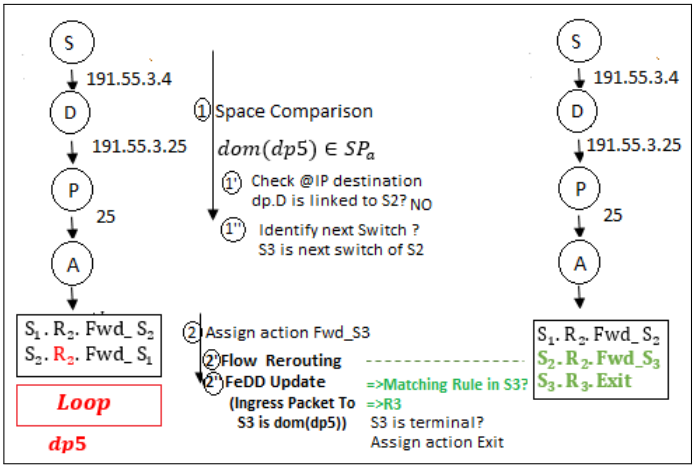

Fig. 9. Application of the loop resolution process.

- Case 2: $dom(dp_i) \in SP_d$, we just change the action of the rule r_i which caused a loop to "Drop" (L16-L17).

2) Access Violations Resolution: the inference system rules, shown in Fig.10, apply to triplet (EnV, PaV, FeDD). The

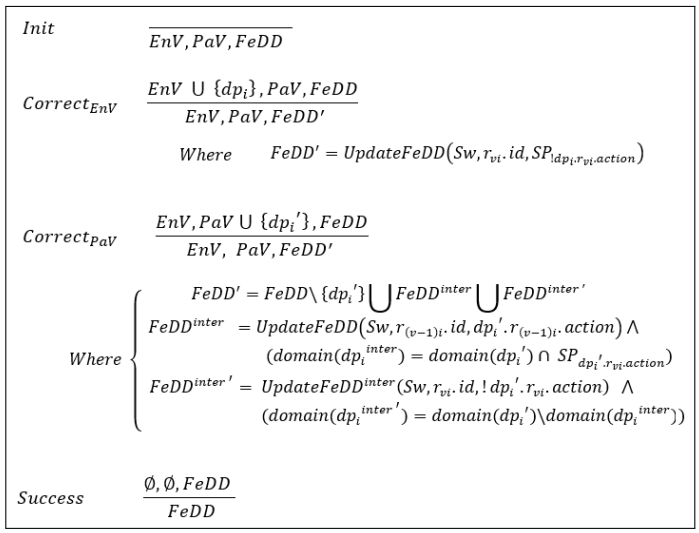

Fig. 10. Inference system for resolving entire and partial access violations.

inference rule $Correct_{EnV}$ is used to correct EnV. It deals with each dp_i from the set EnV and changes the action of the rule that caused this violation in dp_i. The inference rule $Correct_{PaV}$is used to divide $dp_i^{'}$, into two sets :

1) dp_i^{inter} is the set that has the correct action as defined in the security policy;
2) $dp_i^{inter^{'}}$ represents the subset of $dp_i^{'}$ that should be fixed. This inference rule is used to correct the action of this path dp_i.

The function *UpdateFeDD(Sw, r_{vi}.id, $dp_i.r_{vi}$.action)* allows to update the FeDD by replacing dp_i by the new direct path. As an example, we deal with the case of PaV discussed in Fig.6. We consider $dp1^{inter}$ = $dp1 \cap SP_a$ = the branch represented by these values: $[@src_ip, @dest\_ip, port\_dest]$=$[191.55.3.0, 191.55.3.25, 25]$. Therefore, $dp1$=$(dp1 \setminus dp1^{inter}) \cup (dp1 \cap dp1^{inter})$. Then, by using our inference rule $correct_{PaV}$ to divide this direct path into two sub paths where the first $dp1 \cap dp1^{inter}$ represent paths which are conform to SP and the second one (dp1 $\setminus dp1^{inter}$) is the totally violated path. In order to prove the correctness of our approach, we start with the following definition:

Definition 2. A direct path dp_i in FeDD is called well-configured with respect to SP if and only if for all rule r_{vi} in dp_i, $dom(dp_i) \subseteq SP_{action(r_{vi})}$.

Theorem 1. *(Correctness): if there exists a finite derivation: $(EnV, PaV, FeDD) \vdash^* FeDD^{'}$ then $FeDD^{'}$ is well configured with respect to the security policy SP.*

Proof. if $(EnV, PaV, FeDD) \vdash^* FeDD^{'}$ then we have $(FeDD1^{'}, EnV1, PaV1) \vdash (FeDD2^{'}, EnV2, PaV2) ... \vdash (FeDDn^{'}, EnVn, PaVn) \vdash Success$ where EnVn = $\emptyset$ and PaVn = $\emptyset$. For each step, we update FeDD ($FeDD^{'}$) by applying the function UpdateFeDD, we have two cases: (i) by applying the inference rule $Correct_{EnV}$. In this situation, we assign the same action deployed in SP to the rule r_{vi}. We can easily show by induction on i that for all $1 \leq i \leq n$, if dp_i is in *EnV* then r_{vi} is removed from dp_i. Therefore, $action(dp_i) = action(r_{(v-1)i}) \neq action(r_{vi})$ which is conform to the action applied by SP on the packets matched by dp_i i.e $dom(dp_i) \subseteq SP_{action(r_{vi})}$; and (ii) by applying the inference rule $Correct_{PaV}$. In this case, we divide $dp_i^{'}$ into two sets: (1) the set dp_i^{inter} that has the correct action as defined in SP; and (2) the subset of $dp_i^{'}(dp_i^{inter^{'}})$ that has been fixed by assigning the same action deployed in SP to the rule r_{vi}. Thus, for all $1 \leq i \leq n$, if $dp_i^{inter^{'}}$ is in *PaV* then r_{vi} is removed from $dp_i^{'}$. Hence, $action(dp_i^{inter^{'}}) \neq action(r_{vi})$ which complies with the SP requirement on the packets matched by $dp_i^{inter^{'}}$ (i.e $dom(dp_i^{'}) \subseteq SP_{action(r_{vi})}$). Therefore, our reasoning is correct. □

Theorem 2. *(Completness): for **all** FeDD there exists a finite derivation:$(FeDD, EnV, PaV) \vdash^* FeDD^{'}$ such that $FeDD^{'}$ is well configured with respect to the security policy SP.*

Proof. We have two cases: (i) case 1: FeDD is well configured. Therefore, PaV=EnV=$\emptyset$. In this case, the *Success* rule will be applied; (ii) case 2: we assume that FeDD is not well configured. Then, there exists dp_i in FeDD belonging to one of two sets (1) $dp_i \in EnV$ which means $EnV \neq \emptyset$. But, the rule $Correct_{EnV}$ removes this dp_i from EnV and assigns the action that conforms to SP ; (2) $dp_i \in PaV$ which means $PaV \neq \emptyset$. In this case, the rule $Correct_{PaV}$ fixes the subset of $dp_i^{'}$ by removing the rule r_{vi} from PaV. It follows that in finite steps, we will obtain $FeDD^{'}$ as well configured with respect to SP. □

It is easy to show that our inference system is terminating:

Theorem 3. *(Termination): The inference system, shown in Fig. 10, is terminating.*

Proof. The inference system is terminating since the cardinality of EnV + cardinal of PaV decreases at each step of the application of the inference system, in a finite number of steps: cardinal of EnV + cardinal of PaV = 0, then EnV =$\emptyset$ and PaV = $\emptyset$ and therefore the *Success* rule will be applied. □

3) Blackholes and Controller Resolution: the main idea to resolve blackhole defects is to compare the domain of each $dp_i \in BLK$ with the space of set packets in SP. Hence, we have two cases :

- $domain(dp_i) \subseteq SP_d$, we change the action of matched rule in this dp_i to **"Drop"**.
- $domain(dp_i) \subseteq SP_a$ we replace the *Empty* action by **Exit**.

For example, we resolve the blackhole identified at dp3, highlighted in Fig.6, by assigning the action "Drop" to rule

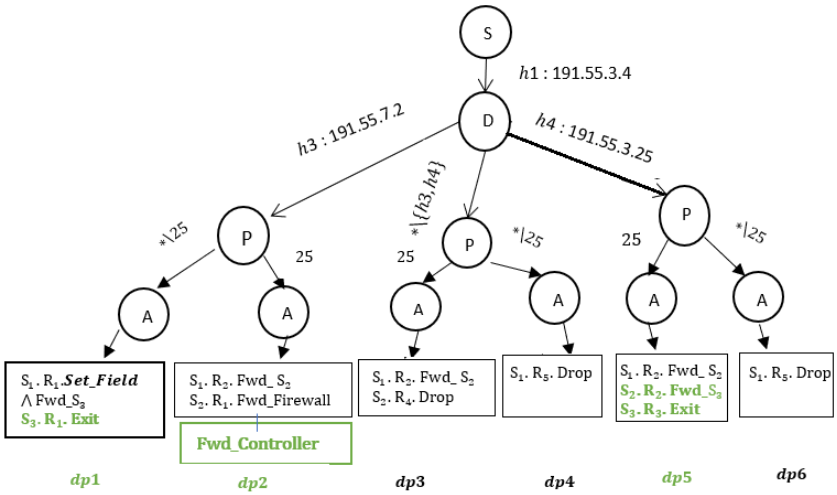

Fig. 11. Modified FeDD of Fig.5

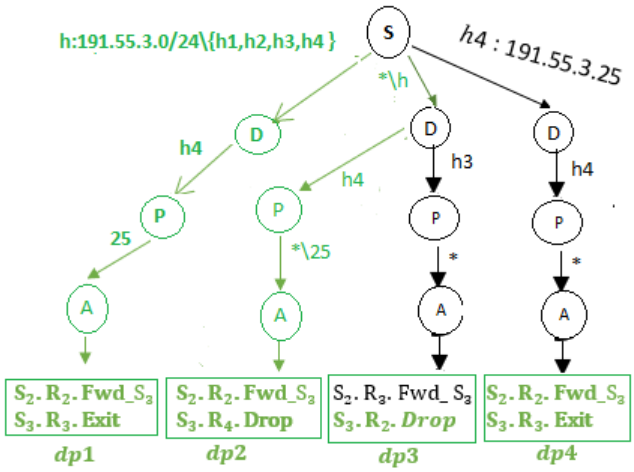

Fig. 12. Modified FeDD of Fig.6

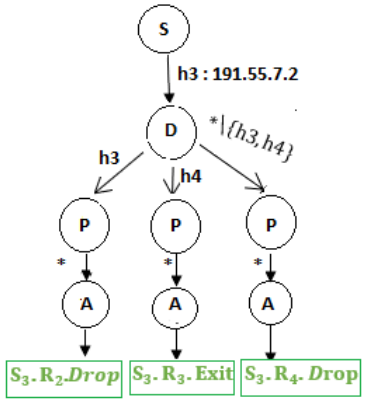

Fig. 13. Modified FeDD of Fig.7

R4 of switch S3 because $domain(dp3) \subseteq SP_d$ (see fr4 depicted in Fig.1). An invariant "Controller" is detected when a switch is enable to decide about a packet, so it forwards it to the controller. Therefore, the SDN Controller will filter a packet according to the security policy. As an example, SMTP traffic will be accepted from source 191.55.3.4 to destination 191.55.7.2 according to second rule deployed in the firewall configuration (see fr2 in Fig.1). After applying our correction methods shown in Fig.8 and Fig.10, we obtain the new updated FeDD shown in Fig.11, Fig.12 and Fig.13.

D. SDN Switch Configuration Validation

We observe that after resolving an invariant "loop" (identified at dp5 of Fig.5 and dp3 of Fig.6) by changing the two rules actions of S2 and S3 (S2.R2.Forward_ S3 and S3.R3.Exit), a partial violation PaV (identified at dp1 of Fig.6) and an entire violation EnV (identified at dp2 of Fig.7) are also well corrected as shown in Fig.11, Fig.12 and Fig.13. As results, we obtain the new switches configurations validated and well consistent after applying our anomalies analysis and resolution techniques as depicted in TABLE I and TABLE II.

TABLE I
NEW SWITCH S2 CONFIGURATION

Rule	Source	Destination	Port	action
1	191.55.3.4	191.55.7.2	*	Forward_Firewall
2	*	191.55.3.25	*	Forward_S3
3	*	191.55.7.2	*	Forward_S3
4	*	*	*	Drop

TABLE II
NEW SWITCH S3 CONFIGURATION

Rule	Source	Destination	Port	action
1	191.55.3.9	*	*	Exit
2	*	191.55.7.2	*	Drop
3	*	191.55.3.25	*	Exit
4	*	*	*	Drop

V. Experimental results and Evaluation

A. Complexity Study

Given a Switch of n rules, the maximum number of paths in the FeDD updated using our solution is $(2n-1)^f$, where f is the number of the fields in each rule. Hence, the complexity of the inference system for modifying FeDD is $O(n^f)$. The inference system, shown in Fig.10, allows to modify FeDD by inserting some direct paths and by modifying the field dp_i.r. Therefore, the complexity of this inference system is equal to the complexity of a tree-set insertion operation (which is equal to $O(log(n))$) plus $O(n^f)$. Thus, the complexity of this inference system is equal to $O(n^f)$. Given that f is typically small (generally, we have 3-5 fields) our algorithms have a reasonable response time in practice.

B. Implementation and Experimentals Results

The experiments were run on desktop with an Intel Core i7 CPU 3.6GHz and 32GB Memory. Then, to implement our methods, we use Java JDK 1.8 with Eclipse. In order to easily integrate our solution, we used all-in-one pre-built virtual machine by SDN Hub [20] within Ubuntu-14.04.4. We emulate networks with Mininet and use the controller OpendayLight with support of Openflow v1.3. It is supposed that we have IP v4 addresses with netmasks and port numbers of 16 bits unsigned integer with range support. For example, IP prefix 191.55.0.0/24 can be converted to the interval from 191.55.0.0 to 191.55.255.255, where an IP address can be regarded as a 32-bits integer. To evaluate the practical value of our methods, we have implemented them based on the FeDD data structure using the rules set provided by two topologies:

1) Simple network topology shown in Fig.1. We used, at first, the following command to build the configuration of our topology from a file "MyTopo":
   `ubuntu@sdnhubvm :/mininet/configuration$ sudo mn --custom MyTopo.py --topo MyTopo`.
   Then, We used the ping tool in order to populate the switches configurations with shortest-path forwarding rules. As result of our application, Fig.14 demonstrates a resolution of loop identified at path3 (case: @IP 191.55.3.25 is not linked to the faulty switch S2). Then, it modifies the action of the second rule of S2 to "Forward_S3". This leads to fix the third rule of S3 (the new action is "Exit"). The green rectangle surrounds the validated rule R2 of S2 in the tabular list.
2) The Internet2 topology [25] which consists of 9 Juniper routers. For Internet2, the configuration files are translated to correspondent OpenFlow rules, and installed at Open vSwitches. Then, we extracted data from switches using the command line tool : `rodvand@atpg$sudo ovs-ofctl dump`

We have also conducted a set of experiments to measure the performance of our algorithms. Hence, we consider time treatment factor that we review by varying the number of rules for each dataset. The maximum number of rules, deployed in a single switch, is 3000. In overall terms, we consider the

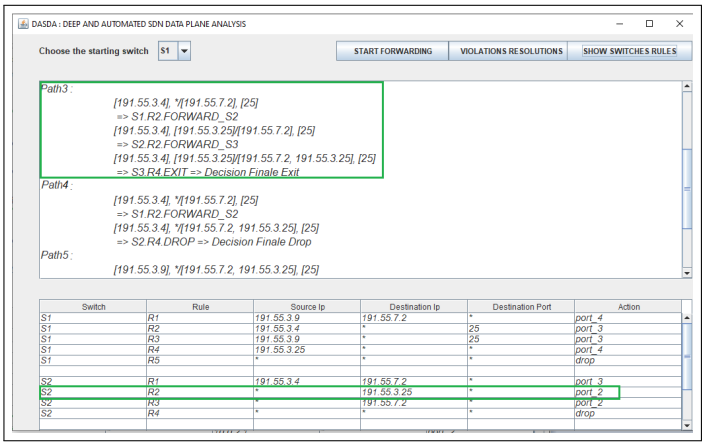

Fig. 14. DASDA violation resolution interface.

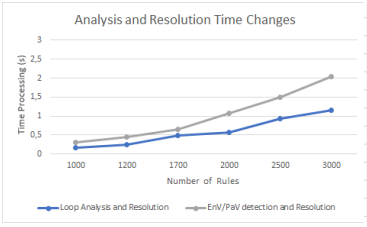

Fig. 15. Resolution Time Changes.

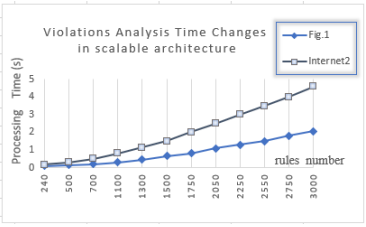

Fig. 16. Scalability Analysis.

average processing time, in seconds, of the main procedures of misconfigurations detection and correction. The violation detection and resolution overhead were increased linearly as the switches size increases as shown in Fig.15. The experimental results, depicted in the Fig.16, show a polynomial increase with the growth of flow rules in scalable architecture such as Internet2 topology. The traffic latency in Internet2 is due to rules complicated dependencies. Therefore, obtained processing time shows that our tool performed efficiently within the case studies.

VI. Conclusion and Future Work

We presented in this paper, techniques for fine-grained management of openflow SDN switch misconfigurations. More precisely, our proposal is intended for a comprehensive discovering and fixing of data plane security invariants based on formal techniques and by using FeDD as data structure. The main advantages of our proposal are the following: First, unlike other works, our approach ensures continuous SDN data plane compliance with the security policy without causing further errors as a result of our accurate and optimal resolution mechanism. This is justified by our study of the implemented methods complexity. Second, we formally proved the correctness and completeness of our formal reasoning for validating SDN data-plane configurations. Third, our experimentations, that have been conducted on different case studies, highlighted promising results. As a future work, we plan to consider techniques for verifying SDN security policies and resolving violations in a real time context.

REFERENCES

[1] N. Yoshiura, K. Sugiyama : Packet Reachability Verification in OpenFlow Networks. In: 9th International Conference on Software and Computer Applications, ICSCA 2020, pp. 227–231. ACM, Langkawi, Malaysia,2020. URL https://doi.org/10.1145/3384544.3384573

[2] Y. Zhang, J. Li, S. Kimura, W. Zhao, S. K.Das : Atomic Predicates Based Data Plane Properties Verification in Software Defined Networking Using Spark. IEEE Journal on Selected Areas in Communications, 2020.

[3] A. Shaghaghi, M. A. Kaafar, R. Buyya, S. Jha:Software-Defined Network (SDN) Data Plane Security: Issues, Solutions, and Future Directions. In: Handbook of Computer Networks and Cyber Security, pp. 341-387. Springer, Cham, 2020.

[4] B. Celesova, J. Val'ko, R. Grezo, P. Helebrandt : Enhancing security of SDN focusing on control plane and data plane. In : the 7th International Symposium on Digital Forensics and Security (ISDFS), pp. 1-6. IEEE, 2019.

[5] W. Saied, N. B. Y. B. Souayeh, A.Saadaoui and A. Bouhoula : Deep and Automated SDN Data Plane Analysis. In SoftCOM, 2019.

[6] A. Banerjee, D. A. Hussain : Maintaining Consistent Firewalls and Flows (CFF) in Software-Defined Networks. In : Smart Network Inspired Paradigm and Approaches in IoT Applications, pp. 15-24. Springer, Singapore, 2019.

[7] Hu.Hongxin, Han.Wonkyu, K.Sukwha, W.Juan, A.Gail, Z.Ziming, Li.Hongda : Towards a reliable firewall for software-defined networks. In Computers & Security, vol. 87, 101597. Elsevier, 2019. https://doi.org/10.1016/j.cose.2019.101597

[8] Q. Li, Y. Chen, P. P. Lee, M. Xu, K. Ren : Security policy violations in SDN data plane. IEEE/ACM Transactions on Networking, 26(4), 1715-1727, 2018.

[9] B. Yamansavascilar, A. C. Baktir : Flowtable pipeline misconfigurations in software defined networks. In : IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 247-252). IEEE , 2017.

[10] W.Han, H.Hu, Z.Zhao, A.Doupé, G.-J.Ahn, K.-C.Wang, J.Deng: State-aware network access management for software-defined networks. In ACM, 2016.

[11] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. FLOWGUARD: building robust firewalls for software-defined networks. In HotSDN, 2014.

[12] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In USENIX, 2013.

[13] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. In USENIX, 2013.

[14] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In SIGCOMM, 2011.

[15] G. Pickett. Staying persistent in software defined networks. In Black Hat Briefings, 2015.

[16] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In ICNP, 2013.

[17] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In CoNEXT, 2012.

[18] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang. Enabling layer 2 pathlet tracing through context encoding in software-defined networking. In HotSDN, 2014.

[19] All-in-one sdn app development starter vm. http://sdnhub.org/tutorials/sdn-tutorial-vm, 2019

[20] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In SafeConfig, 2010.

[21] Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, Yuemei Zhang: Mind the Gap: Monitoring the Control-Data Plane Consistency in Software Defined Networks. In CoNEXT, 2016.

[22] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In PLDI, 2014.

[23] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test Openflow applications. In USENIX, 2012.

[24] A.Wundsam, D.Levin, S.Seetharaman, A.feldmann, et al. OFRewind: Enabling record and replay troubleshooting for networks. In USENIX, 2011.

[25] The Internet2 Observatory Data Collections. http://www.internet2.edu/observatory /archive/data-collections.html, 2019.