# A Formal Approach for Automatic Detection and Correction of SDN Switch Misconfigurations

Wejdene Saied
*Digital Security Research Lab, (Sup'Com)*
*University of Carthage, Tunisia*
wejdene.saied@supcom.tn

Adel Bouhoula
*College of Graduate Studies*
*Arabian Gulf University, P.O. Box 26671, Kingdom of Bahrain*
a.bouhoula@agu.edu.bh

*Abstract*—**Software-defined networking (SDN) is a network architecture that enables the network to be centrally controlled using software. The network administrators can reprogram the network using SDN without changing hardware devices to provide new solutions for controlling network traffic. However, SDN has its drawbacks in security, scalability, and elasticity. The security validation of SDN configurations is an important issue that should be addressed. Therefore, there is a need for automated methods to analyze, investigate and fix switch configurations faults. The objective of our work is to propose: (1) a new formal approach to discover security challenges using Flow entries Decision Diagram (FeDD) analysis, to identify loop freedom, access violation, black-holes, and controller misconfiguration; (2) an optimal and fine-grained resolution mechanisms to correct these misconfigurations in different topologies: (3) a tool that implements the proposed techniques and effectively helps administrators in detecting and resolving switch misconfigurations.**

*Index Terms*—**Software Defined Networking, Security Policy, Invariants Detection, Flow entries Decision Diagram, Formal Method.**

## I. Introduction

Traditional networks follow an "inside the box" paradigm. This increases the complexity and cost of network configuration, management and troubleshooting. Thus, proposals for a new networking paradigm, namely Software Defined Networking (SDN), have emerged. The principal endeavors of SDN are to decouple control plane from the data plane and to centralize network's intelligence. However, this new architecture is prone to many data plane faults, mainly caused by inconsistent rules in the switch configuration. The recent violation resolution proposals help administrators manually resolve each reachability issue. These mechanisms can cause possible defects in the switch configuration and can subsequently damage network security [3]. To deal with switch rules analysis problem, many solutions have been proposed [6], [14], [15]. For example, the 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. The authors, in [2] adopt the concept of atomic predicates and the parallel process computational framework Spark to verify data plane properties. However, they do not propose any resolution mechanism for these defects. ATPG [17] automatically generates test packets by injecting network probes. However, it cannot localize the faulty rule. Moreover, this tool is limited to detect only liveness properties. [21] uses a Bloom-filter tagging method in order to detect the flow table inconsistencies. However, they do not incorporate rewrites into the current framework, in order to support actions that need to modify packet headers. In addition, they do not provide an automatic resolution for repairing the faulty switch configurations. These approaches can detect only some basic invariants. Some other approaches try to verify the compliance of switch configuration with respect to the security policy [12], [16], [18]. For example, [7] addresses the challenges created by the interaction between flow path and firewall authorization space by proposing the FlowMon tool. However, their solution cannot detect indirect violations caused by rule dependencies. FlowChecker [20] only identifies intra-switch misconfigurations within a single flow table. 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 [13] for defending covert channel attacks [8]. However, the header change rules still cannot take action in the forwarding graphs for verifying the reachability. Authors in [11] propose the FlowGuard tool for building SDN firewalls. They propose resolutions strategies designed for diverse network update situations but, they cannot inspect dynamic packet modifications.

Related works [4], [10], [22]–[24] do not address automated resolutions, they only raise alarms, indicate violations and ignore rule dependencies and some invariants within security constraints. In this work, we propose a new approach to detect and fix misconfigurations in a SDN switch.

This paper is organized as follows: Section 2 overviews the formal representation of OpenFlow switch flow table and security policies and details FeDD structure. Section 3 details our approach. Finally, we present our conclusions and discuss our plans for future work.

978-3-903176-31-7 © 2020 IFIP

## II. Preliminaries

### 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. Openflow Switch Flow Table [19]

Flow table entries consist of a set of instructions which are applied to all matching packets. Every packet has a set of actions associated with it. These are policies determining what should happen to the packet: it decides how packets are processed after they have matched against an entry found in the flow table. Examples of actions are forward (output) the packet to port X, drop the packet, modify the packet header and send to controller. The action set is empty by default.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 $action_i = \{Forward, Drop, Empty, Set\_Field \wedge Forward, Controller\}$ where action "Controller" sends packets to the controller which will filter according to the security policy.

### C. Flow Entries Decision Diagrams Representation

In this paper, we used 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 root node. A directed path from the root to a terminal node is called a decision path $dp_i$. We discussed in [5], the algorithm used to construct a FeDD. 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:

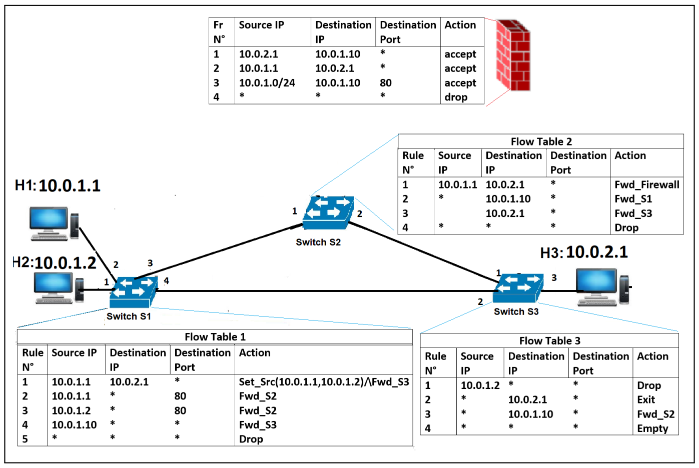
- $dp_i$.S, $dp_i$.D, $dp_i$.P are the domain of 3-tuple fields (Source address IP, Destination address IP and Port destination) matched by the direct path $dp_i$.
- $dp_i$.Sid.r is the identifier of the current switch that owns the rule matching the domain of packets in the $dp_i$.
- $dp_i$.r.id identifies the rule overlapped with the packets domains represented by this $dp_i$.
- $dp_i$.r.actions 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 the flow entries decision diagram. Therefore, FeDD of our network is : $FeDD = \cup_i FeDD_i = \cup_{ik:1..n} dp_k$

## III. Approach Overview

### A. Case Study

To make our study concrete, we use an example of network topology shown in Fig.1. It consists of three switches S1, S2 and S3 , three hosts where H1 and H2 are linked to the switch S1 and one host H3 is attached to S2. We also consider one firewall application as the set of requirements to be respected and used to identify the switches misconfigurations. We assume that the firewall rules are consistent. For a concrete example,



| Fr N° | Source IP | Destination IP | Destination Port | Action |
|---|---|---|---|---|
| 1 | 10.0.2.1 | 10.0.1.10 | * | accept |
| 2 | 10.0.1.1 | 10.0.2.1 | * | accept |
| 3 | 10.0.1.0/24 | 10.0.1.10 | 80 | accept |
| 4 | * | * | * | drop |

Flow Table 2

| Rule N° | Source IP | Destination IP | Destination Port | Action |
|---|---|---|---|---|
| 1 | 10.0.1.1 | 10.0.2.1 | * | Fwd_Firewall |
| 2 | * | 10.0.1.10 | * | Fwd_S1 |
| 3 |  | 10.0.2.1 | * | Fwd_S3 |
| 4 | * | * | * | Drop |

Flow Table 1

| Rule N° | Source IP | Destination IP | Destination Port | Action |
|---|---|---|---|---|
| 1 | 10.0.1.1 | 10.0.2.1 | * | Set_Src(10.0.1.1,10.0.1.2)/\Fwd_S3 |
| 2 | 10.0.1.1 | * | 80 | Fwd_S2 |
| 3 | 10.0.1.2 | * | 80 | Fwd_S2 |
| 4 | 10.0.1.10 | * | * | Fwd_S3 |
| 5 | * | * | * | Drop |

Flow Table 3

| Rule N° | Source IP | Destination IP | Destination Port | Action |
|---|---|---|---|---|
| 1 | 10.0.1.2 | * | * | Drop |
| 2 | * | 10.0.2.1 | * | Exit |
| 3 | * | 10.0.1.10 | * | Fwd_S2 |
| 4 | * | * | * | Empty |

Fig. 1. Network Topology Study.

we illustrate the rules in the three flow tables respectively to S1, S2 and S3:

- In the flow table 1, rule 1 alters the source address H1 of the ingress packet, replaces it by new one 10.0.1.2 and sends this flow to S3. Rule 2 and rule 3 redirect the HTTP traffic from the source address 10.0.1.1. and 10.0.1.2 to switch S2. Rule 4 forwards all other packets towards 10.0.1.10 to S3.
- In the flow table 2, rule 1 directs the source address 10.0.1.1 to the firewall. Rule 2 forwards other packets with destination address 10.0.1.10 to switch S1 and H3 to S3.
- In the flow table 3, rule 1 at switch S3 drops all traffic from H2. Other rules are plain forwarding rules ensuring connectivity. The default action is Empty.

We notice that when the ingress packet matches rule 1 in the flow table 1, its source address IP must be modified to new one 10.0.1.2 and forwarded to switch S3 which in turn drop this packet. However, according to the second rule (fr2) in the firewall configuration, the flow must be forwarded not rejected: this scenario demonstrates the security policy violations and therefore, the switch misconfiguration must be fixed.

### B. Switch Misconfigurations Detection

*1) Security Policy Space Partition:* In Fig.2, we represent our inference system to convert a list of firewall rules into two disjoint security policy subspaces denied $SP_d$ and allowed space $SP_a$ as defined in section 2. In fact, for each *fr* in *SP*, if this rule is an "accept" rule, its domain is compared with existing domains in the denied space $SP_d$. If the dom(fr) is covered by any existing domain in $SP_d$, the covered spaces(s) is removed from dom(fr) and the modified dom(fr) is added into $SP_a$. Hence, $SP_a$ is updated by applying the inference rule *Append*_$SP_a$. The similar process is applied to a "deny" rule and by applying the inference rule *Append*_$SP_d$.

$$
\begin{array}{ll}
Init & \dfrac{}{(SP,\emptyset,\emptyset)} \\[2ex]
Append\_SP_a & \dfrac{((fr \Rightarrow accept) \cup SP), SP_a, SP_d)}{(SP, SP_a \cup (dom(fr) \setminus SP_d), SP_d)} \\[2ex]
Append\_SP_d & \dfrac{((fr \Rightarrow deny) \cup SP), SP_a, SP_d)}{(SP, SP_a, SP_d \cup (dom(fr) \setminus SP_a))} \\[2ex]
Stop & \dfrac{(SP, SP_a, SP_d)}{Stop}
\end{array}
$$

Fig. 2. Inference System for Partitioning Security Policy space.

2) *Misconfigurations classification:* In our work, at first, our goal is to propose an automatic method that parse network properties with respect to the security policy from our data structure FeDD already constructed. To achieve our goal, we propose a solution based on inference systems as depicted in Fig.3. In order to formally specify the analysis and detection process and prove the correctness of our approach, we first present the following definition:

$$
\begin{array}{lll}
Init & \dfrac{}{FeDD,\emptyset,\emptyset,\emptyset,\emptyset} & \\[2ex]
Loop & \dfrac{FeDD \cup \{dp_i\}, LP, BLK, EnV, PaV}{FeDD, LP \cup \{dp_i\}, BLK, EnV, PaV} & if\ \exists(Sw, Sw') \in\ previousPaths\ \ where\ \ Swid = Sw'id \\[2ex]
BLackH & \dfrac{FeDD \cup \{dp_i\}, LP, BLK, EnV, PaV}{FeDD, LP, BLK \cup \{dp_i\}, EnV, PaV} & if\ \exists r_n \in\ dp_i\ \ where\ \ r_n.dp_i.actions = Empty \\[2ex]
AccessViolation & \dfrac{FeDD \cup \{dp_i\}, LP, BLK, EnV, PaV}{FeDD, LP, BLK, EnV', PaV'} & \\[2ex]
 & where \begin{cases} EnV' = EnV \cup \{dp_i\} \quad ;PaV' = PaV & if\ \ dom(dp_i) \subseteq SP_{!dp_i.r_vi.actions} \\ PaV' = PaV \cup \{dp_i\} \quad ;EnV' = EnV & if\ \ dom(dp_i) \nsubseteq SP_{dp_i.r_vi.actions} \ \wedge \\ & \quad dom(dp_i) \cap SP_{dp_i.r_vi.actions} \neq \emptyset \end{cases} & \\[2ex]
Success & \dfrac{\emptyset,\emptyset,\emptyset,\emptyset,\emptyset}{Success} & \\[2ex]
Failure & \dfrac{\emptyset, LP, BLK, EnV, PaV}{Failure} & if\ \ (LP \neq \emptyset\ \vee\ BLK \neq \emptyset\ \vee\ EnV \neq \emptyset\ \vee\ PaV \neq \emptyset)
\end{array}
$$

Fig. 3. Inference System for discovering various invariants from FeDD.

**Definition 1.** FeDD is called invariants-free if and only if $\nexists dp_i \in FeDD$ that verifies one of these conditions:

- Loop (LP): a direct path $dp_i \in FeDD$ invokes forwarding of loop iff the previous Paths stores twice the same switch traversed by this $dp_i$. Hence ***LP*** is the set containing all $dp_i$ causing loop defects and identified according to the rule *Loop* in the inference system defined in Fig.3.
- Blackhole (BLK): a direct path $dp_i \in FeDD$ depicted a blackhole iff the packet matched the default action **Empty** as configured in the switch. Hence ***BLK*** is the set containing all $dp_i$ causing blackhole defects and hence *BLK* is identified by applying the rule *BlackH* .
- Entire Violation (EnV): A direct path $dp_i \in FeDD$ is **totally** violated iff **all** the packets tracked by this path apply a different action as applied in the security policy SP. Formally: $(dom(dp_i) \subseteq SP_{!dp_i.r_{vi}.action}) \wedge (dom(dp_i) \cap SP_{dp_i.r_{vi}.action} = \emptyset)$
- Partial Violation (PaV): A direct path $dp_i \in FeDD$ is **partially** violated iff **some** packets tracked by this path apply a different action as applied in the security policy SP. Formally:$dom(dp_i) \cap SP_{dp_i.r_{vi}.action} \neq \emptyset$. In Fact, ***EnV*** and ***PaV*** are identified according to the rule *AccessViolation* defined in Fig.3.

**Theorem 1.** *(Soundness of Success) if* $(FeDD, \emptyset, \emptyset, \emptyset, \emptyset) \vdash^* Success$ *then FeDD is invariants-free.*

*Proof.* FeDD is *loop-free*, then $\forall dp_i \in FeDD$, we apply the inference rule *Loop* to define the set of *LP* where for this direct path $dp_i$, we verify the precondition of *Loop*, it means that $\exists Swid$ in the previousPaths was visited twice. Also, FeDD is *blackhole-free* then $dp_i$ applies the default action **Empty** in the flow table Ft. Thus, for this direct path $dp_i$ we apply the inference rule *BlackH* to define the set of *BLK* which contains all Blackholes paths. Otherwise, FeDD is *access violation-free*. Thus, we apply the inference rule *AccessViolation* to distinguish two sets of violations EnV and PaV. In fact, it compares the domain of the direct path with the set of packets of the security policy having two different action : if it is totally included by it then we have entire Violation *EnV* and if it is partially included by it then we have a partial Violation *PaV*. It follows that in all steps *Pass* inference rule is applied, ie. $LP = \emptyset$ and $BLK = \emptyset$ and $EnV = \emptyset$ and $PaV = \emptyset$, therefore $(FeDD, \emptyset, \emptyset, \emptyset, \emptyset) \vdash^* Success$. □

**Theorem 2.** *(Soundness of Failure)* $if\ \ (FeDD, \emptyset, \emptyset, \emptyset, \emptyset) \vdash^* Failure$ *then FeDD is not invariants-free.*

*Proof.* $if\ \ (FeDD, \emptyset, \emptyset, \emptyset, \emptyset) \vdash^* Failure$ then we have : (1) $LP \neq \emptyset$; or (2) $BLK \neq \emptyset$; or (3) $EnV \neq \emptyset$ or (4) $PaV \neq \emptyset$ then we conclude that FeDD is not invariants-free. □

For example, if we consider the network topology shown in Fig.1, we have three sets of possible input addresses (h1:10.0.1.1, h2:10.0.1.2 and h3:10.0.2.1) and by applying inference system, shown in Fig.3, we obtain, as illustrated by Fig.4 and Fig.5, the network invariants discovered from our FeDD.

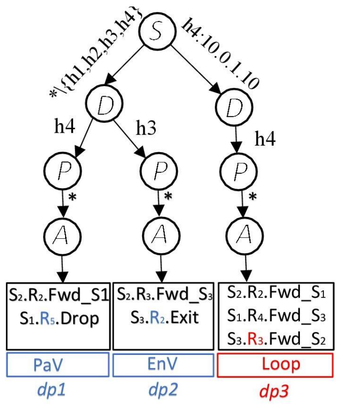


Fig. 4. Discovering invariants from FeDD2.

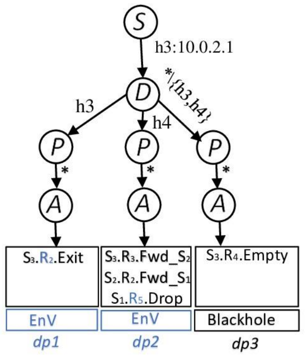


Fig. 5. Discovering invariants from FeDD3.

## C. Misconfigurations resolution techniques

1) ***Inference System for Resolving Loop Forwarding:*** The main idea is to compare the domain of each $dp_i \in LP$ with

existing header spaces in the space *SP* set. Thus, we have two cases: Case 1: $dom(dp_i) \in SP_d$, we just change the action of the rule $r_{li}$ which caused a loop to **Drop** by applying the inference rule $Correct_{Loop1}$ depicted in Fig.6; and Case 2: $dom(dp_i) \in SP_a$, we have two situations: (1) the destination of this $dp_i$ is linked to the switch that caused a loop, therefore we just modify the rule action to **Exit**; (2) otherwise, we forward a packet to next switch according to variable *Paths* ( the set of possible paths to send a packet from source to destination) until we attain the last switch (terminal), and change the rule action of the terminal switch to Exit. Thus, to achieve our goal, we apply the inference rule $Correct_{Loop2}$. In order to prove

$$Init \quad \frac{}{FeDD, LP, Paths}$$

$$Correct_{Loop1} \quad \frac{FeDD, LP \cup \{dp_i\}, Paths}{FeDD', LP, Paths} \; if \;\; dom(dp_i) \subseteq SP_d \;\; where \;\; FeDD' = UpdateFeDD(Swid_l, r_{li}.id, Drop)$$

$$Correct_{Loop2} \quad \frac{FeDD, LP \cup \{dp_i\}, Paths}{FeDD', LP, Paths} \; if \;\; dom(dp_i) \subseteq SP_a \;\; where \;\; FeDD' = UpdateFeDD(Swid_l, r_{li}.id, r_{li}.actions)$$

$$r_{li}.actions = \begin{cases} Exit & if \;\; isLastSw(dp_i.Swid_l) \\ Forwad\_NextSw & Otherwise \;\; where \;\; NextSw = Next(Swid_l) \in Paths \end{cases}$$

$$Success \frac{FeDD, \emptyset, Paths}{Success}$$

Fig. 6. Inference System for resolving loop defects.

the correctness of our approach, we start with the following definition:

**Definition 2.** A direct path $dp_i$ in FeDD is called totally well-configured iff for all rule $r_{vi}$ in $dp_i$, $dom(dp_i) \subseteq SP_{action(r_{vi})}$.

**Theorem 3.** *(Correctness): if* $(FeDD, LP, Paths) \vdash^* Success$ *then for all* $dp_i$ *in FeDD,* $dp_i$ *is corrected according to the security policy.*

*Proof.* if $(FeDD, LP, Paths) \vdash^* Success$ then we have $(FeDD1', LP1, Paths) \vdash (FeDD2', LP2, Paths) \ldots (FeDDn', LPn, Paths) \vdash^* Success \;\; where \;\; LPn = \emptyset$. For each step, We use the UpdateFeDD function to update our FeDD graph. The result is its derivative, called $FeDD'$, and provided according to one of two situations: (i) we assign the action "Drop" to the rule $r_{vi}$ by applying the inference rule $Correct_{Loop1}$; and (ii) we assign the action "$Forward_Nextsw$" to the rule $r_{vi}$ until to reach the terminal switch by applying the inference rule $Correct_{Loop2}$. In this situation, we assign the same action deployed in SP to the rule $r_{vi}$ (the action "Exit" i.e it agrees to forward traffic).
We can induce on i that for all $1 \leq i \leq n$, if $dp_i$ is in LP then $r_{vi}$ is removed from $dp_i$. Therefore, $action(dp_i) \neq action(r_{vi})$ which is conform to the action applied by SP on the packets matched by $dp_i$ i.e $dom(dp_i) \subseteq SP_{action(r_{vi})}$. Hence, our reasoning is correct. □

In our illustrative example, dp3 of Fig.4 depicted that the rule R2 of switch S2 caused a loop. According to the rule fr3 of the firewall, $dom(dp3) \in SP_a$, and $dp3.D = \{10.0.1.10\}$ is not linked to S2. Therefore, we apply the action Forward_S3 where S3 is the next switch of S2, then S3 is a last switch, so, we assign the action *Exit* to the rule R3 of S3 (the matched rule with dom(dp3)). Thus, to correct this loop rule, we apply our inference rule $Correct_{Loop2}$ shown in Fig.6 and therefore, the corrected dp4 is presented in Fig.7.

*2) **Access Violations Resolution** :* First, we try to correct the entire violation by removing misconfigured rules using an inference system. It deals with each $dp_i$ from the set EnV via changing the action of the rules that caused the violation in $dp_i$. To fix partial violations we use an inference system that allows to divide each partially misconfigured direct path $dp_i^{'}$, into two sets: 1) $dp_i^{inter}$ is the subset of paths that has the correct action as defined by the security policy and 2) $dp_i^{inter'}$ represents the subset of $dp_i^{'}$ that should be fixed.
To clarify this reasoning, we refer to our illustrative example, particularly the case PaV depicted by Fig.5. We consider $dp1^{inter} = dp1 \cap SP_a$ = the branch represented by the following values : $[@src_ip, @dest\_ip, port\_dest] = [10.0.1.0/24, 10.0.1.10, 80]$. Therefore, $dp1 = (dp1 \setminus dp1^{inter}) \cup (dp1 \cap dp1^{inter})$. Then, we divide this direct path into two sub paths where the first $dp1 \cap dp1^{inter}$ represents paths which are conform to SP and the second one (dp1 $\setminus dp1^{inter}$) is the totally violated path. Thus, PaV identified at dp1 of Fig.4 is resolved by adding two branches (dp1 and dp2) as shown in Fig.7.

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

1) $dom(dp_i) \subseteq SP_d$, we change the action of the matched rule in this $dp_i$ to **Drop**;
2) $dom(dp_i) \subseteq SP_a$, we replace the Empty action by **Exit**.

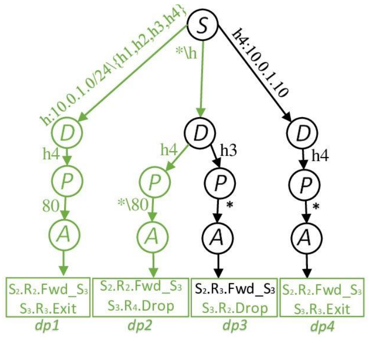


Fig. 7. the new corrected FeDD2.

## IV. Conclusion

We defined a comprehensive solution for enhancing the security of SDN configurations via detecting, analyzing and resolving defects. To ensure a high level of surety, our proposal relies mainly on FeDD, as data structure, and formal techniques, a set of inference systems, which allow formally proving security properties. We have implemented our new technique and the first computer experiments were very promising. We plan to experiment with more complex topologies, such as the Stanford backbone [25]. As a future work, we plan to consider techniques for verifying security policy and resolving violations in real time.

## 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] H.Hongxin, H.Wonkyu , A.Gail-Joon , and Z.Ziming. 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] N. McKeown, T. Anderson, H.Balakrishnan, G.M. Parulkar, L.L. Peterson, J.Rexford, S.Shenker, and S.T.Jonathan. Openflow: enabling innovation in campus networks. In: Computer Communication Review, pp. 69–74. ACM, 2008.

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

[21] Z.Peng, L.Hao , H.Chengchen, H.Liujia , X.Lei , W.Ruilong, Z.Yuemei. 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] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public, 2020