A Graph based Formalism for Detecting Flow Conflicts in Software Defined Network

Biswajit Halder[1], Mridul Sankar Barik[1], and Chandan Mazumdar[1]

[1]Jadavpur University, Kolkata, INDIA
Email: {biswajit.appl, mridulsankar, chandan.mazumdar}@gmail.com

***Abstract*—Software Defined Network (SDN) paradigm has revolutionized the way enterprise networks are designed by way of separating the control and data plane. It introduces a programmable network architecture which enables rapid and open innovation in different network functions that are allowed to install flow rules in forwarding elements via protocols like OpenFlow. Packet Processing also becomes easier and lucrative due to availability of packet information across different layers. But all these benefits may turn into great challenges because of the use of some features in OpenFlow itself. One of them, the *set_field* feature is widely used by network functions like firewall, router, load balancer etc. to modify packet header while in transit. Un-monitored use of this feature may cause packets to loop through switches, adversely affecting the network performance. Also, different network functions may install flow rules that directly or indirectly may violate each other. In this paper, we introduce a graph based formalism to detect forwarding rules that cause forwarding loop, direct or indirect flow violation. This helps network administrators to avoid possible security breaches, network congestion or even complete network failure.**

***Index Terms*—Software-Defined Network(SDN), Firewall, Directed Analytic Graph(DAG), Flow rule Violation, Forwarding loop**

I. Introduction

Traditional enterprise networks typically employ distributed set of firewalls and other security appliances which provide overall security of the network in a cooperative manner. But, in such scenarios, security appliances are generally configured in isolation and often without considering the global context of overall network security objectives. This practices cause device mis-configurations leading to major security vulnerabilities.

In recent years Software-Defined Networking (SDN) [7] paradigm has revolutionized the way enterprise networks are implemented and managed. SDN separates control plane from the data plane, which are otherwise sandwiched in a traditional network active devices such as a switch or a router. SDN switches are nothing but data plane elements which forward packets based on its installed flow rules. These flow rules are installed by a control plane element the SDN controller. In SDN network-wide visibility and centralized control over the underlying network active components enables easy deployment of network security functions such as firewalls and intrusion detection systems.

A firewall in SDN environment runs as a controller application like other applications such as routing, load balancing etc. Also, OF based packet filters can perform filtering on VLAN ID, VLAN PCP, Ethernet Type, IP ToS etc., that is not possible in traditional security devices. These controller based applications are comparatively much faster as only the first packet of a flow should pass through the corresponding application module, while rest of the packets are handled by OF switches. Even if a controller application module goes down for a while, the corresponding network function is not affected totally as OF switches continue to process packets based on available flow rules [16] [8].

However, OpenFlow lacks necessary features, which renders OF-based firewalls inherently stateless. Thus, making them vulnerable to unauthorized flow rule over-writes by different applications in switches resulting in firewall bypass. In SDN environment, the flow rules are dynamically modified by different controller applications and thus can become inconsistent with respect to the overall security policy of the network. Earlier solutions to tackle this problem used different abstractions such as header space, graphs etc. to represent the network state and then apply custom algorithms to infer any inconsistency. But as the network size grows, these approaches face severe restrictions in managing and analyzing the state information.

In this paper, we propose a graph based formalism for real-time detection of conflicts in SDN environment. This approach involves generating a directed graph from the current forwarding rule set from firewall and other applications. This graph effectively shows how the header content of a packet gets modified as it is forwarded through different OpenFlow devices. From this directed graph it is very easy to detect any direct or indirect flow rule violation and formation of forwarding loops. Also, we have defined a property graph data model for storing this graph in Neo4J graph database. We have used queries in Neo4J's built in Cypher graph query language to implement different policy violation detection scenarios. In section II we have briefly sketched the related works, section III shows basic working principle of SDN and issues that we have tried to solve followed by the proposed solution and performance analysis in section IV. And finally we conclude our work in section V.

II. Related Work

Since the beginning of this decade SDN has become a hot topic in computer network research. Many such efforts have proposed ideas to improve firewall performance, to detect and remove firewall policy misconfiguration, to find out anomaly in firewall policy and to find out conflicts in flow rules. Based on types of issues addressed we have categorized these works in three different groups.

- **Anomaly Detection:** Ehab S. Al-Shaer [3] proposed a user friendly framework for automatic discovery of policy anomalies in legacy firewalls, and also for anomaly-free policy editing for rule insertion, removal and modification. The same authors in another work [2] presented a way to discover inter firewall anomalies. Jeffrey and Samak [9] presented a model checking technique using Binary Decision Diagram(BDD) and satisfiability of Boolean formulae (SAT) for analyzing the behavior of firewall policy configurations, and reporting anomalies.
- **Detection of Flow Rule Violation:** FlowChecker [5] is a tool to verify misconfigurations of a single FlowTable of a switch. FortNOX [11] is another security mediation service that deals with flow (firewall policy) misconfiguration. FLOVER[13] is another solution to verify compliance of dynamically assigned rules with invariant security policy in real time. Header Space Analysis (HSA) [10] is a mechanism based on analysis of packet header to detect forwarding loop in traditional network. In another work Kordalewski et a. [6] has used HSA along with directed graph for detection of intra network forwarding loop in SDN environment.
- **Firewall Performance Optimization** Fulp et al. [14] proposed a solution for firewall performance optimization and integrity. In this work they addressed reordering of a firewall rule set to minimize the average number of comparisons to determine the action, while maintaining the integrity of the original policy.

For today's fast changing network detection of any kind of conflict in real time is an important aspect and use of graph database makes the process faster. The data representing complex interconnected relationships between flow rules is best captured by a graph data model. It is well recognized that relational databases fail frequently when dealing with large interconnected data. Graph databases are a class of NoSQL database that uses graphs as the underlying model for data representation and storage. Graph databases represent node adjacency via direct pointers. This avoids expensive join operations or other index lookups for graph traversal. Graph databases have been shown to be orders of magnitude faster than relational databases for graph traversal. Open Network Operating System (ONOS) [4] is an experimental distributed SDN control platform. It abstracts network view as graph data model and implements the model in Titan graph database with Cassandra key value store for distribution and persistence. Gravel is a graph database based controller and improves performance of earlier Ravel [15] approach which uses relational database. It stores entire network state in a relational database and individual controller applications are given abstraction of this state in form of SQL views. The key challenge is to orchestrate and manage these multiple views of different application via which they try affect the common network state. Effects of application updates to these higher level abstractions are reflected to the underlying network state via SQL triggers which implement custom application specific control functions.

In our implementation we have used Neo4J [1], a popular graph database that supports ACID (atomicity, consistency, isolation, durability) transaction properties usually found in relational databases. Moreover, Neo4J has a built-in query language Cypher, that allows retrieval of both explicitly defined information as well as information that can be implied (e.g., through graph traversal).

III. Basic Working Principle

A. Flow installation in SDN Environment

Each switch forwards packet based on the flow rules stored in it. Flow rules are nothing but forwarding rules of firewall or other applications translated in switch (data plane) understandable format. Policies are abstract or higher level description of what traffic should be allowed and what should not. Whereas, rules are formalized representation (a pattern/configuration) of policy. For example, "Allow all http request traffic from 10.0.0.1, port Any to 10.0.0.2, port 80." is a firewall/application policy and its corresponding forwarding rule is "10.0.0.1/Any 10.0.0.2/80 Allow".

Each flow rule contains an action field, which specifies the operation to be performed on the matched packet and the switch port through which it should be forwarded after execution of the said operation. On receiving a packet if the switch does not find any matching rule in its flow table, the packet information is forwarded to the OF controller where it passes through one or more applications and finally one flow rule corresponding to the packet is installed by the controller and associated action is executed on the packet and subsequent packets with same flow characteristics.

Figure 1 shows a simple topology with four host and three switches.

Sample firewall rules are given in table-1 below:

TABLE I
Sample firewall rules

Protocol	Src IP	Dst IP	Src. Port	Dst. Port	Action
TCP	10.0.0.1	10.0.0.2	Any	80	Allow
TCP	10.0.0.2	10.0.0.1	80	Any	Allow

When 10.0.0.1 sends a http request to 10.0.0.2 for the first time, the packet header information is forwarded to the controller by switch-1, as there is no matching flow rule initially. Firewall application at the controller matches the header information against firewall forwarding rules. As the matching forwarding rule allows the packet, the firewall application installs a flow rule in switch-1 to forward the

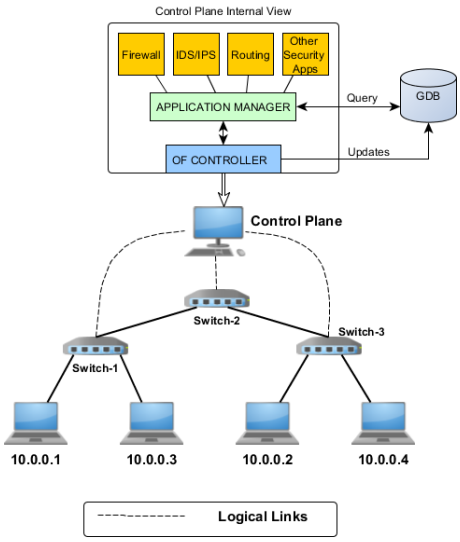

Fig. 1. Simple topology describing flow installation based on firewall rule

packet to a particular switch port which is determined by the controller based on the physical switch topology graph. The controller chooses the forwarding port such that the distance to destination through the identified port is minimum. Each flow rule generally contains the fields as shown in table-II, where one or more field(s) value may be wild card as per firewall application configuration. When an application wants to install a rule it requests controller with required credentials and controller installs flow in same way. The only difference is that credentials are forwarded by applications instead of switches.

TABLE II
FIELDS IN A FLOW RULE

Network Protocol	Source MAC Addr.	Destination MAC Addr.	Source IP Addr.
Destination IP Addr.	Source Port	Destination Port	*Actions*

Here, *Actions* = set of actions (like modifying content of a header field) that should be performed on the packet.

Similar actions take place when switch-2 and switch-3 receives the same packet.

B. Flow Conflicts

OpenFlow introduces some new challenges which are not seen in traditional networks. It introduces *set_field* option that allows modification of packet header by forwarding elements while in transit. This option is very useful for dynamic routing and other redirection mechanism. To implement NAT/PAT (Network/Port Address Translation) in SDN, routing applications need to modify packet header content and the *set_field* option becomes handy in this scenarios. In todays network where dynamic and automated control is very important, this option plays a major role. For example, based on traffic statistic, instead of delivering a packet directly, packet header can be modified and before delivering it to the intended receiver the packet can be made to pass through several security applications. DefenseFlow [12] is an example of such an application that uses packet redirection mechanism to prevent DDoS attacks. Moreover, instead of using a separate proxy server administrators can use OpenFlow based switches themselves to act as proxy server by using *set_field* option. But all this advantages of OpenFlow comes with problems like *forwarding loop* and *direct or indirect flow violation*. These problems can occur either in a single controller or in multiple controller scenario.

Table III shows set of forwarding rules for the example network of figure 2. For simplicity we have assumed that, for these rules only the source and destination addresses are modified. Our, proposed method is capable of handling any number of field modifications.

TABLE III
FIREWALL AND APPLICATIONS RULE SET

Rule #	Src $\rightarrow$ Dst	Action
R01	E $\rightarrow$ G	Forward
R02	C $\rightarrow$ G	set source address to A, forward
R03	E $\rightarrow$ B	Forward
R04	A $\rightarrow$ H	set source address to F, forward
R05	F $\rightarrow$ D	Forward
R06	E $\rightarrow$ G	Drop
R07	A $\rightarrow$ E	set destination Address F, forward
R08	B $\rightarrow$ D	set destination address to C, forward
R09	F $\rightarrow$ H	set destination address to C, forward
R10	B $\rightarrow$ G	set source address to C, forward
R11	A $\rightarrow$ F	set destination address G,forward
R12	A $\rightarrow$ C	Drop
R13	B $\rightarrow$ F	set source address to E, forward
R14	A $\rightarrow$ D	set source address to B, forward
R15	A $\rightarrow$ G	set destination address to D, forward
R16	F $\rightarrow$ C	Forward
R17	B $\rightarrow$ C	set destination address to G, forward
R18	E $\rightarrow$ F	Forward
R19	Any $\rightarrow$ Any	Forward

1) Direct Flow Violation: Direct Flow violation occur when each corresponding fields of one Forwarding rule are exactly the same except the action field. For example: A packet from 'E' to 'G' match both of the forwarding rules R01 and R06, but the actions are forward and drop respectively.

2) Forwarding Loop: Besides allow or drop options OpenFlow introduces many other actions related to packet forwarding. Source address, destination address, source port, destination port and many other layer 2/3/4 header fields can be modified using *set_field* action while it passes through OpenFlow enabled devises. In many cases, to cater to present needs, administrators or applications add forwarding rules without examining the current rule set. This can create forwarding loops which may lead to network congestion or even failure. For example, consider the rules R02, R07, R08, R10, R11, R14, R15 and R17 added by administrators and/or applications at different times.

Normally, switches forward a packet to a particular port only after executing matched flow rule actions on the packet. It decides on the output port based on MAC-to-port information

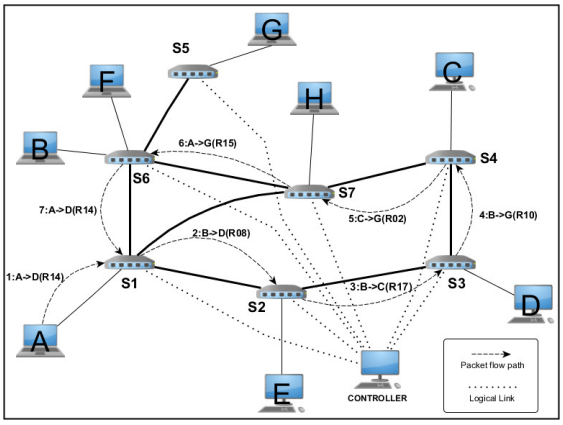

Fig. 2. Network Topology-1: Forwarding Loop

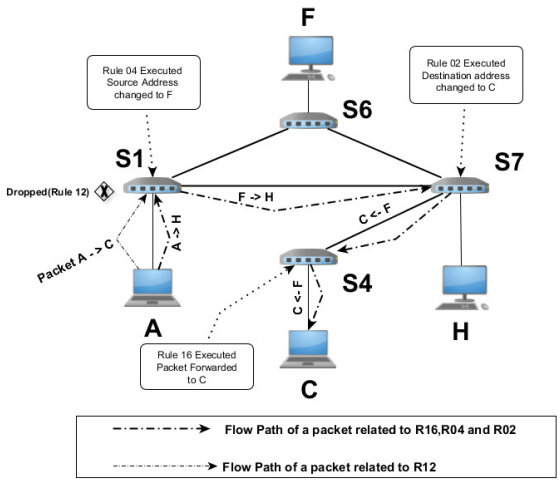

Fig. 3. Network Topology-2: Intra-Network Indirect policy Violation

which is pre-computed based on the shortest path to the destination. From figure 2 it is clear that when host A sends a packet to D rule R14 is executed in S1 and the modified packet with destination D and source B is forwarded to S2 (as shortest path from S1 to D passes through S2 next). Similarly Rule R08 is executed on S2 and modified packet with destination C is sent to S3 and so on. This sequence repeats again when switch S6 executes rule R15 and forwards the packet with source A and destination D to switch S1. This creates a loop. Too many packet from source A to destination D will cause congestion in the network.

3) Intra-Network Indirect Flow Violation: Indirect flow violation occurs when final action executed by a set of forwarding rules together conflicts with another forwarding rule's action. Figure 3 shows a topology which is part of the topology shown in figure 2. Rules R12, R04, R09 and R16 describe the scenario of indirect Flow violation. If host A sends a packet to Host C rule R12 is executed on switch S1 and the packet is dropped. But, when A sends a packet to H, Rule R04 is executed on S1. After modification of source address to F the packet is forwarded to S7 where rule R09 is executed and destination address is updated to C. After that the packet is forwarded to S4, which delivers it to host C as per rule R16. This is a clear violation of rule R12.

IV. Proposed Solution

In this paper, we propose a solution to the problems discussed in section III-B. It first generates a directed graph from the forwarding rules installed by different applications. We present an algorithm for this purpose. Then we show how graph queries in Cypher language can be used for detection of intra-network forwarding loop and intra-network indirect flow violation.

A. Graph Data Model

In our representation each graph node and edge satisfy the following properties:

- Each node in the generated graph (except two designated nodes 'DROP' and 'Forward'), its outgoing edge and target node of the edge together represents a forwarding rule.
- Each node contains all the field (except action) values of the concerned forwarding rule. If a rule is written based on source address, destination address, source port, destination port and protocol, every node contains a field related to all of these fields.
- Destination of each directed edge represent next state (i.e. whether the packet should be forwarded, should be dropped or should be modified. If the target of the edge is 'Drop' node then the packet should be dropped, if the target is 'Forward' node then the packet need to be forwarded without any modification and if the target is any node other than 'Forward' or 'Drop' then the packet will be forwarded after header modification.

Algorithm $GraphGen()$ generates a directed graph $G(V, E)$ from a set of rule R, where,

- $V = V_{in} \cup V_m \cup v_f \cup v_d$
- R_i is the i^{th} forwarding rule
- $R_{default}$ is the default forwarding rule i.e. the rule that defines what action will be taken against the packet that does not match any other forwarding rule.
- V_{in} is the set of nodes each of which represent one of the forwarding rule
- v_i^{in} is the node that represent the header content of the packet that matches rule R_i. Attributes of node v_i^{in} are same as the fields of the rules except the action field.
- V_m is the set of nodes that match with modified contents of each forwarding rule (if there is any header modification action).
- v_i^{out} is the node that represent modified header contents of a packet after executing action of a forwarding rule R_i on the packet. Attributes of node v_i^{out} is same as v_i^{in} with modified values in one or more attributes.
- v_f is a designated forward node that represent action forward. Value of each attribute is set to Any.

Algorithm 1: GraphGen(R)

input : Set of Forwarding Rules R
output: Directed Graph G=$\{V,E\}$

```
1  begin
2      Add node v_f
3      Add node v_d
4      for R_i ∈ R do
5          if v_i^in ∉ {V_in ∪ V_m} then
6              Add v_i^in in V_in
7          switch f_A(R_i) do
8              case forward do
9                  Add edge (v_i^in, v_f) to E
10             case drop do
11                 Add edge (v_i^in, v_d) to E
12             case modify do
13                 if v_i^out ∉ {V_in ∪ V_m} then
14                     Add v_i^out in V_m
15                 Add edge (v_i^in, v_i^out) to E
16     for v_p ∈ {V_in ∪ V_m} and (v_p × V) = ∅ do
17         if f_A(R_default) == forward then
18             Add edge (v_p, v_f) to E
19         else
20             Add edge (v_p, v_d) to E
```

- v_d is a designated drop node that represent action drop. Value of each attribute is set to Any.
- $E = \{(V_{in} \times V_m) \cup (V_{in} \times v_f) \cup (V_{in} \times v_d)\}$
- f_A: $R \rightarrow \{forward, drop, modify\}$. The action $modify$ indicates modification of header field using set_field.
- outdegree(N) : Number of outgoing edge(s) of node N.

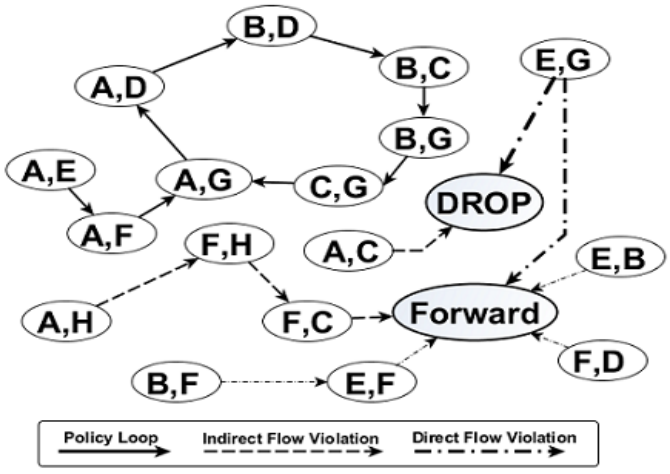

Fig. 4. Directed graph of for forwarding rule set mentioned in Table-III

Forwarding Loop detection is another major task. Directed graph generated by algorithm **DigraphGen(R)** for the example rule set of Table-III is shown in Figure-4. For ease of understanding, we have used different types of directed edges to show possible violations among forwarding rules.

In our implementation, we have executed Cypher queries on the directed graph generated by our algorithm for detecting forwarding loops and the set of rules that are responsible.

Query 1 - Detection of Direct Flow Violation

Objective is to check whether a node representing a rule has more than one successor. The following query finds paths from a node with Src_IP='E' and Dst_IP='G' to it's immediate neighbors and returns count of such paths. Clearly a count of greater than two for any rule indicates direct flow violation.

For a rule with Src_IP='E' and Dst_IP='G'

1) Find node n representing the rule and it's immediate neighbor x and store path for each x in p (line 1-2).
2) return number of such neighbor. Clearly neighbor more than two indicates direct violation (line 3).

```
1 match (n:packet)-[:MOD]->(x:packet)
2 where n.src='E' and n.dst='G'
3 return count(x)
```

Query 2 - Detection of Indirect Flow Violation

For a rule with Src_IP='A' and Dst_IP='C'

1) Find node $n1$ representing the rule and it's immediate neighbor x (line 1-2).
2) Find nodes $n2$ and $n3$, such that Src_IP of node $n2$ and Dst_IP of node $n3$ match with that of node $n1$ (line 4-5).
3) Find all paths $p1$ from $n2$ to $n3$ (line 7).
4) Find all successor y of node $n3$ whose source or destination does not match with that of x (line 9-10).
5) Compute path $p2$ from node $n2$ to y (line 12).
6) Compute path $p3$ from node $n1$ to x (line 14).
7) Return path $p2$ and $p3$ where path $p3$ represent the rule and $p2$ represent all possible path that indirectly violate the rule.

```
1  match (n1:packet)-[:MOD]->(x:packet)
2  where n1.src='A' and n1.dst='C'
3  with n1,x
4  match (n2:packet), (n3:packet)
5  where n2.src=n1.src and n3.dst=n1.dst
6  with n1,n2, n3,x
7  match p1=(n2)-[:MOD*..]->(n3)
8  with n1, n2, n3,x
9  match (n3:packet)-[:MOD]->(y:packet)
10 where x.src<>y.src or x.dst<>y.dst
11 with n1, n2, n3,x,y
12 match p2=(n2:packet)-[:MOD*..]->(y:packet)
13 with n1, n2, n3,x,y,p2
14 match p3=(n1:packet)-[:MOD*..]->(x:packet)
15 return p2,p3;
```

Query 3 - Detection of Loop

To check whether a node is part of any loop or not

For a rule with Src_IP='A' and Dst_IP='D'

1) Find node $n1$ representing the rule (line 1-2).
2) Find all possible loops L where $n1$ is part of them (line 4).

```
1 match (n1:packet)
2 where n1.src='A' and n1.dst='D'
3 with n1
4 match L=(n1:packet)-[:MOD*..]->(n1:packet)
5 return L
```

Query 4 - To check whether a path starting from any node has a loop or not

For a rule with Src_IP='A' and Dst_IP='E'

1) Match all node in the graph (line 1).
2) Compute all possible loop $L1$ in the network (line 3).
3) Find node $n2$ representing the rule (line 5).
4) Find all path from $n2$ to any node in any loop in $L1$ (line 7-8).
5) sort out those loops ($L2$) that are connected to $n2$ (line 10-11).
6) Return all path from $n2$ that has a loop (line 12).

```
1 match (n1:packet)
2 with n1
3 match L1=(n1:packet)-[:MOD*..]->(n1:packet)
4 with n1,L1
5 match (n2:packet) where n2.src='A' and n2.dst='E'
6 with n1,n2,nodes(L1) as ln
7 match p=(n2:packet)-[:MOD*..]->(x:packet)
8 where x in ln
9 with p, nodes(p) as sn
10 match L2=(y:packet)-[:MOD*..]->(y:packet)
11 where y in sn
12 return p, L2;
```

Table-IV provides brief idea about the performance of the proposed algorithm. One important observation is that, sometimes small loop detection or indirect flow violation detection may take more time, if the number of paths passing through the node is significant. In Table-IV, N_R = size of the rule set, N_L = number of rules causing a forwarding loop, N_V = number of rules responsible for causing a indirect flow violation.

TABLE IV
PERFORMANCE RESULTS

	Loop Detection Time (in ms)			Indirect Flow Violation Detection Time (in ms)		
N_R	N_L	T_{avg}	T_{max}	N_V	T_{avg}	T_{max}
500	5	3	4	3	948.2	1423
500	10	5.1	7	6	1693.6	1933
500	15	7.4	11	8	3450.12	3674
1000	5	4.4	8	3	3843	3967
1000	10	4.2	7	6	3815.6	4024
1000	15	6.4	12	8	3696	3654

V. CONCLUSION AND FUTURE WORK

OpenFlow brings innovation in network development and makes a network programmable. But it also introduces options for modification in packet header contents that causes serious problem like flow violations. In this paper, we have introduced a graph based formalism that generates a directed graph from available flow rules installed by different controller applications in SDN environment and checks for flow violations and forwarding loops. A graph database Neo4J is used to store the directed graph generated by our algorithm and Cypher queries are executed on it to detect different kind of flow violation scenarios. As a future extension of this work we intend to use this mechanism for detection of flow violations and forwarding loops in a multiple controller environment.

Acknowledgments

We wish to thank all the members of Centre for Distributed Computing, Jadavpur University for their continious support in this work. We would also like to thank Council of Scientific and Industrial Research (CSIR), Govt. of India, for financial support.

REFERENCES

[1] Neo4j graph database. https://neo4j.com/, Jul 2017.
[2] E. Al-Shaer and H. Hamed. Modeling and management of firewall policies. volume 1, pages 2–10, April 2004.
[3] E. S. Al-Shaer and H. H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, pages 17–30, March 2003.
[4] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.
[5] Z. Chen, Q. Gao, W. Zhang, and F. Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.
[6] R. R. David Kordalewski. A dynamic algorithm for loop detection in software defined network. Technical report, University of Toronto, December 2012.
[7] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, Apr. 2014.
[8] H. Hu, G.-J. Ahn, W. Han, and Z. Zhao. Towards a reliable sdn firewall. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, 2014. USENIX.
[9] A. Jeffrey and T. Samak. Model checking firewall policy configurations. In *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on*, pages 60–67, July 2009.
[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.
[11] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM.
[12] Radware. Ddos protection: Load balancing and application delivery solutions — radware. https://www.radware.com/, Feb 2017.
[13] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Model checking invariant security properties in openflow. In *2013 IEEE International Conference on Communications (ICC)*, pages 1974–1979, June 2013.
[14] A. Tapdiya and E. W. Fulp. Towards optimal firewall rule ordering utilizing directed acyclical graphs. In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pages 1–6, Aug 2009.
[15] A. Wang, X. Mei, J. Croft, M. Caesar, and B. Godfrey. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 5:1–5:7, New York, NY, USA, 2016. ACM.
[16] J. Wang, Y. Wang, H. Hu, Q. Sun, H. Shi, and L. Zeng. *Cyberspace Safety and Security: 5th International Symposium, CSS 2013, Zhangjiajie, China, November 13-15, 2013, Proceedings*, chapter Towards a Security-Enhanced Firewall Application for OpenFlow Networks, pages 92–103. Springer International Publishing, Cham, 2013.