Traffic Steering of Middlebox Policy Chain Based on SDN

Qichao He, Ying Wang, Wenjing Li, Xuesong Qiu
State Key Laboratory of Networking and Switching Technology
Beijing University of Posts and Telecommunications
Beijing, China
Email: {heqc, wangy, wjli, xsqiu}@bupt.edu.cn

***Abstract*—The delivery of services typically requires packets to be steered through a sequence of middleboxes to improve network security and performance. One constraint on the deployment of services is that middleboxes are tightly coupled to the physical network topology. As a result, ensuring successful deployment requires error-prone and complex low-level configurations. Software-Defined Networking (SDN) can eliminate the need to configure network devices manually to deploy services. However, in terms of steering middlebox-specific traffic in data plane, applying the existing capabilities supported by OpenFlow protocol may lead to incorrect forwarding decisions when there is a loop in the route used to steer traffic. In this paper, we present an implementation using tagging to discriminate different instances of the same packet arriving at the same ingress port on the same switch (i.e. the existence of the loop). Moreover, we propose an algorithm to judge the existence of the loop in a physical sequence of switches and decide which switches are responsible for adding tags. The experimental result demonstrates that our implementation can properly steer traffic through a specific sequence of middleboxes even when there are loops in forwarding path.**

Keywords—Middlebox; Network Management; Software-Defined Networking; Traffic steering

I. Introduction

The delivery of end-to-end services provided by operators often demands various service functions including traditional network appliances. Common examples of network appliances are firewalls, content filters, intrusion detection systems, deep packet inspection, web proxies, load balancers, network address translation (NAT) and wide area network (WAN) accelerators. All these network appliances are generally referred to as middleboxes or inline services because end users are often unaware of their existence in their traffic's path [1]. Dedicated middlebox hardware is widely deployed in enterprise networks to improve network security and performance. However, the composition of middleboxes specified by services remains a challenging task. The challenge stems from topological dependencies, configuration complexity and other aspects. The deployment of services is often coupled to network topology. For example, assuming that a service needs to route traffic through a firewall, it means the firewall is required to be placed on the network path (often via configuration of VLANs) or some modifications of the network topology have to be made to enable the steering. Such constraints introduced by topological dependencies are very likely to limit the network operator to a low utilization of service resources and reduce the flexibility of service deployment. This also restricts the size and the capacity of network resources. Typically, a middlebox required by a service is physically inserted on these topologies to ensure that traffic traverses the middlebox. However, this is not an optimal approach from the perspective of packet delivery. In addition, as more middleboxes with strict ordering are required by services, more complex configuration is needed to adjust to those alterations. Another effect of topological dependency is that traffic may be routed through some middleboxes no matter whether they need to be applied or not due to the complexity of network changes and device configuration [2].

Software-Defined Networking (SDN) offers a promising alternative for *middlebox policy enforcement* by using logically centralized management, decoupling the data and control planes, and providing the ability to programmatically configure forwarding rules [3]. The delivery of services typically requires packets to be steered through a sequence of middleboxes. SDN can eliminate the need to statically deploy service delivery using error-prone and complex low-level configurations. Nevertheless, in terms of steering middlebox-specific traffic in data plane, applying the existing capabilities supported by OpenFlow protocol [4] can lead to incorrect forwarding decisions when there is a loop in the route used to steer traffic, since flow-based forwarding rules cannot discriminate different instances of the same packet arriving at the same ingress port on the same switch. Switches may demand information that indicates the processing state of a packet to make accurate decisions.

In this paper, we present an implementation to address the issue of ambiguous forwarding in data plane. To tackle this problem, we utilize the VLAN ID in the VLAN tag to encode processing state. Furthermore, we propose an algorithm to judge the existence of the loop in a physical sequence of switches and decide which switches are responsible for adding tags. In order to specify the characteristics of traffic to be served and the sequence of middleboxes to be applied, we also define an XML-based data format along with a REST-based NBI for OpenFlow controllers.

The rest of this paper is organized as follows. Firstly, we review the related work in Section II. In Section III, we describe the issue relating to steering middlebox-specific traffic in data plane. After that, we present our implementation in Section IV. Experimental results are presented in Section V and this paper is concluded in Section VI.

II. Related Work

Given the need of evolution of current deployment model, the Internet Engineering Task Force (IETF) has formed a new working group to define and publish standards related to service function chaining. The Service Function Chaining (SFC) Architecture document [5] provides architectural concepts about service function chaining. A Service Function Chain (SFC) defines a list of abstract service functions and the order in which they are to be applied required by a service. A Service Function Path (SFP) is instantiated for an SFC by selecting specific service function instances that the packet will visit when it traverses the network. Middlebox policy chain is an instance of the concept of "service function chain".

The middlebox placement problem is conceptually similar to the VNF placement problem. In [6], the authors present a model for placing virtual network function in a scenario where physical hardware and instances of virtual network function provide services cooperatively. The execution time of the proposed algorithm remains less than 16 seconds, ensuring it can react to demand rapidly. Authors in [7] formulate the problem of network function placement as an integer linear programming problem. The formulation aims to determine the locations for placing virtual network function instances while at the same time minimize the resource utilization. Since a significant number of researches have been done in this topic, we do not focus on middlebox placement in this paper. However, the middlebox placement scheme is the input needed by our implementation, hence we define an XML-based data format and a REST-based NBI to specify the middlebox placement scheme.

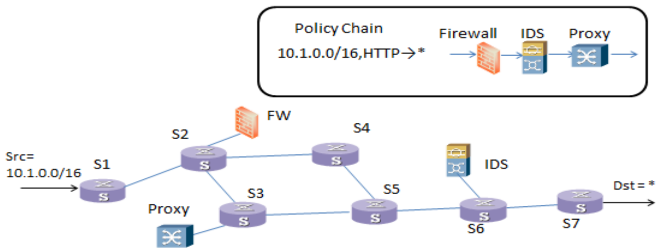
As for traffic steering among service functions, Paul Quinn and Jim Guichard [8] propose a single service-level data plane encapsulation format called Network Service Header (NSH), which acts as the SFC encapsulation component required by SFC Architecture. The NSH header has two fields for constructing a service path: the service path identifier (SPI) and the service index (SI). The SPI identifies a service path that interconnects the service functions. It provides a level of indirection between the service topology and the network transport. The SI identifies a packet's location within a service path, and it must be decremented by service functions after performing required services. However, NSH requires a control plane that conveys NSH related information, for example to instruct service function forwarders to map SPI/SI to network transport protocol. Besides, this header only works in NSH-aware network.

The approaches in [1], [3] do not use a special header to steer traffic. Aiming at reducing the memory footprint of flow table on each switch, StEERING[1] utilizes the pipeline with multiple tables introduced in OpenFlow 1.1 to implement a scalable and flexible architecture. This design, combined with metadata encoding service chaining, handles the integration of different types of traffic steering policies easily and efficiently. Nevertheless, it does not mention how to deal with the case when there is a loop in the route used to steer traffic. An approach for resolving the loop issue is proposed in [3]. SIMPLE[3] presents a SDN-based policy enforcement layer for middlebox-specific traffic steering. It utilizes the existing capabilities of OpenFlow protocol to push tags onto packets in order that the downstream switches can use these tags to know the processing state of each packet before deciding their forwarding actions. As a result, traffic can be steered through the desired sequence of middleboxes without mistake.

However, SIMPLE[3] does not go into the details of the implementation and neither does it propose an approach for judging the existence of the loop. As a supplement to this paper, our work provides an algorithm to detect the existence of the loop and decide which switches are responsible for adding tags when the loop exists. Besides, our work also presents an implementation utilizing the VLAN tag to annotate the processing state of each packet for traffic steering.

III. Problem Statement for Enforcing Middlebox Policy Chain in Data Plane

This section describes a problem associated with utilizing SDN for middlebox policy chain. In order to make the statement concrete, we give an example in Fig. 1. As shown in the figure, the administrator wants the policy chain Firewall-IDS-Proxy to be applied when HTTP traffic traverses the network.



Policy Chain	Physical Sequence
FW-IDS-Proxy	S1 S2 FW S2 S4 S5 S6 IDS S6 S5 S3 Proxy S3 S5 S6 S7
FW-IDS-Proxy	S1 S2 FW S2 S3 S5 S6 IDS S6 S5 S3 Proxy S3 S5 S6 S7

Fig. 1. Example to illustrate the requirement specified by a middlebox policy chain. For the middlebox policy chain: Firewall-IDS-Proxy, different physical sequences of switches and middleboxes that can implement the policy are shown in the table.

We illustrate the issue through the physical sequence of switches and middleboxes: $S1 \rightarrow S2 \rightarrow FW \rightarrow S2 \rightarrow S3 \rightarrow S5 \rightarrow S6 \rightarrow IDS \rightarrow S6 \rightarrow S5 \rightarrow S3 \rightarrow Proxy \rightarrow S3 \rightarrow S5 \rightarrow S6 \rightarrow S7$, which can be used to implement the middlebox policy chain for HTTP traffic. Another physical sequence also has similar issue. Obviously, a packet would visit the switch S6 three times. Each time S6 needs to decide which action to take: if the packet has already visited the FW, then forward it to IDS; if the packet has already visited the FW and IDS, then forward it back to S3 for Proxy; if the packet has already visited all three middleboxes, then send it to the destination. However, it is difficult for S6 to make a choice between the action of forwarding packet to IDS and the action of sending packet to the destination just based on the header fields and the ingress port property of the packet. The problem here is that even though the flow entries in S6 contain the valid forwarding actions, this may not be realizable because matching packet header fields against the match fields of corresponding flow entries in S6 cannot differentiate the processing state in the context. That is to say, only using source/destination IP address, source/destination MAC address, and ingress port in match fields of flow entry is not sufficient for this situation.

IV. Data Plane Design And Implementation

According to section III, if a switch performs packet lookups and forwarding just based on the packet header fields, it may cause ambiguous forwarding when there is a loop in the physical sequence. Therefore, a method using tags is presented in this section to solve the problem.

The presented implementation is based on the following prerequisites: The first thing is that middleboxes do not modify packet headers. The second thing is that another application performs a middlebox placement algorithm on the basis of processing load or even switch constraints (i.e., the number of forwarding rules a switch can support) to get the physical sequence of middleboxes used to implement the specified policy chain. Our module takes the information as input and translates it into forwarding rules to avoid ambiguous forwarding. Consequently, an XML-based data format and a REST-based NBI are defined for the input needed by our module. The introduction of the data format is presented in part B.

A. *Unambiguous Forwarding*

Looking at Fig. 1 again, when the same packet sent by both FW and Proxy is received on the same ingress port, S6 is unable to determine whether this packet is from FW or Proxy, which may result in incorrect forwarding decision. In other words, when a switch receives the same packet multiple times at the same ingress port, it needs to know which middleboxes this packet has traversed in the middlebox processing chain. Therefore, we can add tags to packet headers according to physical topology and the physical sequence of middleboxes to annotate a packet with its processing state.

When a controller is required to configure forwarding rules for steering traffic through the specified sequence of middleboxes, it first needs to check whether there is a loop. If the sequence needed for routing is loop free, it means each directional link appears at most once in the sequence, the use of packet header fields and ingress port in forwarding rules can identify the processing state correctly in this situation. If the controller checks out the existence of a loop in the physical sequence, the use of packet header fields and ingress port may not be enough to help related switches to know which middleboxes a packet has traversed. Therefore, a tag can be pushed onto a packet to address this problem. The added tag is called ProcState tag. ProcState tags can use VLAN ID, MPLS labels, or other unused fields in the IP header depending on the fields supported in the SDN switches. Here we choose VLAN ID as ProcState tags since VLAN tags are well supported by OpenvSwitch and most open source controllers such as Floodlight.

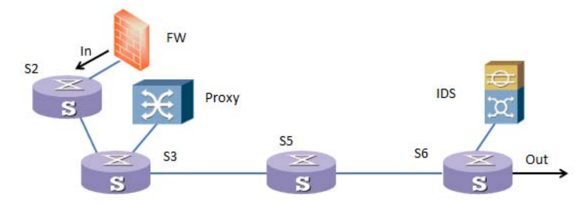
As is shown in Fig. 2, the switch S3 is responsible for adding the ProcState tags to packets from middlebox FW and the directly connected middlebox Proxy, so the corresponding rules in S3 are:

$\{\mathrm{HTTP, from\ FW}\} \rightarrow \{\mathrm{add\ ProcState} = \mathrm{FW, forward\ to\ S5}\}$;
$\{\mathrm{HTTP, from\ Proxy}\} \rightarrow \{\mathrm{add\ ProcStat} = \mathrm{Proxy, forward\ to\ S5}\}$.

We use push-VLAN action defined by OpenFlow protocol to achieve the addition of tags. The switch S6 can use the ProcState tags added by S3 to differentiate different instances of the same packet arriving at the same ingress port. Since middleboxes in the policy chain do not need to maintain or be aware of the context of the policy chain, S6 will remove the ProcState tag from a packet before forwarding it to middlebox IDS. The related forwarding rules inserted in S6 are as follows:

$\{\mathrm{HTTP, from\ S5, ProcState} = \mathrm{FW}\} \rightarrow \{\mathrm{remove\ ProcState, forward\ to\ IDS}\}$;
$\{\mathrm{HTTP, from\ S5, ProcState} = \mathrm{Proxy}\} \rightarrow \{\mathrm{remove\ ProcState, forward\ to\ destination}\}$.

In our implementation, we propose an algorithm to judge the existence of the loop in a physical sequence of switches and decide which switches are responsible for adding tags.



Traffic	Ingress Port	Proc State	Action List
HTTP	S2	-	1. add ProcState=FW 2. forward to S5
HTTP	S5	-	forward to Proxy
HTTP	Proxy	-	1. add ProcState=Proxy 2. forward to S5

Traffic	Ingress Port	Proc State	Action List
HTTP	S5	FW	1. remove ProcState 2. forward to IDS
HTTP	IDS	-	forward to S5
HTTP	S5	Proxy	1. remove ProcState 2. forward to destination

Fig. 2. Example of flow entries in S3 and S6 to illustrate the approach.

B. *XML-Based Data Format for Defining Input*

We have mentioned that our module is only responsible for installing forwarding rules to steer traffic through a physical sequence of middleboxes, it does not have the function of selecting middleboxes used to implement a specified policy chain. Therefore, an XML-based data format is defined for the input needed by our module. This part presents the semantics of the elements in this format.

Fig. 3 shows the detail of the XML-based data format. The root element *policies* can have one or more child elements named *policy*. Each *policy* element denotes a middlebox policy chain. A *policy* element contains an *identification* element and a *forwarders* element. The *identification* element stands for the traffic to be steered through this policy chain. The content of element *identification* is made up of four child elements. These four child elements are *sourceMAC*, *destinationMAC*, *sourceIP* and *destinationIP*, and the name of each child element implies its semantic meaning. Floodlight uses the contents of these four child elements to identify the source host and destination host in the network. If Floodlight identifies the source host and destination host successfully, then the locations of these two hosts in the network can be easily obtained, which are necessary in the calculation of the path used for steering. Moreover, the contents of these four child elements are also used in match fields of flow entry for matching. The *forwarders* element consists of one or more *forwarder* element. Each *forwarder* element represents a location for a middlebox. The order in which these *forwarder* elements appear implies the physical sequence of middleboxes. A *forwarder* element contains a *datapathid* element and a *port* element. These two child elements together identify the location of a middlebox in the network. The *datapathid* element must have a valid value of datapath ID defined by OpenFlow protocol. The *port* element must have a valid port number representing an OpenFlow port, a middlebox connects to the network via this port.

```
<?xml version="1.0" ?>
<policies>
  <policy>
    <identification>
       <sourceMAC>00:00:00:00:00:01</sourceMAC>
       <destinationMAC>00:00:00:00:00:02</destinationMAC>
       <sourceIP>10.0.0.1</sourceIP>
       <destinationIP>10.0.0.2</destinationIP>
    </identification>
    <forwarders>
       <forwarder>
          <datapathid>00:00:00:00:00:00:00:03</datapathid>
          <port>3</port>
       </forwarder>
       <forwarder>
          <datapathid>00:00:00:00:00:00:00:05</datapathid>
          <port>3</port>
       </forwarder>
       <forwarder>
          <datapathid>00:00:00:00:00:00:00:03</datapathid>
          <port>4</port>
       </forwarder>
    </forwarders>
  </policy>
  <policy>
       ......
  </policy>
  ......
</policies>
```

Fig. 3. The data format for defining input needed by our module.

C. Loop Judgment

A controller determines the sequence of switches used for steering according to the input in the format defined in Section B. As for the example in Fig. 4, the traffic sent from h1 to h2 is required to be steered through the physical sequence of middleboxes FW1-IDS1-Proxy1. Obviously, the route computed by controller is h1 → S3 → FW1 → S3 → S4 → S5 → IDS1 → S5 → S4 → S3 → Proxy1 → S3 → S4 → S5 → h2.

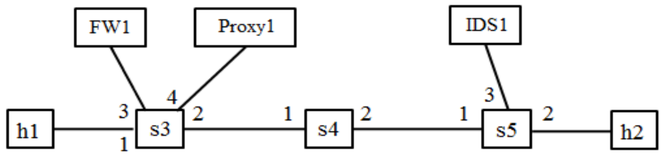


Fig. 4. Experimental physical topology

The numbers in the Fig. 4 are port numbers of switches. Here we use the notation *switch-port* pair to denote a port on a switch. For example, S3-3 represents the port 3 on switch S3. The port information is necessary because OpenFlow packets are received on an ingress port and processed by the OpenFlow pipeline which may forward them to an output port. We also define the concept of *Segment* as a path between two adjacent middlebox or host along the physical sequence. Based on this definition, the sequence of switches used for steering this traffic can be divided into four segments.

The segment 1 from h1 to FW1 can be expressed as S3-1→S3-3.

The segment 2 from FW1 to IDS1 can be expressed as S3-3→S3-2→S4-1→S4-2→S5-1→S5-3.

The segment 3 from IDS1 to Proxy1 can be expressed as S5-3→S5-1→S4-2→S4-1→S3-2→S3-4.

The segment 4 from Proxy1 to h2 can be expressed as S3-4→S3-2→S4-1→S4-2→S5-1→S5-2.

The number of notation *switch-port* in each segment must be an even number because each switch in each segment firstly appears with an ingress port to receive packets and secondly appears with an output port to forward packets. Our module needs to check whether there is a loop among these four segments before installing forwarding rules in switches. The key idea of judging the existence of the loop is that there are some directional links appearing more than once in these four segments. Each endpoint of a directional link is a port on a switch (i.e., a directional link is between two switches) so we can use a *switch-port* pair such as S3-2→S4-1 to identify a directional link.

Apparently, the path in segment 3 does not need tags because there is no directional link appearing in other segments repeatedly. Both segment 2 and segment 4 have the same part S3-2→S4-1→S4-2→S5-1, but their next *switch-port* notations after S5-1 are not the same. It means packets received on port 1 of switch S5 need to be forwarded to different ports. And S3-2 is the first one in this same part so upstream switch S3 is responsible for adding tags.

If two segments have the same *switch-port* notation indicating an output port, the next *switch-port* notations after it in the two segments must be the same because an output port on a switch connects to a unique ingress port on another switch. In the example above, S3-2 appears in segment two and segment four, so S4-1 would appear after it in both of them. Since both S4-1 and S5-1 indicate ingress ports on switches, we can just compare those ingress ports on switches in segments to reduce the number of comparisons. If two segments have some identical *switch-port* notations indicating ingress ports, we need to find the first one and the last one. The *switch-port* notation before the first one decides which switch is responsible for adding tags. The last one decides which switch is responsible for removing tags before sending to middleboxes.

Here we explain the purpose of finding the first identical *switch-port* notation indicating an ingress port and the last one. Assuming the same packet needs to be steered through three segments in Fig. 5. As illustrated in Fig. 5, the direction of the three segments is from left to right. The numbers in the Fig. 5 are port numbers of switches. These three segments all have some identical *switch-port* notations. We can obtain that the first identical *switch-port* notation indicating an ingress port is the port 4 on switch S3 based on the comparison of the first segment and the second segment, so S2 is responsible for adding tags to packets from middlebox A and middlebox C. Similarly, we also know S1 is responsible for adding tags to packets from middlebox A and middlebox E after comparing the first segment and the third segment. As a result, both S1 and S2 need to add tags to packets from middlebox A, it is not reasonable. Since S1 is before S2, we choose S1 to execute the action of adding tags to packets from middlebox A. S2 is still responsible for adding tags to packets from middlebox C. S2 uses the packet ingress port to distinguish a packet coming from middlebox A or middlebox C. So a segment needs to compare with all the other segments to choose the earliest one of those first identical switch-port notations indicating ingress ports. On the contrary, a segment needs to find out the latest one to decide a switch, which performs the action of removing tags from packets.

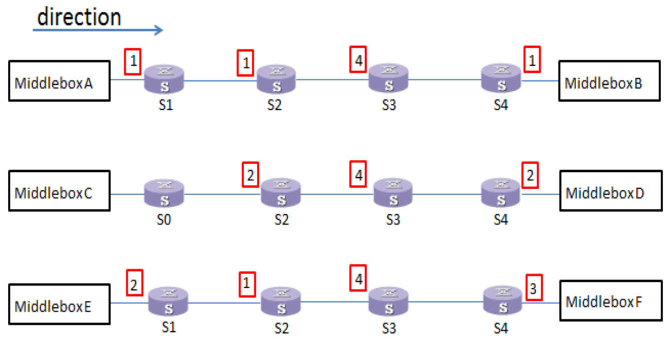


Fig. 5. Example to illustrate the purpose of finding the first identical switch-port notation indicating an ingress port and the last one.

The *NodePortTuple* class represents a port on a switch (i.e. *switch-port*). An instance of *List<NodePortTuple>* class stands for a sequence of *switch-port* pairs in a segment. The method *size()* of *List<NodePortTuple>* class returns the number of *switch-port* in a segment represented by an instance. The notation *list1[i]* denotes an element with index value of *i* in a sequence. The method *contains()* of *List<NodePortTuple>* class returns true if this sequence contains a specified element. And the method *indexOf()* returns the index of the first occurrence of a specified element in this sequence.

Algorithm Calculate the first and last identical *switch-port* indicating ingress ports between two segments

Input: List<NodePortTuple> list1, List<NodePortTuple> list2, list1.size() < list2.size()
Output: indexes of the first and last identical *switch-port* indicating ingress ports in each list
Procedure:

```
01: first1, last1, first2, last2 = 0
02: M = list1.size()
03: for (i=0; i < M; i=i+2)
04:     if list2.contains(list1[i]) && list2.indexOf(list1[i]) is
          odd
05:         first2 = list2.indexOf(list1[i])
06:         first1 = i
07:         for (j = 1; j < M - i; j++)
08:             if list1[i + j] != list2[first2 + j]
09:                 last1 = i + j - 1
10:                 last2 = first2 + j - 1
11:                 return
12:         end for
13:     end if
14: end for
```

V. Experimental Result

In this section, we conduct an experiment to show the result of our implementation. We use the topology in Fig. 4 as our experimental physical topology. The controller used in this experiment is Floodlight [9]. Our module is built and runs on top of it. Mininet [10] is used to create an SDN network. The version of OpenFlow protocol is 1.3.

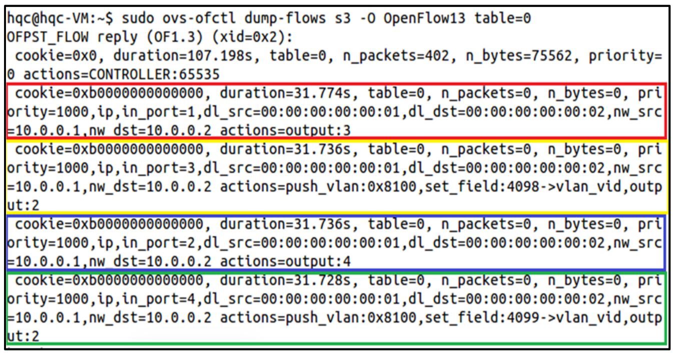
Assuming the administrator wants to route the traffic sent from h1 to h2 through the policy chain Firewall-IDS-Proxy. An application performs a middlebox placement algorithm, obtains the physical sequence of middleboxes FW1-IDS1-Proxy1 used to implement this policy chain and transmits the result to the controller via a REST-based NBI, using the data format designed in part B. The content of this file is displayed in Fig. 3. As illustrated in Fig. 3, the h1's IP address and MAC address are 10.0.0.1 and 00:00:00:00:00:01. The h2's IP address and MAC address are 10.0.0.2 and 00:00:00:00:00:02. The datapath IDs of switches S3, S4, S5 are 00:00:00:00:00:00:00:03, 00:00:00:00:00:00:00:04, 00:00:00:00:00:00:00:05. The numbers in the Fig. 4 are port numbers of switches. All this information is required to generate flow entries used for steering traffic.

Our module takes the information as input, and computes a route to steer the traffic through FW1, IDS1 and Proxy1 specified by the middlebox placement module. The computed route has been illustrated in Section IV-C, it is h1 → S3 → FW1 → S3 → S4 → S5 → IDS1 → S5 → S4 → S3 → Proxy1 → S3 → S4 → S5 → h2 . Then the module performs the function of judging loop based on the route and installs the corresponding flow entries in these switches.

To evaluate the effectiveness of our proposed approach, we observe the experimental results from two aspects. Firstly, we examine the flow entries in related switches to confirm that whether our module generates and installs flow entries in proper switches automatically according to the middlebox policy chain. Secondly, we observe whether the relevant flow entries installed by our module play a role in steering the corresponding traffic. The OpenFlow protocol defines that each flow entry has its counters (e.g. packet counter), and updates them when packets are matched. In addition, the command *ping* has an option *-c*, this option provides the ability to set the number of packets that are sent by a host. Therefore, we can set the option *-c* to an exact value when execute the command *ping*, then check the packet counter of each flow entry to see whether the value of each counter equals to the specified value. Now that we have elaborated on our verification method, then we need to verify whether it can solve the problem in the data plane.

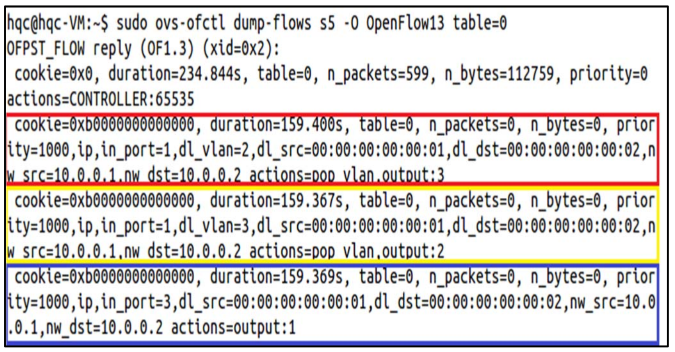
Fig. 6 shows all the flow entries of S3. The flow entries installed by our module are marked in rectangles of different colors. The flow entry in red rectangle forwards packets from h1 to FW1. The flow entry in yellow rectangle pushes a new VLAN tag with VLAN ID 2 onto the packet from FW1, and forwards it to S4. The decimal number after the word *set_field* contains the OFPVID_PRESENT bit (0x1000) defined by OpenFlow, we need to remove the bit to get the value of VLAN ID. The flow entry in blue rectangle forwards packets from S4 to Proxy1. The flow entry in green rectangle pushes a new VLAN tag with VLAN ID 3 onto the packet from Proxy1, and forwards it to S4.

The flow entries in S5 installed by our module are displayed in Fig. 7. The flow entry in red rectangle removes VLAN tag from the packet coming from S4 with VLAN ID 2 (*dl_vlan=2* in Fig. 7), and forwards it to IDS1. The flow entry in yellow rectangle removes VLAN tag from the packet coming from S4 with VLAN ID 3 (*dl_vlan=3* in Fig. 7), and forwards it to h2. The flow entry in blue rectangle forwards packets from IDS1 to S4.



```
hqc@hqc-VM:~$ sudo ovs-ofctl dump-flows s3 -O OpenFlow13 table=0
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=107.198s, table=0, n_packets=402, n_bytes=75562, priority=
0 actions=CONTROLLER:65535
 cookie=0xb000000000000, duration=31.774s, table=0, n_packets=0, n_bytes=0, pri
ority=1000,ip,in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src
=10.0.0.1,nw_dst=10.0.0.2 actions=output:3
 cookie=0xb000000000000, duration=31.736s, table=0, n_packets=0, n_bytes=0, pri
ority=1000,ip,in_port=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src
=10.0.0.1,nw_dst=10.0.0.2 actions=push_vlan:0x8100,set_field:4098->vlan_vid,outp
ut:2
 cookie=0xb000000000000, duration=31.736s, table=0, n_packets=0, n_bytes=0, pri
ority=1000,ip,in_port=2,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src
=10.0.0.1,nw_dst=10.0.0.2 actions=output:4
 cookie=0xb000000000000, duration=31.728s, table=0, n_packets=0, n_bytes=0, pri
ority=1000,ip,in_port=4,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src
=10.0.0.1,nw_dst=10.0.0.2 actions=push_vlan:0x8100,set_field:4099->vlan_vid,outp
ut:2
```

Fig. 6. Flow entries in S3 before matching traffic.



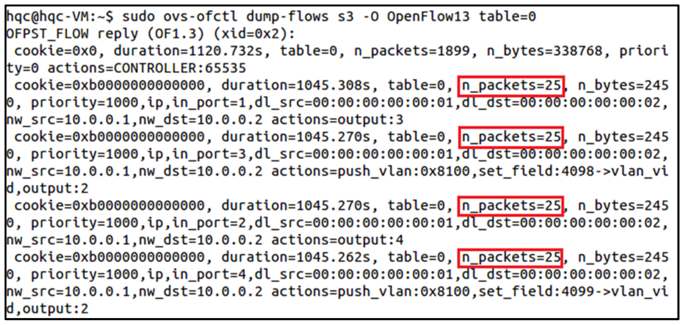
```
hqc@hqc-VM:~$ sudo ovs-ofctl dump-flows s5 -O OpenFlow13 table=0
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=234.844s, table=0, n_packets=599, n_bytes=112759, priority=0
actions=CONTROLLER:65535
 cookie=0xb000000000000, duration=159.400s, table=0, n_packets=0, n_bytes=0, prior
ity=1000,ip,in_port=1,dl_vlan=2,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,n
w_src=10.0.0.1,nw_dst=10.0.0.2 actions=pop_vlan,output:3
 cookie=0xb000000000000, duration=159.367s, table=0, n_packets=0, n_bytes=0, prior
ity=1000,ip,in_port=1,dl_vlan=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,n
w_src=10.0.0.1,nw_dst=10.0.0.2 actions=pop_vlan,output:2
 cookie=0xb000000000000, duration=159.369s, table=0, n_packets=0, n_bytes=0, prior
ity=1000,ip,in_port=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0
.0.1,nw_dst=10.0.0.2 actions=output:1
```

Fig. 7. Flow entries in S5 before matching traffic.

Before executing the command *ping*, the decimal number after the word *n_packets* in each flow entry is 0, shown in Fig. 6 and Fig. 7. Then we execute command *h1 ping -c 25 h2* for a test. The result after executing the command is shown in Fig. 8 and Fig. 9, the decimal number after the word *n_packets* in each flow entry is 25. It means each flow entry installed by our module matches 25 packets, and S5 can make correct decision between the action forwarding to IDS1 and the action sending to h2. It also demonstrates that our implementation can properly steer traffic through a specific sequence of middleboxes even when there are loops in forwarding path.

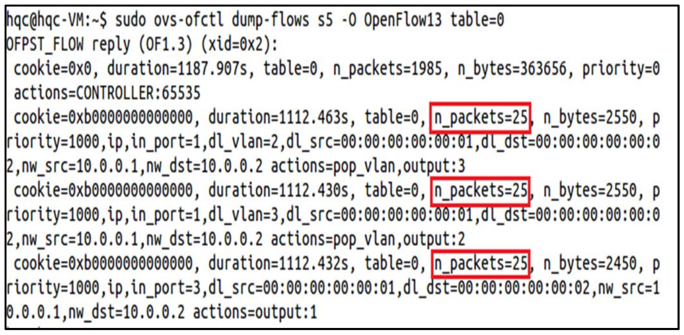
VI. CONCLUSION

In this paper, we present an implementation for solving an issue related to steering middlebox-specific traffic in data plane. To this end, the VLAN ID in the VLAN tag is utilized to encode processing state, and an XML-based data format is defined to specify the middlebox policy chain as the input of our method. Furthermore, we propose an algorithm to judge the existence of the loop in a physical sequence of switches and decide which switches are responsible for adding tags. The experimental result demonstrates that our implementation can properly steer traffic through a specific sequence of middleboxes even when there are loops in forwarding path.



```
hqc@hqc-VM:~$ sudo ovs-ofctl dump-flows s3 -O OpenFlow13 table=0
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=1120.732s, table=0, n_packets=1899, n_bytes=338768, priori
ty=0 actions=CONTROLLER:65535
 cookie=0xb000000000000, duration=1045.308s, table=0, n_packets=25, n_bytes=245
0, priority=1000,ip,in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,
nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:3
 cookie=0xb000000000000, duration=1045.270s, table=0, n_packets=25, n_bytes=245
0, priority=1000,ip,in_port=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,
nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=push_vlan:0x8100,set_field:4098->vlan_vi
d,output:2
 cookie=0xb000000000000, duration=1045.270s, table=0, n_packets=25, n_bytes=245
0, priority=1000,ip,in_port=2,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,
nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:4
 cookie=0xb000000000000, duration=1045.262s, table=0, n_packets=25, n_bytes=245
0, priority=1000,ip,in_port=4,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,
nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=push_vlan:0x8100,set_field:4099->vlan_vi
d,output:2
```

Fig. 8. Flow entries in S3 after matching traffic.



```
hqc@hqc-VM:~$ sudo ovs-ofctl dump-flows s5 -O OpenFlow13 table=0
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=1187.907s, table=0, n_packets=1985, n_bytes=363656, priority=0
 actions=CONTROLLER:65535
 cookie=0xb000000000000, duration=1112.463s, table=0, n_packets=25, n_bytes=2550, p
riority=1000,ip,in_port=1,dl_vlan=2,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:0
2,nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=pop_vlan,output:3
 cookie=0xb000000000000, duration=1112.430s, table=0, n_packets=25, n_bytes=2550, p
riority=1000,ip,in_port=1,dl_vlan=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:0
2,nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=pop_vlan,output:2
 cookie=0xb000000000000, duration=1112.432s, table=0, n_packets=25, n_bytes=2450, p
riority=1000,ip,in_port=3,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=1
0.0.0.1,nw_dst=10.0.0.2 actions=output:1
```

Fig. 9. Flow entries in S5 after matching traffic.

VII. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (61501044).

REFERENCES

[1] Ying Zhang et al., "StEERING: A software-defined networking for inline service chaining," 2013 21st IEEE International Conference on Network Protocols (ICNP), Goettingen, 2013, pp. 1-10.

[2] P. Quinn and T. Nadeau. "Problem Statement for Service Function Chaining", IETF Request for Comment:7498, April 2015.

[3] Z. A. Qazi, C.-C. Tu, L. Chiang et al., Simple-fying middlebox policy enforcement using sdn. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, pages 27-38, 2013.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, 2008.

[5] J. Halpern and C. Pignataro. "Service Function Chaining (SFC) Architecture", IETF Request for Comment: 7665, October 2015.

[6] H. Moens and F. D. Turck, "VNF-P: A model for efficient placement of virtualized network functions," 10th International Conference on Network and Service Management (CNSM) and Workshop, Rio de Janeiro, 2014, pp. 418-423.

[7] A. Mohammadkhan, et al., "Virtual function placement and traffic steering in flexible and dynamic software defined networks," The 21st IEEE International Workshop on Local and Metropolitan Area Networks, Beijing, 2015, pp. 1-6.

[8] P. Quinn and J. Guichard, "Service Function Chaining: Creating a Service Plane via Network Service Headers," in IEEE Computer Journal, vol. 47, no. 11, pp. 38-44, Nov. 2014.

[9] Floodlight. [Online]. Available: http://www.projectfloodlight.org

[10] B. Lantz, B. Heller, and N. McKeown, "A network in a Laptop: Rapid Prototyping for Software-Defined Networks," in HotNets, 2010.