Detecting Malicious Activities using Veriflow

Akshatha Kadri[1], Nikhita Kataria[2]

[1]akshatha.kadri@gmail.com
[2]nikhitakataria@gmail.com

Abstract

This paper aims to detect malicious activities present in SDN by combining the concepts from two state of the art solutions VeriFlow and Sphinx. The presented solution establishes a two way communication between SDN controller and Veriflow. It also adds the capability to receive and parse TCP/UDP packets and detect attacks like ARP spoofing, network DDOS, firewall bypass, and fragmentation attack.

Keywords: SDN, Openflow, VeriFlow, Mininet, POX

I. INTRODUCTION

Malicious activities have become a norm in SDN. It is becoming increasingly difficult to detect the different types of attacks happening in the network. In the past work, VeriFlow[1] has been presented as a powerful solution to detect anomalies by maintaining flow graph statistics and classifying the network into equivalence classes on the basis of new rules being introduced in the network. It leverages the communication established using OpenFlow messages. Though, VeriFlow has intelligence to classify and detect anomalies using forwarding rules, it does not have the capabilities to proxy every packet flowing through the network. In our work, we construct a stripped down version of an SDN controller running using POX libraries. We proxy every *packet_in* message exchange happening to and fro from the controller. This empowers existing work of VeriFlow to detect a new variety of attacks such as ARP Spoofing, Network DDOS, Firewall Bypass, Fragmentation attacks etc.

II. ARCHITECTURE

VeriFlow provides capabilities to act as a proxy between the hosts, controller and the switches existing in the SDN. A SDN network setup using VeriFlow involves following components illustrated in Figure 1: *Controller*: An intelligent and strategic control point to manage flow of packets in a network.
Topology: Actual connection of switches, routers and host/end machines to setup a full fledged network or a virtual network setup by tools like Mininet.
Proxy channel from Network to Controller: An interface to intercept messages sent to the controller.
Model/Protocol: OpenFlow communication protocol to model the messages to access the forwarding plane in a network.

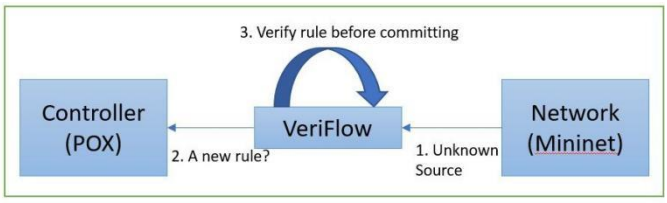

Figure 1: VeriFlow Actions Before Our Project

In this work we establish a communication channel from *Controller to Veriflow* and *equip VeriFlow with per packet header information.* As part of this communication, we forward *packet_in* openflow messages from controller to VeriFlow. OpenFlow version 1.0 integrated with Veriflow has following structure of a *packet_in* message where data encapsulates the ethernet frame with IP header which is 32 bit aligned from the whole structure:

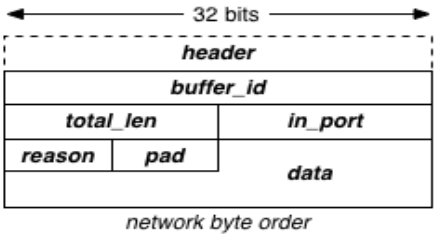

Figure 2: Structure of packet_in message

With the modified communication channel in order to detect attacks, the control flow will look similar to following figure:

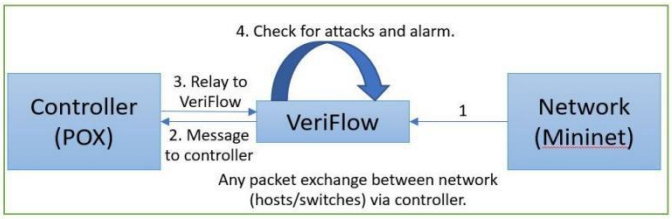

Figure 3: VeriFlow Actions After Our Project

III. DETECTING ATTACKS

The data redirected from controller to VeriFlow is an ethernet frame that has to be parsed to extract the information present in the header of the data packets.

We are extracting the data in the below format:

[ARP REPLY hw:1 p:2048 00:00:00:00:00:10>00:00:00:00:00:02 10.0.0.4>10.0.0.5]

The first field here is the ARP protocol used to determine IP to MAC address mappings during network bootstrap. “hw:” is the hardware type as specified in ARP Header in RFC 826, “p:” states the protocol type (2048 stands for IPv4 as per IEEE 802 network specifications[11]), next pair delimited by ‘>’ is the source and the destination mac followed by source and destination IP’s delimited by same character). Notice since this an ARP REPLY, network as well as physical address is available in the header.

The next step is to detect ARP spoofing attacks using the data we obtained at Veriflow.

A. ARP spoofing attacks

Address Resolution Protocol is used by Network layer protocols to obtain the MAC address of a host/router in the network. It is a broadcast message sent

to all the machines(hosts and routers) in the network. The host computer/router to whom the IP address belongs will respond with an ARP reply and thus completing the communication.

Issue with this set up is that any malicious entity can send the ARP response with its Mac address, thus poisoning the ARP cache and the topology view of the network in controller.

We detect such attacks by defining 3 cases

- There is a response when there was no request sent for the particular IP
- There are more number of ARP replies than the number of ARP requests sent
- There are multiple responses and Mac address is different in these responses.

B. Topology

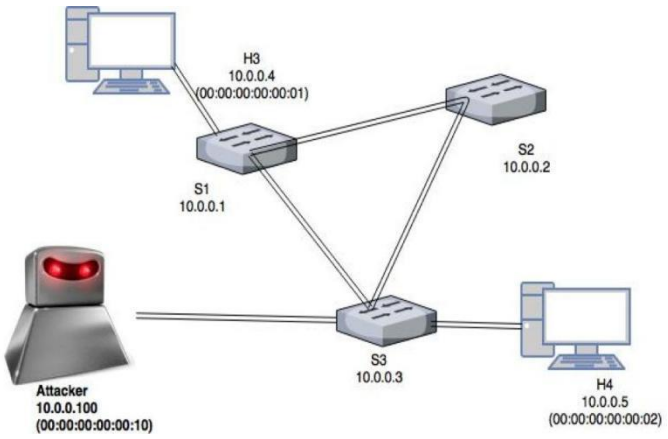

Figure 4: Topology of the network

The basic network topology is as shown in Figure 4. We are using mininet to simulate a network with 3 switches and 2 hosts. There is a single malicious entity sending ARP reply for the 3 cases described above. There are 5 links connecting the switches and hosts together. The malicious user can be connected to the network using any of these switches.

After encoding the topology into mininet, we start mininet. Since the network is coming up for the first time, ARP requests are sent to get the network topology. We configure POX controller to also send a few packets that mimic ARP poisoning attacks. The configuration is is done using following python code where attacks are performed in a round robin manner:

```
try:
        S = socket.socket(socket.AF_INET,
except:
socket.SOCK_STREAM);
S.connect(VeriFlow_IP, VeriFlow_Port);
```

The key to the map is the IP address. Specifically,

- destinationIP address when we are parsing ARP request

```
spoofed_packet = “” global count
If “ARP REPLY” in str(packet.payload):
        If count == 0:
                Print “Sending spoofed src ip”
                new_pack = str(packet.payload)
                /* Spoof IP in current packet_in */ spoofed_packet =
                    self.change_src_ip(new_pack) count = 1
```

```
            s.send(spoofed_packet) elif count = 1:
            Print "Sending duplicate replies"
            count = 2
            spoofed_packet= str(packet.payload) count = 1
            s.send(spoofed_packet) sleep(1) s.send(spoofed_packet) sleep(1) s.send(spooed_packet)
    elif count = 2:
            Print "Sending spoofed src ip"
            new_pack = str(packet.payload)
            /* Spoof MAC in current packet_in */ spoofed_packet =
                self.change_src_mac(new_pack)
            Count = 1 s.send(spoofed_packet)
    else:
            s.send(str(packet.payload))
```

Code Segment 1: Python Code for ARP Spoofing

In the above code segment, VeriFlow_IP and VeriFlow_Port are the ip address and port specified while starting VeriFlow. We use sleep calls in Code Segment 1 because the POX controller takes some time to deallocate the previous packet_in buffers and allocate buffers for new *packet_in* messages. If sleep is not used, payload in *packet_in* messages is merged, yielding wrong results. Finally, to enable VeriFlow to detect ARP spoofing, we create a structure ArpChecks and add an unordered_map<string, ArpChecks> as shown in Code Segment 2.

- Source IP address when we are parsing ARP reply

A counter for request, a counter for reply and corresponding Mac address is maintained in the ARPChecks to detect attacks.

Case 1: The ARP reply does not have corresponding ARP request.

This is detected by checking if an entry exists in the map structure. Future work might be to add time information and check against it.

Case 2: The ARP reply count is greater than ARP request count.

We detect this by checking the counters in the map data structure.

Case 3: Duplicate ARP reply with different Mac address This is detected by checking the entry we made for the previous MacAddress and comparing it with the current Mac address.

```
struct ArpChecks {
        int arpReqCount; int arpRespCount char * macAddr;
};

unordered_map<string,struct ArpChecks*> arpDataMap
```

Code Segment 2: Data Structure

This datamap is updated and reports the attacks using following piece of code:

```
   if(!isRequest) { if(arpDataMap.find(srcIP) ==
arpDataMap.end())
      {
```

```
            /* Alert Admin */
        } else {
            arp = this->arpDataMap.find(srcIP); if ((arp->second->arpReqCount)<
                (arp->second->arpRespCount)) {
                    /* Alert Admin */
            } else if(arp->second->macAddr != NULL && strncmp(arp->second->macAddr,
                            macAddr,17)!=0) {
                /* Alert Admin */


            } else {
              arp->second->arpRespCount++;

arp->second->macAddr=strdup(macAddr);
}
} else {
              if (this->arpDataMap.find(srcIP) == this->arpDataMap.end()) {
                    arp = newArp(struct ArpChecks); arp->arpReqCount = 0;
                     arp->arpRespCount=0; arp->macAddr=NULL; arpDataMap[srcIP] = arp;
              } else {
                     arp =  arpDataMap.find(srcIP); arp->second->arpReqCount++;
            } /* end of else */
        } /* end of else */
    } /* end of if for isRequest */
```

Code Segment 3: Code to categorise alert traffic

In the above code segment, there are three conditions where administrator is presented with an alert ***Case 1:*** ARP Spoofing detected with no entry for src. ***Case 2***: ARP Spoofing detected with more responses than requests.
Case 3: Different Mac Addresses received as part of the replies.

The second part of the code fragment is used to update or add a mac to ip mapping in the unordered hashmap and to update the number of arp replies seen per request. The later data is specifically added to detect *Case 2* specified above.

C. Fragmentation attack detection

IP Packets can be used to disguise TCP packets from IP filters. There are 3 types of Fragmentation attacks for security consideration:

- *Tiny Fragment attack* as stated in RFC 1858
  It is possible to make IP packets headers small enough to force some of the TCP header fields to the next fragment, hence the filter rules that search for a specific pattern does not match.

  STD 5, RFC 791 states:
  Every internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet

header may be up to 60 octets, and the minimum fragment is 8 octets.

To counter this variant of the attack, we add the check for minimum IP length check.

- *Tiny First Fragment attack*
  It is possible that the TCP header is very small in the first fragment. This indicates that the packet was not properly formed since the minimum length for TCP header is 20 bytes or the packet is sent having malicious intent. This indicates malformed packet and possible attack. We flag such packets and drop the same from the network.
- *Large Fragment attack*
  If a fragment is too large, it can cause the system to crash. It is more widely known as Ping Of Death. Attacker can use many small fragments which reassemble at the destination to form a packet that is larger than the allowed size for an IP packet. Although this issue is obsolete because of the updates in operating systems, there needs to be a check in the system to not allow and drop such packets. We implement functionality to filter this kind of packets using simple checks and drop such packets when found.

We examine this information using following conditions:

Condition 1:
IF More_Fragment_Flag is set in packet_in payload and length < Minimum_IP_HEADER_Length:
/* Report it as a tiny fragment attack */
drop_fragment();

Condition 2:
IF More_Fragment_Flag is NOT set in packet_in payload and length < Minimum_IP_HEADER_Length:
/* Report it as a tiny fragment attack */
drop_packet();

Condition 3:
IF More_Fragment_Flag is set in packet_in payload and length > Maximum_IP_Header_length:
/* Report it as a large fragment attack */
drop_packet()

Condition 1 detects the scenario where a fragment with a valid header is received with length less than the minimum expected length. Note only the first fragment contains the header information while rest just contain the fragmentation id, data, length of data and more fragment flag information. In this case, the fragment is dropped which prevents the reassembly of packets at the destination. *Condition 2* detects where the packet has no fragments but has length less than the minimum required header length. *Condition 3* detects the headers which have length more than the maximum size allowed for an IP Packet.

D. Statistics Based Prevention

A network denial of service is when there is unusually large number of packets that consume the network bandwidth and limit normal service to genuine users. With Veriflow, it becomes easy to detect such attacks since, every new packet will consult the controller about the incoming packet and establishes a route before routing the packet to the determined path. We assume that there are large number of unknown

hosts who are simultaneously sending a burst of packets into the network (or a specific IP). Since Veriflow is aware of the network state, we can check to see the number of new equivalence class generated. This means that we can detect DOS attacks using Veriflow by simply monitoring the new classes that are generating. We extend this idea and compute the moving average of number of classes created. If there is an unusually large number of new classes generated, we announce that there is a denial of service attacks ongoing. The algorithm is simple.

Algorithm:

For every new Rule

1. *Increment the number of packets*
2. *For every interval of designated time*

   -Check if the number of packets generated

   If (new_packet_cnt-old_packet_cnt > threshold)

   // Alert the controller about the DOS attack

   -Assign the new value to the packet count and update the time.

Using the algorithm presented above, we can detect a Network Denial of service attack by using the number of new rules generated. The packets can be sent for further evaluation and/or DOS defenses like ASV can be activated. This logic however does not safeguard network against DOS attacks that are using already established links. If the link is already established, Veriflow will not receive this rule to evaluate the network traffic and hence will not detect the attack.

E. Detection of Firewall Bypass

Firewall is an important network security system that monitors traffic entering and leaving the system. Veriflow hold the network view using the Equivalence classes and as a result we can detect if a new rule installed tries to ByPass the Firewall. We implement this by checking the next hop of the packet, alternatively we can check the *'visited'* nodes in Veriflow and detect the same. We drop the rule and corresponding packet and alert the system for the packets trying to Bypass Firewall.

We use the following VeriFlow API's to detect this scenario:

i) *GetAffectedEquivalenceClasses*: This function returns the equivalence classes affected by a rule and returns them .

ii) *GetForwardingGraph*: This function generates the forwarding graphs for equivalence class specified in the input.

iii) *ProcessCurrentHop*: This function helps in traversing the flow graph. We use this function to determine the next hop for a location.

We use the following algorithm to detect this attack:

1. *PassesFirewall = false;*
2. */* Get equivalence classes affected using the rule */*
   EqClass = GetAffectedEquivalenceClasses()
3. */* Get forwarding graph using equivalence classes obtained in Step 1. */*
   gSet = GetForwardingGraph(EqClass)
4. *for each g in gSet:*
   a. *while rule.location is not empty*
      i. *nextHop = processCurrentHop(rule.location)*

ii. *If nextHop = NULL*
    1. *break;*
iii. *If nextHop is configured firewall:*
    1. *PassesFirewall = true;*
    2. *break;*
iv. *Else*
    1. *nextHop = processCurrentHop(*

nextHop.location)

5. *If PassesFirewall == false:*
    a. *Report as a bypass attack.*
    b. *Remove Rule from VeriFlow* *Pseudocode 1 - Firewall ByPass Attacks*

The above pseudocode traverses the hops from every source to a destination as specified in the rule. This adds a lot of complexity as some intermediate hops traversed for examination will be common between a source and destination pair. We do not need to explicitly add rule in the veriflow database as *getAffectedEquivalenceClasses* does that internally but we need to remove the faulty rule as stated in Step 5(b).

The above stated test can be enhanced to filter the sources according to a policy where source and destination is specified and VeriFlow only filters if the source and destination are blacklisted as per the policy. This policy can be implemented using two data structures:

1. By maintaining key value pair of source and destination IP.
2. By maintaining two arrays of source and destination IP's.

We give priority to the first but also maintain two data structures to maintain arrays of source and destination IP's:

unordered_map<string,string> blacklistmap; String src_ip[MAX_LEN];
String dest_ip [MAX_LEN];

To accommodate these data structures, the condition 4 in Pseudocode 1 changes as the following:

4. If (blacklistmap.get(rule.location) = rule.dest_ip) or if (rule.location exists in src_ip) or if(rule.dest_ip exists in dest_ip):
    a. *For each g in gSet:*
        i. *While loop as stated in step 4(a) in Pseudocode.*

The above modification to step 4 lets the admin maintain a list of source-destination pairs, individual sources or individual destinations to which firewall should be an en route hop. This lets us decrease the complexity involved in Pseudocode 1 and also opens up discussion of ways in which policies can be aggregated with VeriFlow. One such solution is integration of NetPlumber and VeriFlow for policy management.

IV. EVALUATION AND RESULTS

The experiments are conducted on an Microsoft azure instance running 16.02 version of Ubuntu. Mininet latest version is used and it is equipped with automatic detection of POX controller. VeriFlow has been compiled with x86_64 architecture as a binary.

VeriFlow is started first using the CLI command "./VeriFlow <port_number> <controller_ip>

<controller_port> <topology_file> as it has to process the openflow messages as soon as the controller starts. Next, pox controller is started with the python script "pox.py" with log.level of DEBUG and a custom controller to relay the openflow messages using sockets to the port on which VeriFlow is running. Once the controller is setup, mininet is started using "sudo mn" command with our custom topology explained in section III (B). Upon instantiation, mininet sets up the virtual topology and connects to the pox controller.

We execute "pingall" command to generate traffic from all hosts to all other hosts. This generates ICMP and ARP messages so that the hosts can identify the mac addresses in the network. Our setup was able to detect following attacks:

A. *Spoofed MAC Address in the ARP REPLY packet using packet_in->data.*

In the following requests, the replies are received with different MAC Addresses from the same IP:

[ARP REQUEST hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:00 10.0.0.5>10.0.0.4]

[ARP REPLY hw:1 p:2048 00:00:00:00:00:10>00:00:00:00:00:02 10.0.0.4>10.0.0.5]

[ARP REPLY hw:1 p:2048 00:00:00:00:00:01>00:00:00:00:00:02 10.0.0.100>10.0.0.5]

In this case, VeriFlow flags the following error:

Mac Address different from first response!

B. *Spoofed IP Address in the ARP REPLY packet using packet_in->data.*

In the following requests, the replies are received from IP's different than those in the ARP Request messages.

[ARP REQUEST hw:1 p:2048

00:00:00:00:00:02>00:00:00:00:00:00
10.0.0.5>10.0.0.4]

[ARP REPLY hw:1 p:2048 00:00:00:00:00:01>00:00:00:00:00:02 10.0.0.100>10.0.0.5]

In this case, VeriFlow flagged the following error:

ARP Spoofing detected! No entry for src!

C. *Duplicate ARP Replies.*

In the following messages, more than one replies are received for one request message. This is also an indication of the ARP Spoofing attack: *[ARP REQUEST hw:1 p:2048 00:00:00:00:00:01>00:00:00:00:00:00 10.0.0.4>10.0.0.5]*

[ARP REPLY hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:01 10.0.0.5>10.0.0.4]

[ARP REPLY hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:01 10.0.0.5>10.0.0.4]

In this case, modified VeriFlow flagged the following error:

ARP Spoofing detected! More response than request!

From the above cases, we can conclude that when VeriFlow is equipped with the capability to intercept every message to and fro the controller, it can easily act as a central point to detect anomalies.

We also conducted experiments to detect fragmentation firewall bypass (filtering based) and DDOS attacks. To achieve this, we have written a stripped down version of a controller using POX libraries and redirected the *packet_in* messages directed towards the controller to VeriFlow process.

D. Fragmentation Attacks

Packets that exceed the MTU size need to be fragmented at the source and transported along the data link medium and reassembly of the fragments happens at the destination. In order to reassemble the packets successfully, all fragments should share the same fragmentation ID, must have set the more flag in the IP header depending on whether is the last segment or not, contain information about length of the data carried in the fragment.

Attacks on such fragments can be classified into various categories as stated in Section III (C): *Ping Of Death/Large Fragment* (utilises a ping utility to create an IP packet with length more than the maximum allowed size of 65535 bytes), *Tiny Fragment Attack* (Uses small fragments to push some of the necessary TCP header information to the next segment to bypass certain filtering), *Teardrop Attack* (a UDP attack that uses overlapping offset fields to bring down a host), *Overlapping Fragment Attack* (it overwrites TCP header information of first fragment with valid data to pass through a firewall and overwrites subsequent fragments with malicious data), *Unnamed Attack* (Skip certain fragments by manipulating header offset to prevent reassembly).

In our experiments we target Ping Of Death/Large Fragment attacks and Tiny Fragment attacks. One common preventive measure to above stated attacks is calculating the *length of the fragment* and examining the *more fragment flag* as per the header passed in the packet_in messages. Following logs show the results obtained on running our experiments:

[IP+ICMP 10.0.0.4> (cs:f7a4 v:4 hl:5 l:18 t:64 m:0)]00:00:00 10.0.0.4>10.0.0.5]
[IP+ICMP 10.0.0.4> (cs:f7b4 v:4 hl:5 l:18 t:64 m:0)]00:00:00 10.0.0.4>10.0.0.5]
TINY FRAGMENT ATTACK: PACKET LESS THAN MIN LEN

[IP+ICMP 10.0.0.4> (cs:f7a4 v:4 hl:5 l:18 t:64 m:1)]00:00:00 10.0.0.4>10.0.0.5]
[IP+ICMP 10.0.0.4> (cs:f7b4 v:4 hl:5 l:18 t:64 m:1)]00:00:00 10.0.0.4>10.0.0.5]
TINY FIRST FRAGMENT ATTACK: PACKET LESS THAN MIN LEN

[IP+ICMP 10.0.0.4>10.0.0.5 OPTIONAL data which is
too large to be handled (cs:f7a4 v:4 hl:5 l:65340 t:64 m:1)]
[IP+ICMP 10.0.0.4>10.0.0.5 OPTIONAL data which is
too large to be handled (cs:f7a4 v:4 hl:5 l:65340 t:64 m:1)]
OOPS ! FRAGMENT WAS TOO LARGE ! DONT CRASH ME !!

E. DDOS Attacks

As part of this project, we equip VeriFlow with a new parameter of moving average. We keep an account of the number of new packets received per minute. We also maintain a threshold which states the maximum allowed difference between the number of packets recorded per minute.

In order to carry out this experiment, we flood controller with multiple pingall commands from mininet controller for new destinations that trigger a *packet_in* message to controller. We configured a maximum

allowed difference between the number of packets/second as 50. As we flood the controller, we successfully observe the following message as part of result and perform random sampling to drop packets instead of allowing them to traverse up the network stack and exploit compute resources:

[ARP REPLY hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:01 10.0.0.5>10.0.0.4]]
[ARP REPLY hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:01
10.0.0.100>10.0.0.4]f7a4 v:4 hl:5 l:65001 t:64 m:1)]
[ARP REPLY hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:01 10.0.0.5>10.0.0.4]]
§[ARP REPLY hw:1 p:2048 00:00:00:00:00:02>00:00:00:00:00:01 10.0.0.5>10.0.0.4]]
SUDDEN BURST OF REQUESTS !!! COULD BE DDOS ?? I AM DROPPING RANDOM PACKETS

Also, note that in the above results some of the header information is clubbed together when we send a large number of packet_in messages to the pox controller which unfolds a probable weakness in the way POX libraries buffer the messages in the input queue.

F. Firewall Bypass Attacks

The authors of VeriFlow[1] in the original paper mention about a use case of the libraries provided by VeriFlow to detect the packets that try to bypass the configured firewall. We extended the VeriFlow code to achieve this and test on the BGP rules as per the data set provided by the original authors of VeriFlow. To carry out this experiment we do not run VeriFlow with the Controller. Instead, since VeriFlow has the capability to parse the rules from a text file, we take advantage of this fact and run VeriFlow as an independent process without a controller attached.

As and when a rule is sent through VeriFlow libraries for verification, we calculate the *nextHop* in a recursive manner for every location that is visited in the forwarding graph generated by adding the new rule in VeriFlow database. This is done by first obtaining the equivalence classes generated by the new rule, using the generated classes to get the new forwarding graphs and then traversing each hop using processCurrentHop function provided by the veriflow libraries to find the nextHop of every location; if none of the *nextHop's* is the firewall, it is reported as a bypass attack. On running this logic of using *processCurrentHop*, we were able to detect rules which will lead to packets by passing a firewall. We configured 208.51.134.246 as the IP for firewall and detect bypassing on BGP Rules:

A RULE FILTERED TO MAKE PACKETS SKIP THE
CONFIGURED FIREWALL: src_ip 10.0.4.118 dest_ip 93.190.10.0 (255.255.255.0)
A RULE FILTERED TO MAKE PACKETS SKIP THE
CONFIGURED FIREWALL: src_ip 10.0.0.110 dest_ip
10.0.5.116 (255.255.255.252)
A RULE FILTERED TO MAKE PACKETS SKIP THE
CONFIGURED FIREWALL: src_ip 10.0.4.118 dest_ip
69.194.0.0 (255.255.128.0)

Above are some of the results from our experiments to report a firewall bypass and also mention source and destination IP with network mask.

V. LEARNINGS

With time, network management has become an increasingly complicated and difficult process. Dynamics

involved in establishing and maintaining a network are increasing and so is the need for new solutions to maintain security. Having mentioned that, there have been many learnings during the course of the project.

First, study of architecture and implementation of VeriFlow demonstrating the class of analysis that can be done using simple rule data. It is a very novel illustration of designing solutions using flow rules and classifying authentic traffic from attack traffic.

Second, VeriFlow keeps the infrastructure flexible that lead to a new scope for solutions that can be build on the existing infrastructure. It provides a perfect example of how an infrastructure destined towards providing solutions for cyber security should be designed.

Third, designing a controller using POX libraries helps network engineers to develop scenario based controllers with dynamic features which can be activated on need basis (for instance, if a traffic flow which is not in accordance with the policy specified say using NetPlumber is detected, VeriFlow can be dynamically enabled to examine the rules).

Fourth, VeriFlow and POX provide API's that can interface with multiple languages, Python, C++ etc. This helps in collaborating solutions which have the desired features but are written in completely different languages.

Also, during the experiments, we learnt how to use network simulators like Mininet to test the network topology, design controllers, the semantics and parameters associated with a flow rule and their usage, role and placement of various network elements involved in SDN.

VI. CONCLUSION

Veriflow is an exceptional platform to build on because of its capabilities in detecting anomalies in real time. As we demonstrated in this experiment, there can be a 'one solution' approach to enhance Veriflow to have abilities to detect malicious attacks alongside with the abilities to check for the new rules in the system. Following the Veriflow detection, we can set up a system that can further analyse the packets. This helps in having a single system that detects any anomaly in the network and ensures that the logic to detect anomalies is not scattered.

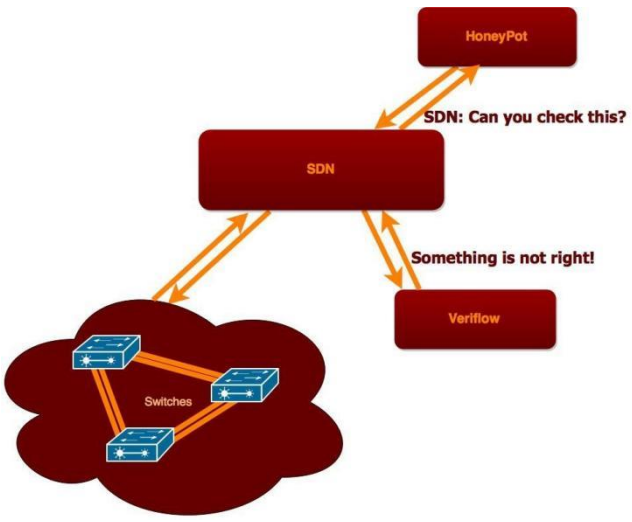

Figure 5: Flow of Actions in SDN

Apart from providing a single point to check the activities that are an anomaly, Veriflow provides an

excellent platform for detecting attacks like ARP spoofing by making changes to the logic used to parse packet related information. Other attacks like accessing fragmentation attacks like tiny Fragment, tiny first fragment and large fragment, firewall bypass and network denial of service attacks can also be achieved using Veriflow.

However, it is important to note that there is a slight delay introduced in Veriflow because of the processing of packets (in addition to Rules) and the capabilities of Veriflow is limited as we saw in the case of Network DOS attack. This is due to the fact that we miss a whole lot of information of what is going on in the network when an existing connection is used. However, capability of Veriflow in checking the malicious activities using the Equivalence classes provides us with a solid groundwork which can be further extended to deter other kinds of malicious attacks like detection of Fake Topology.

VII. FUTURE WORK

Our work focuses on providing a dimension to placement of VeriFlow in the SDN Network and illustrate some of the scenarios where VeriFlow can be used to detect attacks if it is equipped with packet header information along with the rules data. The tests are very basic and can handle much more complication scenarios for example detecting DDOS attack per flow and detect firewall bypass based on dynamic policies. Detection of Fake topology using Veriflow would also be an interesting area that is yet to be explored.

Security in SDN has been a trending topic for quite a while now. Although, controllers possessing a new set of features have been introduced there is not a set yes or no answer to ascertain the security of SDN stack. This keeps a room open for new innovation in security stack for SDN. Open Network Foundation, the pioneer for OpenFlow Protocol has been continuously publishing papers unveiling new security flaws.

There have been many solutions like Floodlight, NetPlumber, HoneyPot, FlowVisor which have been a result of integration of various ideas and technologies that lead to a strengthened security space in SDN. There still exist weak links in protecting and securing the controller, creating a robust policy framework, conducting forensics and remediation and maintaining trust for the elements interacting in the network.

On the flip side, the primary goals of these solutions are expected to be cost effective, simple and secure. A new class of experiments termed as SDSec (Software Defined Security) has been emerging that aims to provide a dynamic distributed system to virtualise the network security as a single logical system build on top of multiple physical systems. IT belongs to the class of NFV(Network Function Virtualization) solutions which decouple a network's features such as designing a firewall and IDS from the proprietary hardware so build a logical software.

VIII. REFERENCES

[1] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," *Proc. 10th USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2013.

[2] V. Mancuso, "Address Resolution Protocol (ARP)," unpublished, [Online]. Available: https://www-sop.inria.fr/members/Vincenzo.Mancuso/ReteInternet/09_arp.pdf

[3] Open Networking Foundation, *OpenFlow Switch Specification Version 1.3.0*, 2012. [Online]. Available: https://opennetworking.org

[4] Mininet, "Mininet: An Instant Virtual Network on your Laptop (or other PC)," [Online]. Available: http://mininet.org

[5] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox, "Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks," arXiv preprint arXiv:1403.8008, 2014. [Online].

Available: https://doi.org/10.48550/arXiv.1403.8008

[6] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting Security Attacks in Software-Defined Networks," *Proc. NDSS*, 2015.

[7] C. Fung, L. Chen, and B. McCormick, "FlowMon: Detecting malicious switches in software-defined networks," unpublished, 2015. [Online]. Available: https://www.researchgate.net/publication/282613951

[8] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks," *Proc. 2015 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, SUCCESS Lab, Texas A&M.

[9] A. M. AbdelSalam, A. B. El-Sisi and R. K. Vamshi, "Mitigating ARP spoofing attacks in software-defined networks", Proc. 25th Int. Conf. Comput. Theory Appl. (ICCTA), pp. 126-131, Oct. 2015.

[10] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, Mar./Apr. 2001.

[11] IEEE 802 Numbers, Internet Assigned Numbers Authority (IANA). [Online]. Available: https://www.iana.org

[12] SDX Central. [Online]. Available: https://www.sdxcentral.com

[13] D. C. Plummer, "An Ethernet Address Resolution Protocol," RFC 826, Nov. 1982. [Online]. Available: https://tools.ietf.org/html/rfc826

[14] G. Ziemba, D. Reed, and P. Traina, "Security Considerations for IP Fragment Filtering," RFC 1858, Oct. 1995. [Online]. Available: https://tools.ietf.org/html/rfc1858

[15] J. Postel, "Internet Protocol: DARPA Internet Program Protocol Specification," RFC 791, Sept. 1981. [Online]. Available: https://tools.ietf.org/html/rfc791