A Self-Defense Mechanism against Volumetric Denial of Service Attacks in OpenFlow Networks

Markus Jungbluth*, Julian Wiegel†
School of Engineering,
Saarland University of Applied Sciences (htw saar)
Email: *kim.markus.jungbluth@htwsaar.de, †kim.julian.wiegel@htwsaar.de

I. Introduction

When the predecessor of our today's Internet, the ARPANET, was taken into operation in 1969 nobody anticipated the misuse of its protocols and mechanisms which are present today.

Recently many networks, organizations and companies started struggling with volumetric Denial of Service attacks. These are carried out against the network's infrastructure rather than individual services and aim at exhausting bandwidth resources in order to render a service or an entire network inaccessible to legitimate users. One should not confuse these kind of attacks with transport or application layer attacks, which aim at depleting resources on a host through vulnerabilities in software.

Volumetric attacks on the infrastructure of the Internet are possible due to several design issues in its current layout. Until today, classical networks do not offer a feature to control ingress traffic to a host or a network. Every packet which is sent out by one of the hosts on the public Internet is delivered to its destination network. The receiving host or network cannot do anything against the delivery of the packet.

In order to address these shortcomings different techniques have been established by the Internet community over time. In 2009 McPherson et. al defined *Remote Triggered Black Hole Filtering* as a new standard [1], which had already been used informally by many Internet Service Providers (ISPs) and carriers. The technique allows to selectively block ingress traffic for a specific IP address using the Border Gateway Protocol (BGP). However the filtering mechanism can only be performed based on the destination address of the malicious traffic aggregate. As of today this mechanism is supported by a majority of carriers and providers and is one of the few globally available tools to fight volumetric attacks. However this technique comes at a cost. While it allows to protect a network from the malicious traffic aggregate, it also disconnects the attacked IP address from the public Internet. Thus ultimately achieving, what the attacker tried to accomplish.

To overcome these issues, a team of engineers from network equipment manufacturers proposed a new standard, which allows for a more granular filtering. [2] They proposed an extension to the BGP, which allows to propagate a set of firewall rules, so-called *Flow Specifications*, within and across Autonomous Systems (ASs).

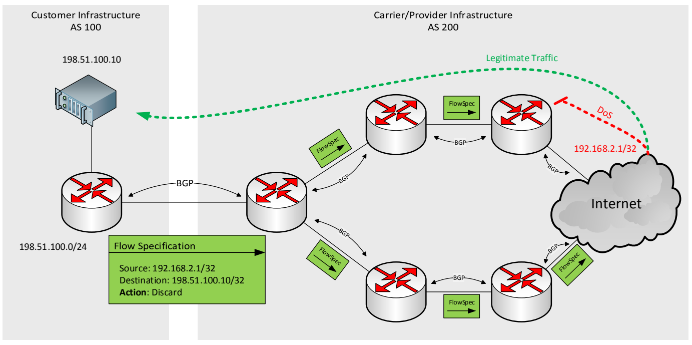

Figure 1. Sample Topology illustrating the dissemination of Flow Specifications over the BGP

Figure 1 illustrates the dissemination process. In the shown topology a host (198.51.100.10) within AS 100 is attacked with a volumetric Denial of Service attack, originating from a host with the IP Address (192.168.2.1). Using Flow Specifications, the network and the host can be protected against the attack. For this, a Flow Specification is announced over the BGP peering between AS 100 and 200, containing information about the traffic agregate to be blocked.

The routers apply the received rules to their firewall tables and subsequently block the transiting malicious traffic aggregate before it can reach its destination. Contrary to the Remote Triggered Black Hole (RTBH) the rules may be built using the source address of the malicious traffic aggregate, thus allowing to mitigate the attack without affecting the reachability of the attacked addresses for legitimate users.

BGP Flow Specifications are a promising feature when it comes to mitigation of volumetric attacks, as they eliminate the existing drawbacks with RTBH Filtering and allow dynamic traffic ingress control without affecting host reachability.

A. *Purpose of our Work*

For our work we have looked at similar defense mechanisms in OpenFlow Networks. When comparing the available match criteria in Flow Specifications to the match fields available in OpenFlow 1.3, we can see a lot of similarities. As shown in Table I almost all criteria found in the Flow Specifications may be modeled using the Match Fields of OpenFlow 1.3.

Within this document we propose a mechanism within an OpenFlow network, which allows individual hosts to request

Table I
COMPARISON OF MATCH-CRITERIA AVAILABLE IN BGP FLOW SPECIFICATIONS AND OPENFLOW

BGP Flow Spec.	Open Flow Match Field (1.3)
Destination Prefix	OFPXMT_OFB_IPV4_DST (also IPv6)
Source Prefix	OFPXMT_OFB_IPV4_SRC (also IPv6)
IP Protocol	OFPXMT_OFB_IP_PROTO
Port	Available through combinations of match criteria
Destination Port	OFPXMT_OFB_UDP_DST (also TCP, SCTP)
Source Port	OFPXMT_OFB_UDP_SRC (also TCP, SCTP)
ICMP type	OFPXMT_OFB_ICMPV4_TYPE
ICMP code	OFPXMT_OFB_ICMPV4_CODE
TCP flags	**Not supported (added in 1.5)**
Packet Length	**Not supported**
DSCP	OFPXMT_OFB_IP_DSCP
Fragment	**Not supported**

filter rules, similar to the ones found in BGP Flow Specifications, to be installed on OpenFlow Switches to protect themselves against volumetric Denial of Service attacks.

Our work comprises the following outcomes:

1) We defined a mechanism for announcing a self-defense filter request within a Local Area Network (LAN)
2) Further, we specified a communication protocol with which the filter request can be transmitted to an OpenFlow Controller
3) We developed a reference implementation and verified its functionality through a series of tests

II. SELF-DEFENSE ARCHITECTURE IN OPENFLOW

We are proposing a mechanism, which allows hosts within a LAN to request filter rules on their network infrastructure to protect themselves from Denial of Service attacks. For this we had to develop a mechanism that allows any host on the network to easily request such a filter. In the following we will describe the Request Mechanisms and associated communication protocol.

A. Request Mechanism

In order for a host to be able to request a filter, it must notify the network of the imminent attack. We have decided to implement a mechanism in which the host sends a UDP datagram containing information about the attack to its default Gateway. We have chosen the default Gateway, as it presents the *next hop*, through which the traffic is being forwarded and is usually known to the host. Further, by using the gateway address as an end-point for the request mechanism, we do not *waste* an additional address for the mitigation service.

To enable our service to process the UDP packets emitted by the host, the OpenFlow network with its associated switches must be properly initialized. As we can take from the sequence diagram shown in Figure 2, upon initialization of the OpenFlow Session, the OpenFlow controller injects a Flow Table Entry into all switches. The entry is constructed in such a fashion, that it forwards all packets destined to the standard gateway with a UDP destination port equal to 5653 to the OpenFlow controller.

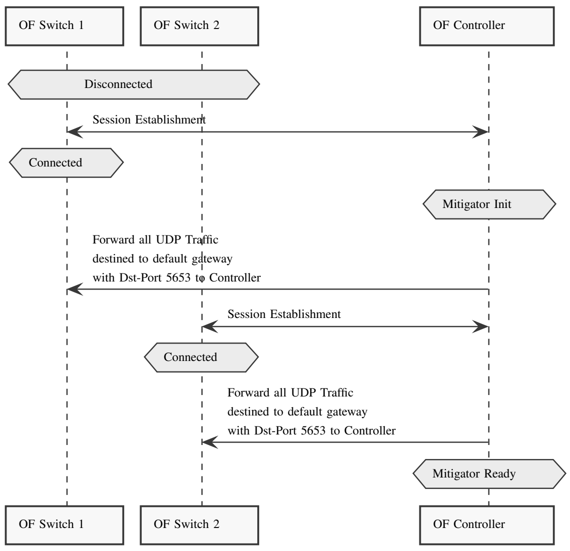

Figure 2. Initialization of our Mitigation Component

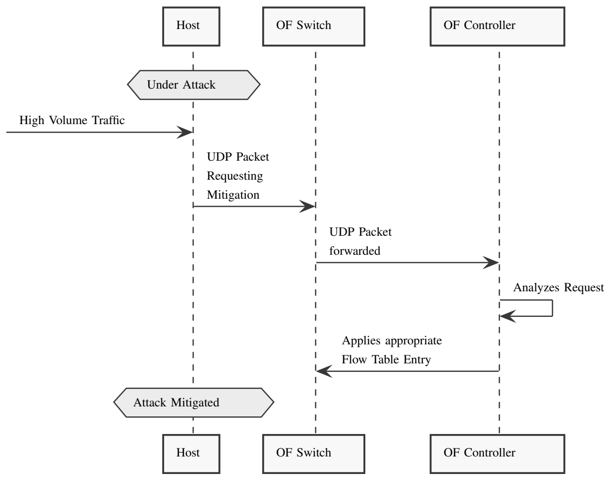

Figure 3. Mitigation process

B. Protocol Definition and Structure

As a client needs a way of serializing the information about the attack, we have designed a protocol with which we can specify the parameters of the malicious traffic aggregate to be blocked by the network. Our protocol is ASCII-Based, with a syntactical structure very similar to the one found in the Hypertext Transfer Protocol (HTTP). We have chosen this format to allow for an easy implementation and debugging of the messages. An example for a syntactically correct request is shown below:

```
METHODNAME Arg1 Arg2 Arg3
Param1: Value1
Param2: Value2
Param3: Value3
```

Listing 1. Sample for syntactically correct request

A protocol message consists of a mandatory method name, followed by an optional series of method arguments and a

Table II
AVAILABLE PARAMETERS IN OUR PROTOCOL

Parameter Name	Value
Protocol	`icmp	tcp	udp`
Source-Port	`[1-65535]`
Destination-Port	`[1-65535]`

list of parameters and their corresponding values. The list of parameters and each parameter itself are separated by a newline character from the method name. The end of a protocol message is indicated by two newline characters.

As POX, which serves as a framework for our implementation is based on OpenFlow 1.0, we can only discard malicious traffic and are unable to rate-limit, as the standard lacks this feature. Therefore the only method name currently used, is `DISCARD`.

In addition to specifying a host address which is to be blocked, an attacked host may also specify a more detailed description of the traffic aggregate using parameters. Currently our protocol supports three parameters, as shown in Table II. A sample of a request using all available parameters is shown below:

```
DISCARD 11.0.0.4
Protocol: udp
Source-Port: 47234
Destination-Port: 53
```

Listing 2. Sample for a valid request

C. Reference Implementation

In order to demonstrate our proposed mechanism, we have designed a reference implementation of the self-defense mechanism. We based our software on POX, an open source OpenFlow controller developed by the Open Networking Lab at Stanford University. [3] The POX software is highly modularized and allows to add new functionalities quite easily. We designed our software as an additional `mitigator` module, which can be used alongside the standard forwarding modules.

The module installs the Flow Entries on each switch, that connects to the controller and processes the UDP packets containing mitigation requests from the clients.

Internally we are using two classes to process incoming requests. The `Decode` class analyzes the syntax of a received message and de-serializes the contained data. Once the message has been parsed, it is passed on to the `Eval` class which interprets the messages and prepares the data to be applied as a mitigating Flow Entry.

Further, we also designed a client application to enable individual hosts on a network to make mitigation requests. The client was built as a simple command line application and parses a set of arguments from which the protocol messages are built.

```
usage: mitigator_client.py [-h] [--protocol
    ↪ {tcp,udp,icmp}] [--src_port SRC_PORT]
    ↪ [--dst_port DST_PORT] src_ip

OpenFlow DDoS Mitigator Client

positional arguments:
  src_ip                IP from which the attack
      ↪ originates.

optional arguments:
  -h, --help            show this help message and
      ↪ exit
  --protocol {tcp,udp,icmp}
                        A Layer 3 protocol which
                            ↪ matches the traffic
                            ↪ aggregate
                        to be blocked.
  --src_port SRC_PORT   A source port which matches
      ↪ in the traffic aggregate
                        to be blocked. (You must
                            ↪ specify a protocol
                            ↪ other than
                        icmp)
  --dst_port DST_PORT   A destination port which
      ↪ matches in the traffic
                        aggregate to be blocked.
                            ↪ (You must specify a
                            ↪ protocol
                        other than icmp)
```

Listing 3. Usage of our client component

Detailed information about the usage of both the server and client component of our application can be found in Section III.

III. TESTING

In order to find out, whether our Self-Defense Request Architecture works as desired, we needed to setup a proper test environment. For conducting our tests, we have started using mininet with OpenvSwitches and our custom topology, later on we intended to port our setup to Zodiac FX OpenFlow-Switches by Northbound Networks and eventually to a HP Aruba 2920-24G Switch.

A. Test Environment

With Mininet [4], a virtual networking environment that can run customized kernel, switching and application code, we have been able to build a testbed for our self-defense mechanism very easily. It comes with a handy CLI, giving us control over every node and switch in our topology. Although it is possible to use pre-defined network topologies, we decided to create our own with the Mininet Python API.

Topology

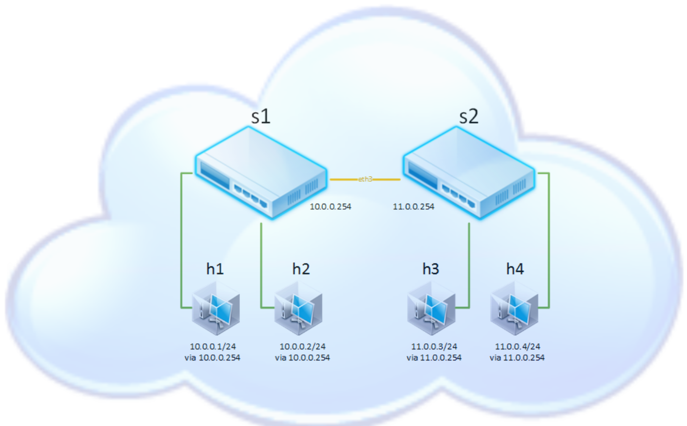

Figure 4. Mininet topology

Figure 4 shows our custom topology, using 4 hosts (H1-H4) and 2 switches (S1,S2). Host 1 and host 2 are connected to switch 1, host 3 and 4 to switch 2, and a link connecting S1 and S2. This topology should represent two AS in different locations as seen in Figure 1. H1 and H2 use a `10.0.0.0/24` subnet with a default gateway set to `10.0.0.254`, while H3 and H4 use a `11.0.0.0/24` subnet, with the default gateway pointing to `11.0.0.254`. As mentioned earlier in the text II-A, the switch interconnection (`eth3` on S1 and S2) gets the default gateway address of each AS. This scenario is not possible in classic IP Networks, as they would require a dedicated transit network.

```
class SdnProjectTopo(Topo):
 def build(self):
  s1 = self.addSwitch('s1')
  s2 = self.addSwitch('s2')
  h1 = self.addHost('h1', ip='10.0.0.1/24',
      ↪ defaultRoute='via 10.0.0.254')
  h2 = self.addHost('h2', ip='10.0.0.2/24',
      ↪ defaultRoute='via 10.0.0.254')
  h3 = self.addHost('h3', ip='11.0.0.3/24',
      ↪ defaultRoute='via 11.0.0.254')
  h4 = self.addHost('h4', ip='11.0.0.4/24',
      ↪ defaultRoute='via 11.0.0.254')
  self.addLink(h1, s1)
  self.addLink(h2, s1)
  self.addLink(h3, s2)
  self.addLink(h4, s2)
  self.addLink(s1, s2)
```

Listing 4. Mininet topology python configuration

Listing 4 shows a snippet of our python code to create the topology, mentioned above, using the Mininet API.

B. Simulating an Attack and its Mitigation

We did not want to reinvent the wheel when it came to packet switching and forwarding, so we searched for an existing solution and found a proper pox module in the forwarding examples folder. [5] The **forwarding.l3_learning** module allows our OpenFlow switches to behave like "normal" layer 2 learning switches, with a slight difference: instead of learning MAC-addresses, the layer 3 learning switch learns IP-addresses and forwards the packets to the specific port without taking care of subnetting/CIDR. This would work fine in a "clean-slate" world, where all internet traffic is handled only by OpenFlow and proper networking stack implementation on host machines. In reality, hosts usually do care about subnets and want to know a gateway to communicate with hosts on a different network. To face this issue, you can specify fake gateways on the command line to satisfy the hosts. This is achieved by listening for ARP requests, and responding to them with the MAC address of the *fake gateway*.

Starting POX Controller and Mininet

```
./pox.py forwarding.l3_learning
    ↪ --fakeways=10.0.0.254,11.0.0.254
    ↪ samples.mitigator
    ↪ --gateways=10.0.0.254,11.0.0.254
...
INFO:samples.mitigator:Installed Flow to
    ↪ redirect all traffic to 10.0.0.254 Port
    ↪ 5653 to the controller
INFO:samples.mitigator:Installed Flow to
    ↪ redirect all traffic to 11.0.0.254 Port
    ↪ 5653 to the controller
INFO:openflow.of_01:[00-00-00-00-00-01 3]
    ↪ connected
INFO:samples.mitigator:Installed Flow to
    ↪ redirect all traffic to 10.0.254 Port
    ↪ 5653 to the controller
INFO:samples.mitigator:Installed Flow to
    ↪ redirect all traffic to 11.0.254 Port
    ↪ 5653 to the controller
```

Listing 5. Command for starting the POX controller

Listing 5 shows the command to start the POX controller with the forwarding.l3_learning module (including the fake gateways) and our mitigator service module, which also takes the gateways as an argument.

```
sudo mn --custom sdn-project.py --topo
    ↪ sdnprojecttopo --controller remote
```

Listing 6. Starting Mininet

This command starts the mininet virtual environment with our custom topology using a remote OpenFlow controller.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

Listing 7. Command for testing connectivity between all hosts in our test environment

Figure 7 shows the *pingall* command to test reachability of all hosts on both subnets.

Simulating an Attack

To simulate attacks, we had to generate just enough traffic to overload the network connection to the client. It should have been still possible for it to send the mitigation request to the gateway anyway. For this purpose, we have used the `hping3` [6] command line utility. Using this tool, it is possible to generate any TCP/IP packet stream, including ICMP and UDP datagrams, at adaptable rates to stress the client.

```
hping3 --udp -s 4711 --keep -q -d 500
    ↪ --faster 10.0.0.1
```

Listing 8. Command used to simulate a volumetric attack

Listing 8 shows the command we were using to start an attack on H1. In this scenario, we are sending UDP datagrams with a payload of 500 Byte on the static source port 4711 at a packet rate of about 100,000 packets/sec. This has shown to be the *sweet spot*, as we were testing different datagram sizes at different packet rates, mostly crashing our POX Controller. While launching the attack, we were generating some background noise by pinging H1 from H3, in order to make the effect of the mitigation visible.

```
python mitigator_client.py 11.0.0.4
Request sent
```

Listing 9. Command used to initiate self-defense with our mitigator client

The mitigator client (Listing 9) is started with the IP address of the attacker as an argument, requesting an OpenFlow entry that blocks the entire traffic from the attacker to the victim.

```
INFO:samples.mitigator:UDP Packet to DstPort
    ↪ 5653 seen
INFO:samples.mitigator:Payload: DISCARD
    ↪ 11.0.0.4
INFO:samples.mitigator:Mitigation triggered
INFO:samples.mitigator:Host 11.0.0.4 blocked
    ↪ for 120 sec (idle) 240 sec (hard)
    ↪ {'src_port': None, 'protocol': None,
    ↪ 'dst_port': None}
INFO:samples.mitigator:Host 11.0.0.4 blocked
    ↪ for 120 sec (idle) 240 sec (hard)
    ↪ {'src_port': None, 'protocol': None,
    ↪ 'dst_port': None}
```

Listing 10. Log messages shown in POX, when the client requested a mitigation of the attack

Listing 10 shows the debugging messages produced by our mitigation service when a new request on UDP Port 5653 was received. The extracted Payload of the incoming UDP datagram shows a `DISCARD` message, triggering the mitigation process. Finally, the OpenFlow entry is sent out to the two switches, blocking all the attack traffic from `11.0.0.4` for at least 120 seconds.

IV. Results and Future Work

In our work we have shown how the capabilities of an OpenFlow network can be used to create a mechanism to protect hosts from volumetric Denial of Service attacks by giving them control over ingress traffic. We have been able to implement a mitigator service, cutting all traffic from the attacker to the victim directly on the access device, stopping single targeted, malicious traffic. Our mitigation service has been successfully tested in the virtual Mininet environment.

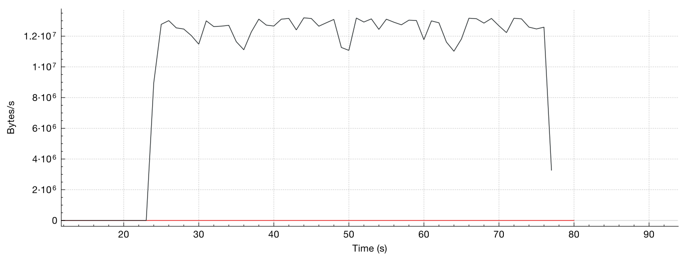

Figure 5. Bandwidth usage while attacking

Figure 5 shows the bandwidth consumed by the *hping3* UDP datagram attack (black graph), alongside with normal ICMP echo request/response noise (red graph), recorded by the IO graphing tool in *Wireshark*. While the attack was peaking at over 12 MB/s, it was immediately interrupted after 54 seconds when our mitigation client submitted its *DISCARD* request to the mitigation service, installing a flow entry dropping all traffic from H4 to H1.

A. Porting to Zodiac FX and HPE 2920 OpenFlow switches

We have tried to port our setup and services to Zodiac FX OpenFlow switches [7], but did not succeed establishing an inter-switch connection, failing physical, Windows XP hosts to reach the two other Windows XP hosts on the second Zodiac switch. We think that this has to be a compatibility problem of the forwarding.l3_switching module in POX with the Zodiac FX hardware as it does not recognize the physical inter-switch connection. The mininet switches are using basically the same switching fabric, only seperated by grouped, bridged interfaces. Keeping in mind that most of the POX modules were written some years ago, mininet being the only testbed available at that time, the developers were maybe not aware of (physical) multi-switch topologies.

Another testbed we tried was an OpenFlow enabled, 24-port HPE Aruba 2920 gigabit switch, which complies with *OpenFlow Switch Specification v1.0.0 (December 2009)* and since the firmware WB.15.14 also *v1.3.1 (September 2012)*. According to [8], seperation of OpenFlow and non-OpenFlow traffic is done on a per-VLAN base. It is possible to configure an OpenFlow instance in two, mutually exclusive modes:

1) Virtualization mode: This operating mode allows OpenFlow- and non-OpenFlow-VLANs to coexist. There can be multiple, independent instances with different OpenFlow settings and controllers.
2) Aggregation mode: This mode enables OpenFlow on all VLANs configured in the switch. There is only one OpenFlow instance and there must be a separate *Management VLAN*, and a *Controller VLAN*.

We chose the *Virtualization mode* to run our tests with, but did not succeed as the 2920 was not accepting any flow table changes.

B. Future Work

We see our solution as a Proof of Concept, to demonstrate that the capabilities of OpenFlow can be leveraged to create a mechanism very similar to the one found in the BGP. Our mechanism can be improved in many ways. For example, the remaining matching criteria available in OpenFlow 1.0 may be added to our protocol, to allow clients to filter traffic more precisely. Further, by porting our application to a controller which supports OpenFlow 1.3 or higher, we could enable clients to request a rate-limit instead of a discard action.

In order to keep both the protocol simple and straightforward, we have abstained from implementing security mechanisms, which ensure that hosts may not request mitigation actions, which affect the connectivity of other participants in the network in a negative way.

Abbreviations

AS	Autonomous System
BGP	Border Gateway Protocol
DoS	Denial of Service
HTTP	Hypertext Transfer Protocol
ISP	Internet Service Provider
RTBH	Remote Triggered Black Hole
LAN	Local Area Network

References

[1] W. Kumari and D. McPherson, “Remote Triggered Black Hole Filtering with Unicast Reverse Path Forwarding (uRPF),” RFC 5635 (Informational), Internet Engineering Task Force, Aug. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5635.txt

[2] P. Marques, N. Sheth, R. Raszuk, B. Greene, J. Mauch, and D. McPherson, “Dissemination of Flow Specification Rules,” RFC 5575 (Proposed Standard), Internet Engineering Task Force, Aug. 2009, updated by RFC 7674. [Online]. Available: http://www.ietf.org/rfc/rfc5575.txt

[3] Pox wiki. Open Networking Lab, Stanford University. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[4] Mininet: An instant virtual network on your laptop (or other pc). Mininet Team. [Online]. Available: http://www.mininet.org/

[5] Pox wiki - open networking lab - confluence. Open Networking Lab, Stanford University. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-forwarding.l3_learning

[6] S. Sanfillipo. (2009, September) hping wiki - home. [Online]. Available: http://wiki.hping.org/

[7] Northbound networks: Zodiac fx. Northbound Networks Pty. Ltd. [Online]. Available: https://northboundnetworks.com/collections/zodiac-fx

[8] H. Aruba. (2017) Hpe arubaos-switch openflow v1.3 adminitrator guide for 16.03. [Online]. Available: http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c05365339