
Article:
Birkinshaw, Celyn, Rouka, Elpida and Vasilakis, Vasileios orcid.org/0000-0003-4902-8226 (2019) Implementing an intrusion detection and prevention system using software-defined networking : defending against port-scanning and denial-of-service attacks. Journal of Network and Computer Applications. pp. 71-85. ISSN 1084-8045

https://doi.org/10.1016/j.jnca.2019.03.005

Implementing an Intrusion Detection and Prevention System Using Software-Defined Networking: Defending Against Port-Scanning and Denial-of-Service Attacks

Celyn Birkinshaw, Elpida Rouka, Vassilios G. Vassilakis*

Dept. of Computer Science, University of York, York, United Kingdom

Abstract

Over recent years, we have observed a significant increase in the number and the sophistication of cyber attacks targeting home users, businesses, government organizations and even critical infrastructure. In many cases, it is important to detect attacks at the very early stages, before significant damage can be caused to networks and protected systems, including accessing sensitive data. To this end, cybersecurity researchers and professionals are exploring the use of Software-Defined Networking (SDN) technology for efficient and real-time defense against cyberattacks. SDN enables network control to be logically centralised by decoupling the control plane from the data plane. This feature enables network programmability and has the potential to almost instantly block network traffic when some malicious activity is detected.

In this work, we design and implement an Intrusion Detection and Prevention System (IDPS) using SDN. Our IDPS is a software-application that monitors networks and systems for malicious activities or security policy violations and takes steps to mitigate such activity. We specifically focus on defending against port-scanning and Denial of Service (DoS) attacks. However, the proposed design and detection methodology has the potential to be expanded to a wide range of other malicious activities. We have implemented and tested two connection-based techniques as part of the IDPS, namely the Credit-Based Threshold Random Walk (CB-TRW) and Rate Limiting (RL). As a mechanism to defend against port-scanning, we outline and test our Port Bingo (PB) algorithm. Furthermore, we include QoS as a DoS attack mitigation, which relies on flow-statistics from a network switch. We conducted extensive experiments in a purpose-built testbed environment. The experimental results show that the launched port-scanning and DoS attacks can be detected and stopped in real-time. Finally, the rate of false positives can be kept sufficiently low by tuning the threshold parameters of the detection algorithms.

*Corresponding author
Email address: vv573@york.ac.uk (Vassilios G. Vassilakis)

Keywords: Intrusion detection and prevention system, software-defined networking, anomaly detection, denial of service, port scanning

1. Introduction

The number and the sophistication of cyber attacks has significantly increased over recent years, from the devastating impact of WannaCry and Petya to the explosion of cryptojacking [1]. Modern day cybercriminals are targeting home users, businesses, government organizations, and critical infrastructure. In many cases, even small delays in the detection and prevention of malicious activity may cause significant damage to the protected systems or may enable cybercriminals to get unauthorized access to confidential data. This necessitates early detection of cyberattacks, typically using anomaly-based detection approaches [2]. In recent years, cybersecurity researchers and professionals have taken a greater interest in exploring the use of Software-Defined Networking (SDN) [3] and network programmability for developing effective and efficient defense mechanisms against cyberattacks [4, 5, 6].

The programmable network is a concept that has recently seen a significant resurgence with much of the attention generated by SDN, which enables network control to be logically centralised by decoupling the control plane from the data plane [7]. The SDN architecture includes a centralised network controller with a global view of the network and an Application Programming Interface (API) for developing network applications. Among the advantages of SDN is the potential to almost instantly drop malicious traffic from a network interface once it has been detected.

Intrusion detection is the activity of detecting unauthorised access to computer systems or devices. It typically involves automated detection and alerts to a security system about intrusions that have happened or are happening [8, 9]. In this work, we design and implement an Intrusion Detection and Prevention System (IDPS) using SDN. Our IDPS is a defense system, continuously monitoring a network for unusual activity and malicious network traffic and implementing countermeasures against cyberattacks. Our design is based on the OpenFlow protocol, which is the most popular SDN protocol today for the communication between the controller and the network switches [10]. In particular, we have implemented and tested two connection-based techniques as part of the IDPS, namely the Credit-Based Threshold Random Walk (CB-TRW) and the Rate Limiting (RL) techniques, a port-scanning detection technique which we call Port Bingo (PB), and a QoS technique which relies on flow-statistics to mitigate against DoS attacks. Our developed solution has been tested against port-scanning and Denial of Service (DoS) attacks, which account for 4% and 10% of the total network attacks respectively, in a recent McAfee report [11], and are generally identified among the most prevalent network attacks. Moreover, the proposed IDPS design and the detection methodologies have the potential to be expanded to defend against a wide range cyberattacks, involving both anomaly-based and signature-based approaches.

The experimental part is carried out in a purpose-built Virtual Machine (VM) testbed environment that includes an SDN switch (OvS 2.7 [12]) and multiple VMs. One VM acts as the SDN controller (Python-based POX controller [13]). Other VMs are used as attackers, targets, and benign hosts. Our extensive experimental results show that the launched port-scanning and DoS attacks can be quickly detected and stopped in real time. Finally, the rate of false positives can be kept sufficiently low by tuning the threshold parameters of the detection algorithms.

The rest of the paper is organised as follows. Section 2 briefly covers the required background of SDN and the OpenFlow protocol. Section 3 reviews the related work on SDN-based defense mechanisms. Section 4 describes our IDPS design and implementation. Section 5 presents our testbed and experimental setup. Sections 6, 7, and 8 present our IDPS experimental results and analysis for port-scanning and different types of DoS attacks. Section 9 concludes the paper by summarising the contributions of our work and discussing possible future directions.

2. Background

According to Casado et al. [14], legacy computer networks are too hard to change. McKeown et al. [7] describe this as "ossification" of computer networks, which they attribute to the coupling of the control plane to the data plane, a bane for innovation cycles at both planes according to RFC 7526 [15]. On the contrary, the SDN architecture is based on the following design principles: (i) separation of the control plane and data plane; (ii) programmability of network services; and (iii) logically centralised control. In the following paragraphs, we briefly describe the core SDN components.

2.1. SDN Flows, Control Models, and Interfaces

Most applications send data in flows that are made of many individual packets. Traditionally, a switch uses the addressing information in *each* packet to make routing decisions for *each* packet. In SDN, the first packet of a flow is sent to the controller and returned to the switch with routing controls that are reused by the subsequent packets in the flow. A flow can be specified with some combination of packet header values such as the source and destination address, the protocol and the application [3]. In SDN the unit of control is a flow of packets. SDN provides either *reactive control* whereby switches consult a controller for routing information, or *proactive control* whereby switches are pre-populated with forwarding rules to reflect the network policy rules. A combination of both types of control can be used.

Interactions between a controller and a switch take place across the southbound API (Fig. 1), typically implemented using the OpenFlow protocol. Interactions between a controller and a network application, take place across the northbound API..

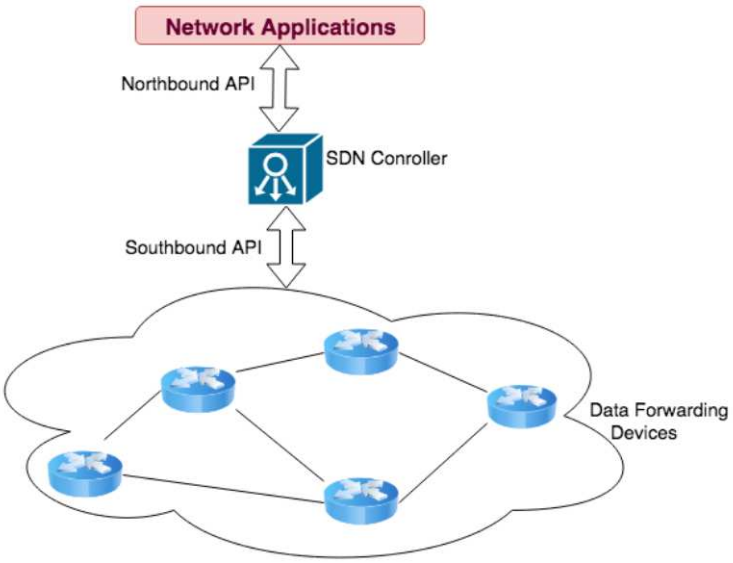

Figure 1: High-level structure of an SDN.

2.2. *SDN Controllers and Applications*

SDN controllers have access to a uniform set of switch statistics, and the ability to implement network controls and security policies. Security checks can be generated inside a controller [16] or by applications that handle events from a controller.

The northbound API from the controller, enables users to extend the functionality of a programmable network through the development of SDN applications, including network security applications. Applications are often user-built to meet some specific need. In terms of network security, applications benefit from the controller's global view of the network, which offers network-wide intrusion detection capabilities. SDN applications typically rely on data from a controller but can also use data from external sources. For example, an intrusion detection application can use the Snort IDPS [17] and its databases for signature detection.

2.3. *The OpenFlow Protocol*

OpenFlow outlines the essential components and the protocol specification for the southbound API, proposed by McKeown et al. [7]. A secure channel between controller and switch is established using Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols followed by the transfer of ''Hello" messages. The traffic between a controller and a switch includes PacketIn and PacketOut packets, which may encapsulate other types of OpenFlow packets.

PacketIn packets are sent from a switch to a controller, typically when there is no flow-entry for a packet. Every OpenFlow switch has a flow-entry called

a *table-miss* which specifies what to do if there is no matching flow-entry. A "Send to Controller" action in the table-miss means the default action of the switch is to send the packet to the controller.

PacketOut packets are sent from a controller to a switch, usually as a response to a PacketIn. To exemplify typical packet encapsulations in OpenFlow, PacketOut packets often encapsulate a FlowMod packet which contains flow-entry information required by the switch.

A flow-table contains the current flow-entries; i.e., those that have been installed and have not expired. Flow-entries are used to match packets and assign actions to them. Actions include: forward the packet to a specific port, send the packet to a controller over the secure channel, drop the packet, or flood the packet on all ports.

When a switch receives a new flow-entry, it caches the instructions so that future packets can be rapidly forwarded without further communication from the controller until the flow-entry is timed-out. Flow-entry parameters include match fields, counters, and actions. Match fields are used to identify different types of incoming packets.

On receipt of an incoming packet, the switch processes the packet for packet-matching based on the priority of the flow-entry, which is set by the controller or an application. If duplicate matches are found for a packet at the end of the matching process, then the switch arbitrarily selects one of the best matching flow-entries. As packets are matched, the switch updates counter fields in the matching flow-entry which provides the controller with switch statistics.

3. Related Work

Studies into SDN security have largely resulted in the development of systems that tackle security issues associated with the use of OpenFlow. For example, researchers have proposed improvements to enable the prioritisation of security related flow-entries [18, 19]. Implementation of such improvements has been complicated by the lack of a standardised northbound API and compatibility issues between different SDN switches and controllers. The following paragraphs summarise the most important proposed security systems in SDN.

Feamster et al. [20] present a system for implementing security policies in SDN, including data-caps and access control. Their system, called Procera, involves scanning would-be users of the network for vulnerabilities. Users that pass the scanning process are authenticated and granted network access until inactivity causes them to time-out from the network and the process can repeat.

FortKNOX [18] is a role-based authentication and Security Enforcement Kernel (SEK) designed for use with the C++ based NOX SDN controller using OpenFlow. The SEK is a control layer component that offers additional security features alongside the NOX controller. FortKNOX is designed to detect and reconcile conflicting flow rules before they are installed in a switch.

FRESCO [19] is an SEK that provides an API for modular applications, a key-management system, an IP reputation model, and a set of Python/OpenFlow-based actions to enforce security. The authors outline an anomaly detection system that is based on two algorithms: Credit-Based Threshold Random Walk (CB-TRW) and Rate Limiting (RL). They test their system using TCP and UDP network traffic. This pairing of algorithms has previously been shown to perform well in simulations using TCP network traffic [21, 19].

SDN-Mon [22] is an SDN-based framework for efficiently providing switch statistics that can be used by a variety of monitoring applications. The authors evaluate the performance of their system and show that it can provide improved monitoring capabilities in a trade-off with switch speed.

Recent studies into SDN intrusion detection systems have shifted towards machine-learning and deep-learning techniques. For example, Tang et al. [23] present an anomaly-based IDS which uses a Gated Recurrent Unit Recurrent Neural Network (GRU-RNN) algorithm that is accurate and inexpensive. Abubakar et al. [24] present a machine-learning IDS for SDN, which provides Snort-based signature detection via a tap on network switches. Their system is supported by an anomaly-based intrusion detection system to enable mitigation against zero-day attacks.

Kreutz et al. [25] have produced a set of attack countermeasures (shown in Table 1) for OpenFlow-based SDN. Our current work has made inroads into the following four countermeasures: attack detection, event filtering, firewall and IPS, intrusion tolerance, packet dropping, and rate limiting.

4. IDPS Design and Implementation

4.1. IDPS Overview

Figure 2 outlines network security in SDN and is used to frame the scope of this work: the implementation of an SDN-based IDPS. Our IDPS relies on the information that is received by the network controller. We intercept PacketIn events and use them to detect anomalies in the network traffic.

In this section, we outline three anomaly detection algorithms, also including their pseudocode. Firstly, we introduce an anomaly detection technique that acts as a countermeasure against TCP-based port-scanning. We then describe our implementations of CB-TRW and RL, which have been designed to minimise the amount of hub-like forwarding through network switches compared to existing implementations. For example, in the case of a DoS attack, hub-like behaviour in switches is undesirable because it amplifies the number of packets that can be sent by an attacker which can impact the network. In addition to these changes, our RL algorithm has been extended to include anomaly detection for TCP, UDP and ICMP network traffic.

4.2. Implementation of Anomaly Detection Against Port-Scanning: Port Bingo

Our countermeasure against port-scanning is based on the principle that the attacker will send packets that are destined to a large number of different ports

Table 1: Attack Countermeasures in OpenFlow Networks.

Measure	Description
Access control	Provide strong authentication and authorisation mechanisms on devices.
Attack detection	Implement techniques for detecting different types of attacks.
Event filtering	Allow or block certain types of events to be handled by special devices.
Firewall and IPS	Tools for filtering traffic, which can help to prevent different types of attacks.
Flow aggregation	Course-grained rules to match multiple flows to prevent information disclosure and DoS attacks.
Forensics support	Allow reliable storage of traces of network activities to find the root causes of different problems.
Intrusion tolerance	Enable control platforms to maintain correct operation despite intrusions.
Packet dropping	Allow devices to drop packets based on security policy rules or current system load.
Rate limiting	Support rate limit control to avoid DoS attacks on the control plane.
Shorter timeouts	Useful to reduce the impact of an attack that diverts traffic.

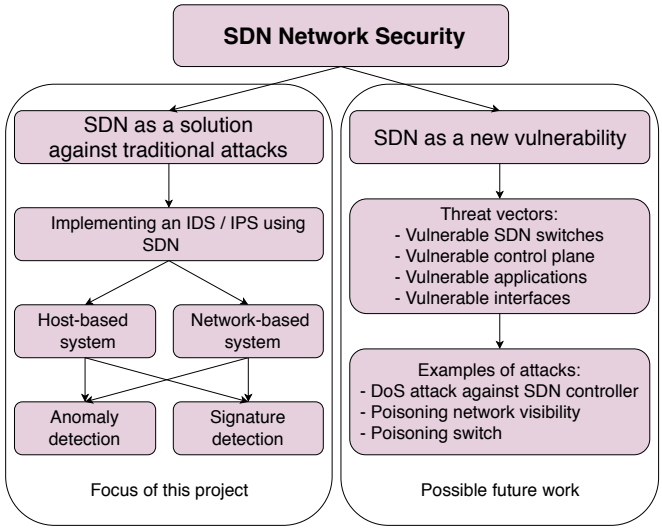

Figure 2: Framework for implementing an IDPS using SDN and focus area of this project.

and that scans will typically prioritise the most valuable TCP port probes in descending order of accessibility. We monitor the TCP packet headers between pairs of network hosts and compare the packet destination ports with a list of twenty ports associated with port-scanning. This is reffered to as the Port Bingo (PB) algorithm and is shown in Algorithm 1. We define an anomaly as an accumulation of a subset of these ports in communications from one host to another, where the subset is equal in length to some threshold value and which has been accumulated in a period that is less than the maximum period used for tracking.

Algorithm 1 IDPS Pseudo-code: Port Bingo (PB)

```
1:  Variables:
2:  top_tcp_port_probes, [80,25,22,443,21,113,23,53,554,3389,1723,389,636,...]
3:  threshold, var1
4:  timeout, var2
5:  tracked_connections, []
6:          ▷ format of a tracked connection: src, dst, set_of_dst_ports, time
7:  procedure INPUT(PacketIn intercepted from the controller)
8:      if PacketIn.type = TCP then
9:          ensure connection is tracked                    ▷ Use Ethernet addresses
10:         delete tracked connection if it has timed-out
11:         if tcp_dst_port in top_tcp_port_probes then
12:             Add tcp_dst_port to tracked_connection.set_of_dst_ports
13:             if len(tracked_connection.set_of_dst_ports) > threshold then
14:                 Log a warning to the POX console
15:                 Set flow-entry: drop packets from attacker
```

4.3. Implementation of CB-TRW

CB-TRW is based on the assumption that TCP connection requests made by a benign network host will generally be successful. If the host initiates a TCP connection with the server by transmitting a TCP[SYN] packet across the network via a switch, assuming the switch does not have a flow-entry to forward the packet, it will be sent to the controller and through the CB-TRW algorithm which will increase the tally of unsuccessful TCP connections by one. If the server replies to the host with a TCP[SYN,ACK] packet, then the TCP connection turned out to be successful, and so the tally of unsuccessful connection initiations is decreased by one. An anomaly occurs when the number of TCP[SYN] packets sent from the host to the server, compared to the number of TCP[SYN,ACK] packets from the server to the host, exceeds a predefined threshold. When an anomaly is detected, the IDPS fires an alert to the POX console and a flow-entry is installed in the network switch to drop packets from the attacker (the default setting is to drop packets indefinitely). Algorithm 2 shows the pseudo-code for the CB-TRW implementation. In the pseudo-code, "Return" indicates the application is exited and the PacketIn

is returned to the controller for normal processing. Since the attributes of PacketIn are accessed using dot notation, this is also used in the pseudo-code. For example, PacketIn.src means the source address of PacketIn.

Algorithm 2 IDPS Pseudo-code: CB-TRW

```
Require: PacketIn intercepted from the controller
1:  THRESHOLD ⇐ 50                                  ▷ threshold for anomaly
2:  tracker ⇐ []
3:  if PacketIn.type != TCP then return
4:  if PacketIn.flags != SYN or SYN,ACK then return
5:  if PacketIn.flags = SYN and not ACK then
6:      if PacketIn.src not in tracker then
7:          Add PacketIn to tracker
8:      else
9:          Increase counter by 1
10:         if counter value > THRESHOLD then
11:             Log a warning to the POX console
12:             Set flow-entry: drop packets from attacker
13: else                                            ▷ Packets are TCP [SYN,ACK]
14:     if PacketIn.src in tracker then
15:         Decrease counter by 1
```

Figure 3 shows the IDPS design, outlining the separate detection (IDS) and prevention (IPS) functionality. In the IDS, the functionality of CB-TRW is shown. In Fig. 3, "Return" indicates the PacketIn is returned to the controller for normal processing.

4.4. Implementation of Rate Limiting

RL is based on the premise that a benign host is unlikely to make many connection-initiations in a short amount of time, whereas an attacker is more likely to do so. The same algorithm can be generalised to include connectionless protocols such as UDP. In the latter case, an anomaly is defined as an excessive number of UDP packets sent from one host to another. To mitigate against TCP-based attacks, the IDPS tracks TCP[SYN] packets sent by a host to a server, flagging an anomaly if the rate of connection-initiations exceeds the threshold value. After some predefined time (e.g., one minute) if the rate-limiting threshold has not been exceeded then the tracked connection is deleted. If an anomaly is detected, the IDPS logs an alert to the POX console and installs a flow-entry in the switch to drop packets from the attacker. Algorithm 3 shows the pseudo-code for our implementation of the RL algorithm.

4.5. Implementation of Attack Blocking and QoS Based on Flow-Statistics

Open vSwitch enables queuing on switch ports which can be used to apply QoS to flow-entries. Our implementation of QoS is designed to detect flooding

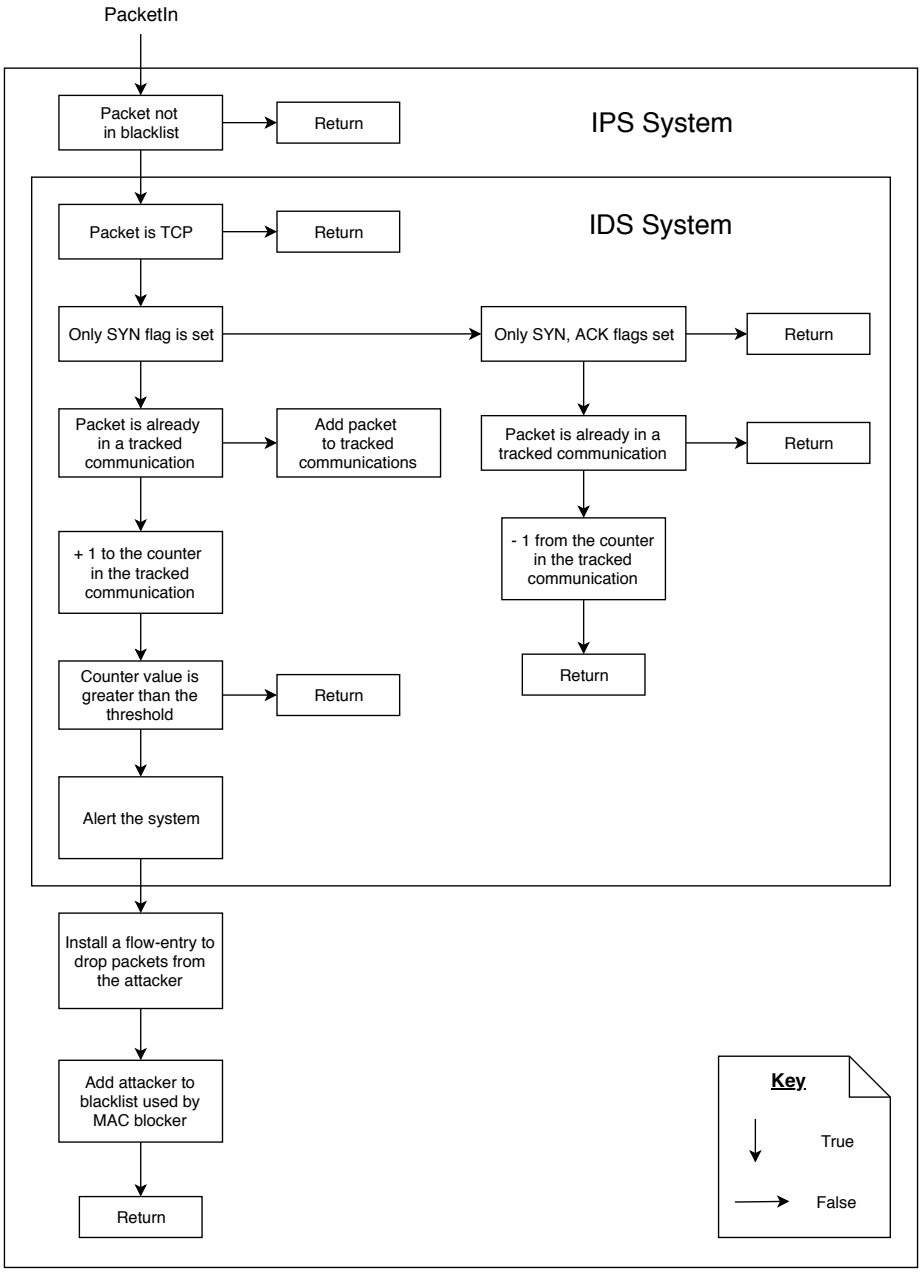

Figure 3: IDPS Design: IDS (CB-TRW) and IPS Functionality

Algorithm 3 IDPS Pseudo-code: Rate Limiting

Require: PacketIn intercepted from the controller

```
1: THRESHOLD ⇐ 50                                          ▷ threshold for anomaly
2: tracker ⇐ []
3: if PacketIn.type in [TCP,UDP,ICMP] then
4:     ensure src=>dst connection is tracked
5:     delete tracked connection if it has timed-out
6:     if PacketIn.type = TCP then
7:         if PacketIn.flags = SYN then                    ▷ Only SYN
8:             Increase TCP RL counter by 1
9:     else if PacketIn.type = UDP then Increase UDP RL counter by 1
10:    else if PacketIn.type = ICMP then Increase ICMP RL counter by 1
11: if counter value > THRESHOLD then
12:    Log a warning to the POX console
13:    Set flow-entry: drop packets from attacker
```

attacks. This is achieved by routinely checking the flow-entry statistics for anomalies. Specifically, we check for an excessive byte-count or an excessive packet-count in flow-entries that carry TCP, UDP, or ICMP packets between two hosts. If an anomaly is detected, the IDPS generates a flow-entry to enqueue subsequent packets on the appropriate egress switch port. Algorithm 4 shows the pseudo-code for our implementation of the QoS algorithm.

5. Experimental SDN Testbed

For the experimental part, we built the SDN testbed shown in Fig. 4. The virtual network runs on a Dell Inspiron with 8GB RAM, an Intel Core i5-3337 CPu and a 64-bit OS, running Ubuntu 16.04 LTS Desktop. Experimentation is performed using the POX controller (Eel version), OpenFlow 1.0, OvS 2.7, and Linux hosts. Ubuntu kernel version 4.4 is used on the testbed machine due to compatibility requirements with OvS 2.7.

We used a virtual Ethernet learning switch, which "learns" how to forward packets by using information from previously received packets. Specifically, the switch correlates the source MAC address of a packet with the switch-port number that the packet was received on. Subsequently, if a packet is received with a destination MAC address that is in the {source MAC address : switch-port number} mapping data, the switch directs the packet towards its destination by sending it out on the switch-port in the mapping. For example, if a packet is received at the switch on port 3 and the packet header includes the source MAC address 08:00:00:00:02:00, the switch can use this information to direct subsequent packets to 08:00:00:00:02:00 out through port 3. Flow-entries include a parameter for the output port.

An OvS bridge named 'ovs-br' is used to connect four VMs. Wireshark is used to capture packets received by ovs-br. VirtualBox 5.2.12 is used with the

Algorithm 4 IDPS Pseudo-code: QoS

Require: Request flow-statistics every five seconds

t1, t2, t3, ... ▷ set threshold values for anomaly
for item **in** flow-statistics **do**
    if network protocol = TCP **then**
        if num_bytes > t1 or num_pkts > t2 **then**
            Set flow-entry: drop TCP packets between these hosts
        else if num_bytes > t3 or num_pkts > t4 **then** Set flow-entry: queue TCP packets between these hosts
    if network protocol = UDP **then**
        if num_bytes > t5 or num_pkts > t6 **then**
            Set flow-entry: drop UDP packets between these hosts
        else if num_bytes > t7 or num_pkts > t8 **then** Set flow-entry: queue UDP packets between these hosts
    if network protocol = ICMP **then**
        if num_bytes > t9 or num_pkts > t10 **then**
            Set flow-entry: drop ICMP packets between these hosts
        else if num_bytes > t11 or num_pkts > t12 **then** Set flow-entry: queue ICMP packets between these hosts

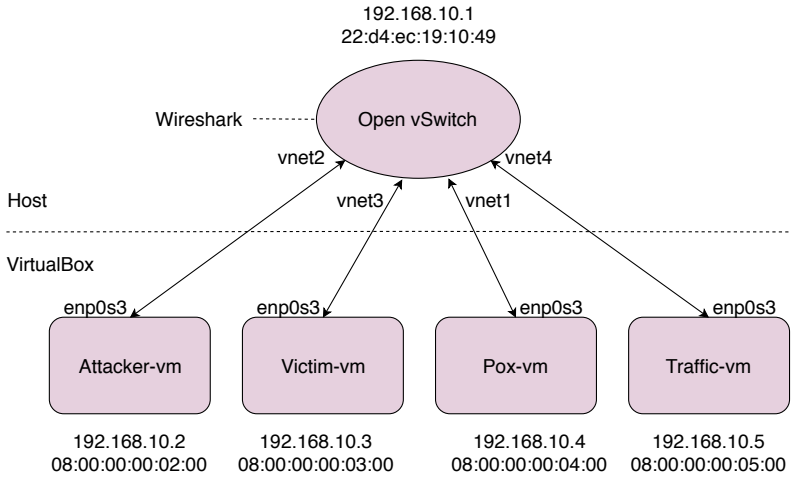

Figure 4: Experimental SDN Testbed.

default settings as the VM hypervisor. Ubuntu 16.04 LTE Desktop is used on the host laptop and as the OS for each VM. The four VMs each have 1024 MB of memory, an Intel PRO/1000 MT Desktop virtual network adapter and a bridged connection to the virtual network. They are connected to the switch with virtual 10 Mbps cables. Each VM is assigned with a static IP address and

an easily identifiable MAC address. Table 2 outlines the VM configurations.

Table 2: VirtualBox VMs created for the testbed

VM Name	Network	IP Address	MAC Address	Software
Attacker-vm	Bridged	192.168.10.2	08:00:00:00:02:00	Nmap, LOIC
Victim-vm	Bridged	192.168.10.3	08:00:00:00:03:00	
Pox-vm	Bridged	192.168.10.4	08:00:00:00:04:00	POX
Traffic-vm	Bridged	192.168.10.5	08:00:00:00:05:00	Tcpreplay

The attacker was equipped with Nmap - a popular network scanning tool [26] and three attack scripts: a purpose-built Python DoS attack script, Dos.py, and two popular DoS scripts, Hammer.py [27] and the Low Orbit Ion Cannon (LOIC) [28]. Dos.py, attempts to send 10K TCP[SYN] packets to the victim, containing a short payload.

Victim-vm was used as a Web server. Traffic-vm was used to send regular HTTP GET requests to the victim's Web server, measuring the round-trip time for the request and response. This was used to indicate the state of the network. The time taken for each request to get a response was logged to file for analysis. Traffic-vm was also used to generate benign traffic using Tcpreplay, a program that can generate network traffic from packet-capture files [29].

6. Anomaly-Based IDPS: Port-Scanning

6.1. Experiment Description

In this experiment, we tested the IDPS against an Nmap port-scan. Initially, we looked at the effect of an unmitigated port-scan on the network. Using Wireshark, we recorded a packet capture showing the Nmap port-scan across OvS interface. The IO-graph of this is presented in Figure 5 and shows that the network traffic from the attack peaked at about 1.5k packets per second (pps) and lasted for about ten seconds. The Wireshark packet capture shows us that the port-scan is using TCP packets and so we would expect our TCP-based algorithms to potentially detect the scan. We used the following threshold settings in the IDPS: PB=10, CB-TRW=1000, and RL-TCP=1000. Our TCP-based QoS algorithm would not be expected to detect this scan since the flow-stats would not identify any anomalies in the number of bytes or the number of packets contained in the flow-entries.

To test the IDPS in our simulation, OvS and Wireshark were started on the host laptop and we used Wireshark to record the network traffic passing through the switch. The POX controller was invoked with the functionality of a virtual Ethernet learning switch and our IDPS:

```
~/pox/pox.py forwarding.l2_learning idps
```

Victim-vm was running a Web server and so port 80 was open:

```
sudo python3 -m http.server 80
```

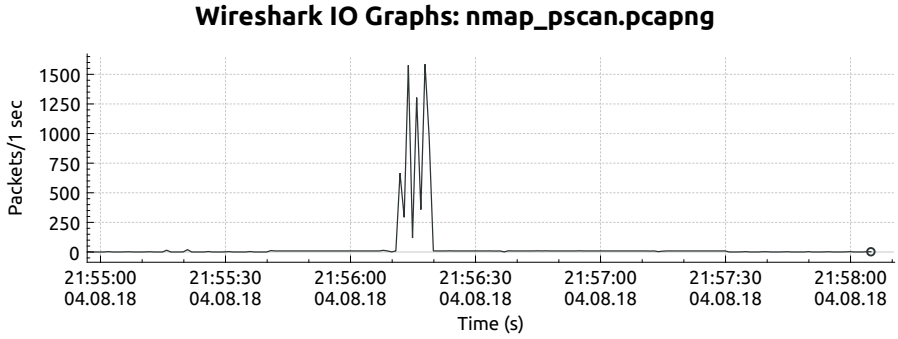

Figure 5: Port-Scan IO Graph.

Traffic-vm measured the impact of the port-scan on the network using a Python script to periodically measure the time to get a response from an HTTP GET requests to the Web server. At this point, the attacker attempted a port-scan on the Web server:

```
nmap 192.168.10.3
```

6.2. *Results and Analysis*

Figure 6 shows the POX console after an intrusion has been detected. An alert has been logged at the level 'WARNING' which shows that our algorithm 'pb_tcp'; the PB algorithm, has detected a port-scan. There is information about the Ethernet addresses involved and the time of detection.

```
celyn@pox-vm:~$ ~/pox/pox.py log.level --WARNING --openflow=INFO --openflow.disc
overy=INFO forwarding.l2_learning idps
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:idps:IDPS going up 2019-02-19 22:35:30.507889
INFO:openflow.of_01:[5a-42-ff-ba-52-4e 1] connected
WARNING:idps:Attack Mitigation using algo: pb_tcp, connection: 08:00:00:00:02:00
 => 08:00:00:00:03:00, time: 22:36:00.045410
```

Figure 6: POX Console: Port-Scanning Intrusion Detection

A flow-entry has been generated by the IDPS to drop packets from the Ethernet address of the attacker to the Ethernet address of the victim, as shown in Figure 7.

```
cookie=0x0, duration=59.026s, table=0, n_packets=587, n_bytes=43324, idle_age=12,
dl_src=08:00:00:00:02:00,dl_dst=08:00:00:00:03:00 actions=drop
```

Figure 7: Flow-Table: Port-Scanning Intrusion Prevention

Wireshark shows us information about the port-scan and our ability to mitigate it. Figure 8 shows packets from the IP address of the attacker to the victim. Since, the attacker is not sending normal flow-based network traffic, each of the packets in the port-scan are first sent to the controller so that they can be forwarded by the switch. As a result, the IDPS is able to check the range of destination ports in the communications and our PB algorithm can detect and block the attack by the time the attacker has sent only tens of packets. Actually, the number of packets that have been sent by the attacker as shown at the bottom of the display in Figure 8 include both the packets sent by the attacker to the controller and the subsequent packets sent by the controller to the switch in order to forward the packets. Therefore, according to the display, the attack has been mitigated after twenty six packets have been sent by the attacker. This figure may be improved with further tuning.

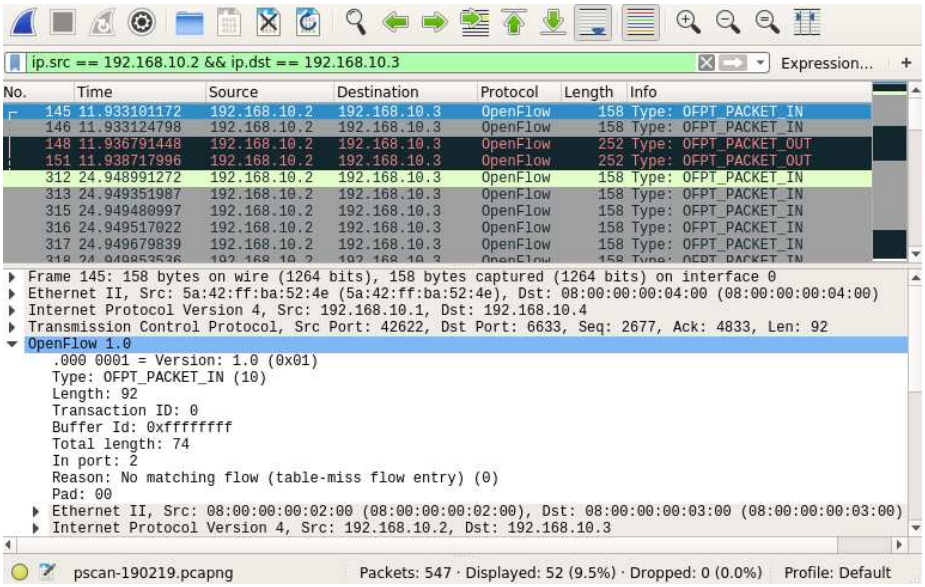

Figure 8: Wireshark: Port-Scanning

If intrusion detection using PB is switched off, we might expect our other TCP-based algorithms, CB-TRW and RL-TCP, to perform. The basic Nmap port-scan requires an attacker to send 1000 packets to the victim and we tested CB-TRW and RL-TCP with threshold settings at 1000. Both of the algorithms were able to detect the port-scan. In the case of CB-TRW, every connection initiation failed and so the intrusion detection required an attack composed of at least a number of packets equal to the threshold used by the algorithm. Likewise, the RL algorithm required the same number of connection-initiation packets to identify an anomaly. The associated Wireshark IO graph for the experiment looked very similar to Figure 5. With these settings, the attack

mitigation did not limit the attack, since all packets were able to be transmitted before communications were dropped by installing an appropriate flow-entry in the switch. However, the attacker was unable to send any subsequent packets on the network. Lower threshold settings for these algorithms would better detect the attack but would attract more false positive alerts caused by benign network traffic.

7. Anomaly-Based IDPS: DoS Attacks

We tested the IDPS using a range of DoS attacks. The experiments were carried out twice for each of the attacks to observe the characteristics of the attacks and to measure the effectiveness of the IDPS; the attacks were performed once with the IDPS switched off and once with the IDPS switched on. Between each experiment, POX was restarted, the switch flow-table cleared, and other scripts were restarted as necessary. For example, the Victim-vm Web server was restarted if it had crashed. The network configuration, shown in Figure 4, is the same as in the previous experiment.

7.1. Brute-force DoS Attack

In this experiment, the attacker attempted a TCP-based DoS attack using the Python script Dos.py. Using a Wireshark display filter it is possible to examine the sequence of packets that the attacker sent to the victim:

```
ip.src == 192.168.10.2
```

The source port of the packets sent by the attacker changed with every transmitted packet. Since the default switch setting is to generate flow-entries that do not include wildcard values for the ports, this means that a unique flow-entry was required for each packet in the attack. According to Wireshark, the first packet of the attack was received at ovs-br was at 20:25:16.37 and the last was sent at 20:26:05.32, meaning that the attack continued for approximately 49.0 seconds.

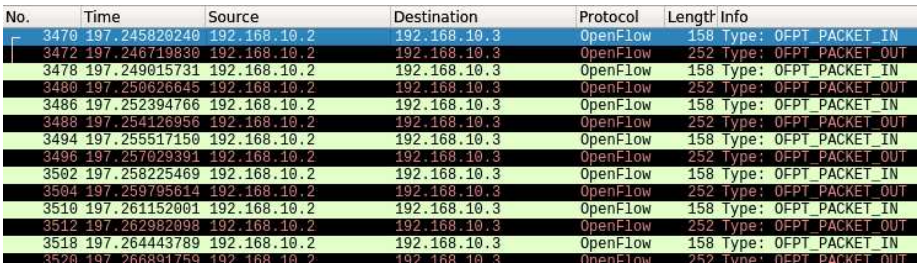

Figure 9: Brute-Force DoS Attack Against Victim Server.

If the Python attack script Dos.py is set to generate 10K packets, Wireshark shows that 20K packets are sent by the attacker. There is a simple pattern

in the packets that are sent by the attacker: one TCP[SYN] packet that is identified by Wireshark's colouring rules as "HTTP" or ordinary HTTP traffic; followed by one TCP[SYN] packet coloured in black that is identified as "Bad TCP", which means the packet is suspected of being a frame segment that is out-of-sequence. Fig. 9 shows a section of the Wireshark output, displaying the repeating pattern of TCP packets that are sent from the attacker to the victim. This repeating pattern in the transmission of packets happened throughout the attack: a PacketIn packet followed by a PacketOut. This behaviour arises when there is no matching flow-entry for a packet: the attacker sends a packet to the switch and then the packet header gets encapsulated using the OpenFlow protocol and sent to the controller as a PacketIn packet.

In theory, there are three possible explanations as to why the switch sends a PacketIn to a controller: (i) a table-miss entry, (ii) TTL checking, or (iii) a "send to the controller" action [30]. Inspection of several packets in Wireshark show the OpenFlow packet encapsulation, which includes the explanation for sending the PacketIn: "Reason: No matching flow (table-miss flow-entry)". The controller returns the packet header to the switch as a PacketOut, which is encapsulated within another OpenFlow packet; a FlowMod packet, as shown in Fig. 10. The parameters in the FlowMod are used to set a flow-entry in the switch. The packet has been buffered at the switch and can now be forwarded to the victim.

The reason that each packet in the attack requires a new flow-entry is down to the fact that every time the attacker transmits a new packet, the source port number is increased by two. Figure 10 shows the FlowMod (denoted as OFPT_FLOW_MOD), i.e. the flow-entry, sent by the controller, showing that no wildcards are in the flow-entry, and since the source port number increases with each packet sent by the attacker, there is never an existing flow-entry in the switch to forward a packet in the attack.

Even though this type of TCP-based packet transmission is not being handled well by the l2 learning switch (every packet to the switch is delayed by a round-trip to the controller), in terms of tracking the connection between the attacker and the victim, and the number of TCP[SYN] packets which are sent to the victim, it is not a problem for the IDPS algorithm.

CB-TRW does not detect this attack because the server is very co-operative; the packets from the victim to the attacker are very similar to the packets sent by the attacker. The victim server responds to the attacker with 20K packets. 10K packets from the victim are TCP[SYN,ACK] packets; the type of packets that decrease the CB-TRW counter used to indicate an attack. All of the packets are sent by the switch to the controller for a flow-entry. The implementation of CB-TRW cannot detect this attack because there are equal numbers of TCP[SYN] packets and TCP[SYN,ACK] packets in response. Therefore, the algorithm identifies successful TCP connections and not an attack.

Another characteristics of the attack that is measurable, is the disproportionate number of non-session-based TCP streams as compared to the number of session-based TCP streams. Every TCP stream in the attack is four packets in total. This information could potentially be used to set an anomaly that

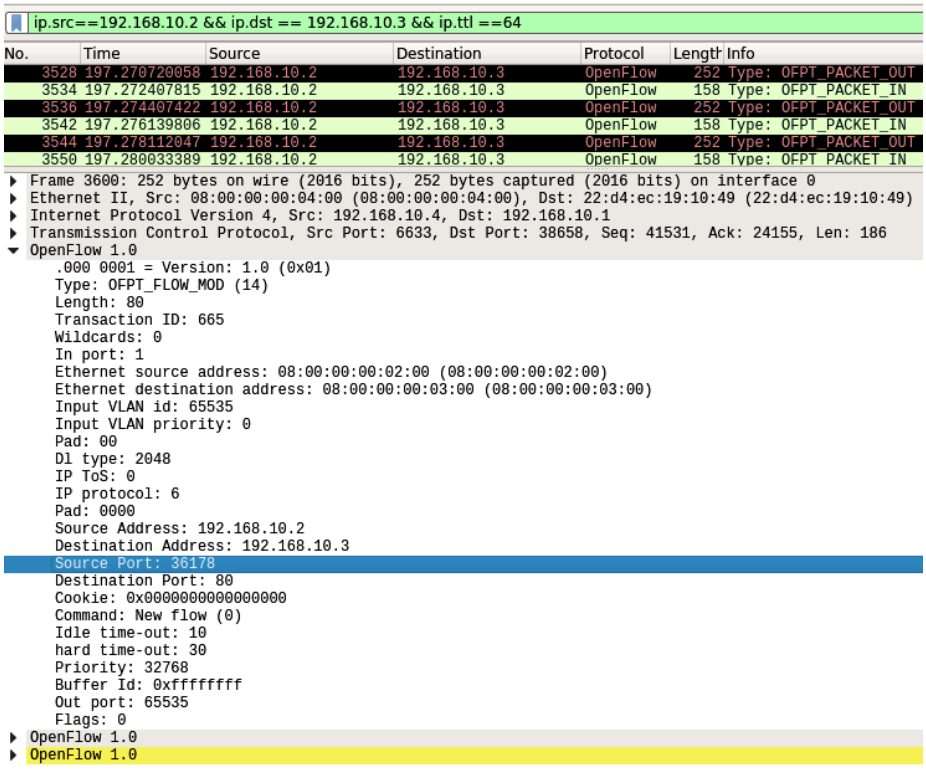

Figure 10: OpenFlow Packet Encapsulation - Flow-Entry.

triggers the IDPS when some threshold is exceeded. These considerations are left for future work.

In summary, using Wireshark to analyse the attack showed that even though only 10K packets were actually transmitted from Attacker-vm, 20K packets were transmitted with the source IP address of the attacker. Further inspection showed that every packet sent by the attacker to the switch, was sent by the switch to the controller and then returned by the controller with a flow-entry to enable the packet to be sent to the victim. Likewise every packet sent by the victim to the attacker also included a round-trip from the switch to the controller and back for a flow-entry. Even though CB-TRW did not detect this attack, we believe that this analysis of the network traffic created by the attack indicates an advantage of our implementation of CB-TRW, in the sense that in existing implementations, each packet from the attacker would be first sent to the controller and then flooded out of the switch on all ports, leading to a significant increase in the number of packets associated with the attack.

This brute-force DoS attack identified a limitation in the CB-TRW algorithm

used by the IDPS; the server returned a TCP[SYN,ACK] for every TCP[SYN] packet sent by the attacker and so there was no anomaly. The intrusion was detected by the RL algorithm; an excessive number of TCP connections were initiated by the attacker. It also highlights one of the flaws in OpenFlow-based SDN: the volume of traffic which can be sent to the controller under certain conditions can be a concern, especially if the controller is physically remote from the switch and the round-trip time for packets between a switch and a controller are increased. However, RL uses a detectable characteristic of the attack; the excessive rate of connection initiations, and blocks the attack.

7.2. Hammer DoS Attack

The attacker used the Python script Hammer.py to launch a DoS attack against the victim's Web server. The IDPS detected the attack and installed a flow-entry in the switch to drop packets from the attacker's MAC address. The attack crashed the server instantly and so Traffic-vm's HTTP requests to the victim all timed out once the attack started.

According to Wireshark, the first packet of the attack was received at ovs-br at 20:31:57.37 and the final packet sent by the attacker was at 20:31:57.63, meaning that the attack lasted for approximately 0.25 seconds before it was prevented by the IDPS.

After the attack, the flow-table included a flow-entry to drop all packets from the attacker's MAC address, as outlined by the following Bash command and subsequent flow-entry:

```
sudo ovs-ofctl dump-flows ovs-br
```

```
cookie=0x0, duration=104.670s, table=0, n_packets=785, n_bytes=58170,
    idle_age=0, priority=65535,dl_src=08:00:00:00:02:00 actions=drop
```

Similarly to the previous experiment (brute-force DoS attack), each TCP packet stream was four packets in length until the final three TCP packet streams, which are TCP streams of only two packets; a packet to the switch which is sent to the controller followed by a packet from the controller to the switch. These three streams do not include any packets from the victim's server and mark the end of the attack. At this stage, the attacker's console was filled with the line 'no connection! server maybe down'.

The IDPS blocked the attacker almost instantly after which point a permanent flow-entry was installed in the switch to drop packets from the attacker and the attacker was added to a blacklist, used for MAC blocking any further packets to the controller. This approach can also be used as a possible mitigation against IP spoofing attacks.

7.3. LOIC DoS Attack

In this experiment, the attacker used LOIC to launch a DoS attack against the victim's Web server. The attack was mitigated using the RL algorithm. The HTTP requests made by Traffic-vm to the server began to timeout soon after

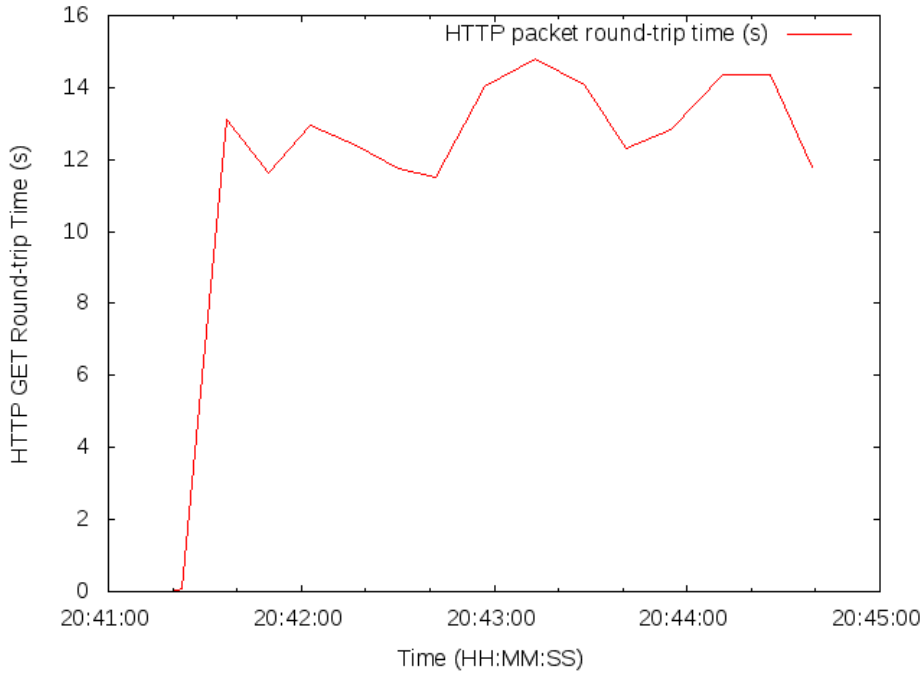

Figure 11: Round-Trip Time For a Benign Packet.

the attack started, as shown in Fig. 11, so the attack impacted the network before it was prevented.

Analysis of the packet-capture shows that CB-TRW cannot be used to detect this attack because the victim's server has returned a sufficient number of TCP[SYN,ACK] packets; the criterion for a successful connection (CB-TRW would have succeeded had the server crashed). An additional experiment was conducted with RL switched off to verify the latter claim. According to Wireshark, the first packet of the attack arrived at ovs-br at 20:41:24.55, and the final packet of the attack, at 20:50:08.46 when the attack was stopped - the attack would have continued indefinitely. In this time, a similar pattern emerged as was described in previous attacks. 518 packets were transmitted from the attacker to the victim; 249 PacketIn packets from the switch to the controller and 269 PacketOut packets from the controller to the switch. The extra twenty PacketOut packets are likely re-transmissions. Each packet required a new flow-entry. As with previous experiments, each of the TCP packets sent by the attacker was transmitted on a different source port (the source port increased by two with each successive TCP packet from the attacker). Furthermore, the TCP packet streams are all four packets in total; two TCP[SYN] packets from the attacker and two TCP[SYN,ACK] packets from the attacker. Assuming that

a TCP stream occurs mainly between the initial three-way handshake and the finishing sequence, then the TCP streams should usually be much greater than four packets, and the majority of the packets in a TCP stream should not have the SYN flag set. This feature may be useful in the specification of an anomaly detection technique.

In summary, in this experiment, the victim server returned a sufficient number of TCP[SYN,ACK] packets to mitigate the occurrence of an anomaly according to CB-TRW. The server remained up throughout the attack but the HTTP GET requests made by Traffic-vm to the server, began to time out soon after the attack started. The IDPS detected and prevented the attack using the RL algorithm. Table 3 outlines the results of the four attacks described in this section.

Table 3: Summary of the results from experiments of Section 8.

	Scan	**Brute-force**	**Hammer**	**LOIC**
Intrusion Detection	CB-TRW	RL	CB-TRW	RL
Intrusion Prevention	Yes	Yes	Yes	Yes
Server crashed	No	No	Yes	No[1]

8. IDPS Algorithms: Threshold Testing Using a Network Traffic Dataset.

In this experiment, we tested the IDPS using a dataset of benign network traffic. We used the results of this experiment to determine appropriate threshold values for the algorithms in the IDPS. The independent variable in our experiment is the threshold value within the IDPS algorithms that differentiates between normal network traffic and network attacks. The dependent variable is the number of false positives that are generated by the IDPS. We also measured the CPU usage of the process that is used to run the Pox controller and the IDPS; python2.7. We have limited the period of time in which the IDPS can detect an anomaly to sixty seconds, since it is sufficient time to detect a DoS attack and a normal Nmap port-scanning attack, while keeping memory usage low in the IDPS. Other variables are assumed to be constant.

The network traffic dataset is a packet capture file containing five minutes of real network traffic from the access point of a busy private network to the Internet [31]. The traffic includes 133 unique Ethernet addresses, 132 different applications, an average throughput of 3k pps, and a peak throughput of 5k pps. Figure 12 shows the IO-graph of the dataset and specifications of the pcap file are presented in Table 4. The dataset was replayed to the switch using Tcpreplay, which provided the necessary switch-controller network traffic.

[1]The victim server did not crash but HTTP requests from Traffic-vm started to time-out

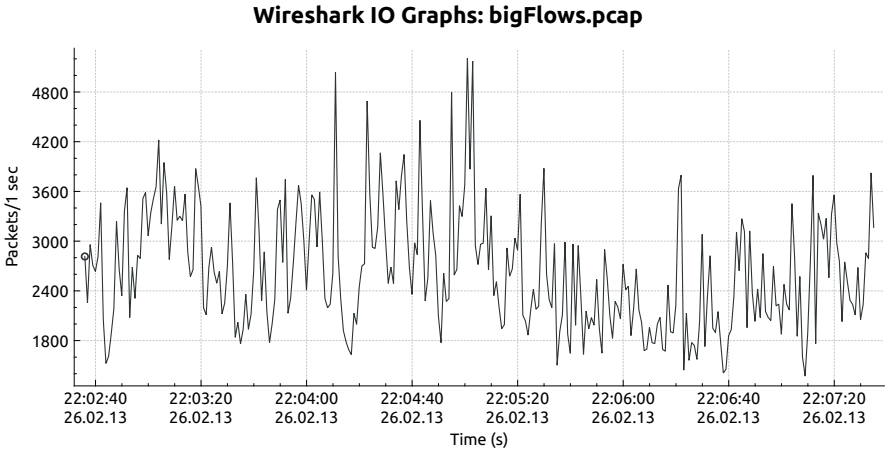

Figure 12: Dataset: Volume of Network Traffic.

Table 4: Specifications for bigFlows.pcap.

Parameter	**Value**
Date and time recorded	26th February 2013 10pm GMT
Size	368 MB
Packets	791615
Percent TCP	80.1
Percent UDP	19.3
Percent ICMP	0.5
Flows	40686
Average packet size	449 bytes
Duration	5 minutes
Number of applications	132
Number of MAC addresses	133

Experiments were performed three times for reliability and Table 5 shows the averages of our results. Figure 13 is a graph that shows the relationship between the threshold setting and the number of false positives for each of the IDPS algorithms. Table 5 shows combined CPU usage of the POX controller and the IDPS across the range of IDPS settings and Fig. 14 shows how the CPU usage changes if the IDPS is switched on or off. At a low threshold setting, the CPU usage falls when the IDPS is switched on. In general terms, this is because the process of mitigating against anomalies is cheaper than forwarding packets using the l2_learning component that is provided with the POX controller. However, Table 5 shows that decreasing the threshold settings can result in an increased number of false positive alerts.

TCP packets make up about eighty percent of our network traffic, and so

it is not surprising that CB-TRW and RL-TCP result in the highest numbers of false positives. UDP traffic makes up almost twenty percent of the total traffic, about a quarter as much as TCP traffic, but the results show that the number of false positives generated by RL-UDP are higher than RL-TCP at low thresholds and no false positives were recorded by RL-UDP at thresholds greater than 100. Given that ICMP packets make up only one half of a percent of the total traffic, RL-ICMP provides a disproportionately high number of false positives at low thresholds. Finally, the PB algorithm does not generate any false positive alerts at order of magnitude settings greater than one. At a threshold setting of ten, the PB algorithm fires an alert if a network host tried to send packets to another host using ten of the twenty top destination ports that are commonly associated with network scanning.

Table 5: False Positive Alerts and %CPU Usage Results from Testing the IDPS Using Benign Network Traffic.

Algorithm	**False Positive Alerts**					**%**
Threshold	**CB-TRW**	**RL-TCP**	**RL-UDP**	**RL-ICMP**	**PB**	**CPU**
IDPS off	-	-	-	-	-	96.4
1	3,742	3,742	3,814	446	18,710	47.4
10	548	548	812	61	0	74.8
100	72	72	42	5	0	94.3
1k	4	4	0	0	0	97.6
10k	0	0	0	0	0	96.9

The selection of appropriate threshold values in the IDPS depends on network security policy and should be adjusted to match the sensitivity of the network. We propose the use of order of magnitude algorithm threshold values to set our IDPS for any further testing against network attacks, as shown by Table 6.

Table 6: Proposed Order of Magnitude Threshold Settings For The IDPS.

Algorithm	**Threshold Setting**
CB-TRW	10,000
RL-TCP	10,000
RL-UDP	1,000
RL-ICMP	1,000
PB-TCP	10

Table 7 shows the results of experiments that use our implementation of attack blocking and QoS based on flow-statistics. The pseudocode of our implementation of QoS as an attack mitigation technique is shown in Figure 4 and it includes a mechanism to drop network traffic, or enter it into a queue on the appropriate switch port, or allow unmitigated packet forwarding

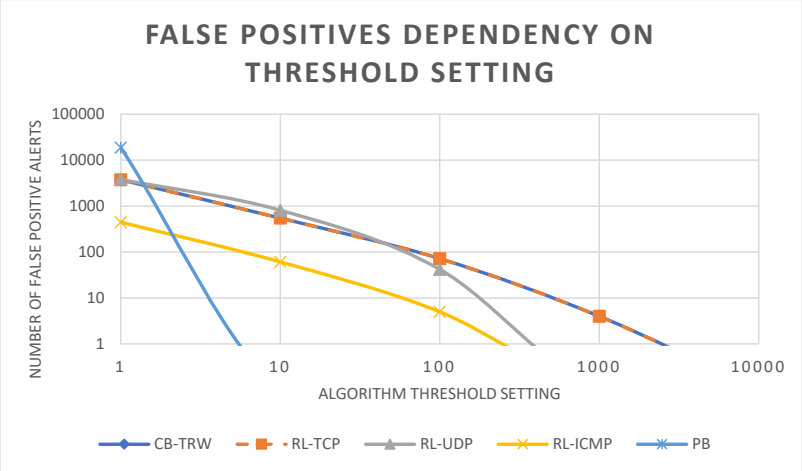

Figure 13: False Positives in Benign Network Traffic.

depending on the number of bytes or the number of packets in the flow-statistics that we retrieve periodically from the switch. The values in Table 7 show the number of false positive alerts that are generated by running the network traffic sample outlined in Table 4 through our virtual network, which is used to model the normal or expected network traffic and to set the boundaries that define anomalous traffic. We propose setting the IDPS to drop packets that are outside of the range of network traffic that is measured using benign network traffic datasets, and enqueue packets that are close to the boundaries of normal traffic flows. For example, Table 7 shows that the network traffic dataset did not include any flow-entries that had been used to forward 10k TCP packets or greater. So we would proceed to carry out further tests on the IDPS using a threshold value of 10k TCP packets before packets are dropped. We also see that there were only two alerts caused by flow-entries for TCP packets that had been used to handle more than 1k TCP packets. In this case, we would carry out further testing on the IDPS by enqueuing subsequent packets matching such flow-entries. In this way we propose the use flow-statistics to identify anomalies such as DoS attacks.

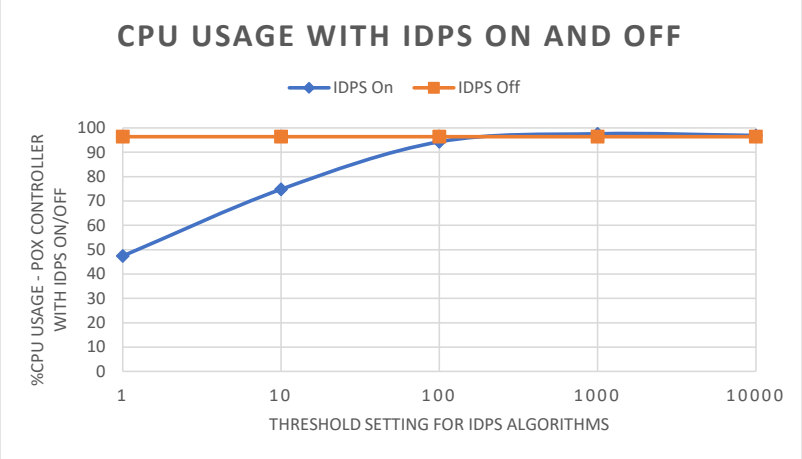

Figure 14: Percentage CPU usage of POX controller with IDPS switched on and off.

Table 7: False Positive Alerts Resulting from QoS Implementation Based on Flow-Statistics and Tested Using Benign Network Traffic Dataset.

- Threshold	False Positive Alerts					
	TCP		UDP		ICMP	
	bytes	pkts	bytes	pkts	bytes	pkts
10	-	329	-	30	-	8
100	-	169	-	314	-	0
1k	4548	2	920	9	6	0
10k	383	0	78	0	0	0
100k	134	0	24	0	0	0
1m	11	-	5	-	0	-
10m	0	-	0	-	0	-

9. Conclusions and Future Work

In this work, we have demonstrated that certain features of SDN can be used to both detect and prevent intrusions as well as to almost instantly drop packets when an attack is detected. An anomaly-based IDPS was designed, implemented, and tested. Two types of rate-based connection monitoring algorithms were used: CB-TRW and RL, including attacks based on TCP, UDP,

and ICMP. We introduced PB as a port-scanning detection technique. Furthermore, we included a QoS algorithm which relies on flow-statistics to defend against DoS attacks. According to the results from extensive experiments in a purpose-built SDN testbed, the IDPS was shown to be capable of detecting Nmap port-scans and various types of DoS attacks. This is done by monitoring communications between hosts on a network and taking automated steps to mitigate attacks. The intrusion detection part of the system included firing an alert to the POX controller console, to notify a user to the presence of an attack, showing which algorithm detected the attack, the time of the attack and a notification about the nature of the attack. The intrusion prevention part of the system automatically created and sent a flow-entry to network switches to drop packets from an attacker. Further testing was also carried out using real network traffic to measure the relationship between false positives, algorithm threshold settings, and the CPU usage.

Potential future work includes implementing countermeasures for a broader range of attacks and protocols. An interesting extension would be the addition of a machine learning module for anomaly detection. Further extensions may include: IDPS optimisation for resource-constrained IoT devices, incorporation of collaborative SDN controllers [32], or multi-step attack detection [33].

References

[1] Symantec, Internet Security Threat Report (ISTR), Volume 23, Tech. rep. (2018).

[2] S. Behal, K. Kumar, M. Sachdeva, D-FACE: An anomaly based distributed approach for early detection of DDoS attacks and flash events, Journal of Network and Computer Applications 111 (2018) 49–63. doi:10.1016/j.jnca.2018.03.024.

[3] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti, A survey of software-defined networking: Past, present, and future of programmable networks, IEEE Communications Surveys & Tutorials 16 (3) (2014) 1617–1634. doi:10.1109/SURV.2014.012214.00180.

[4] W. Li, W. Meng, L. F. Kwok, A survey on OpenFlow-based software defined networks: Security challenges and countermeasures, Journal of Network and Computer Applications 68 (2016) 126–139. doi:10.1016/j.jnca.2016.04.011.

[5] K. Cabaj, M. Gregorczyk, W. Mazurczyk, Software-defined networking-based crypto ransomware detection using HTTP traffic characteristics, Computers & Electrical Engineering 66 (2018) 353–368. doi:10.1016/j.compeleceng.2017.10.012.

[6] C. Yoon, T. Park, S. Lee, H. Kang, S. Shin, Z. Zhang, Enabling security functions with SDN: A feasibility study, Computer Networks 85 (2015) 19–35. doi:10.1016/j.comnet.2015.05.005.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74. doi:10.1145/1355734.1355746.

[8] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, K.-Y. Tung, Intrusion detection system: A comprehensive review, Journal of Network and Computer Applications 36 (1) (2013) 16–24. doi:10.1016/j.jnca.2012.09.004.

[9] Y. Zhang, W. Lee, Intrusion detection in wireless ad-hoc networks, in: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, MobiCom '00, ACM, 2000, pp. 275–283. doi:10.1145/345910.345958.

[10] K. Suzuki, K. Sonoda, N. Tomizawa, Y. Yakuwa, T. Uchida, Y. Higuchi, T. Tonouchi, H. Shimonishi, A survey on OpenFlow technologies, IEICE Transactions on Communications 97 (2) (2014) 375–386. doi:10.1587/transcom.E97.B.375.

[11] C. Beek, T. Dunton, S. Grobman, M. Karlton, N. Minihane, C. Palm, E. Peterson, R. Samani, C. Schmugar, R. Sims, D. Sommer, B. Sun, Mcafee labs threats report, june 2018, https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2018.pdf, accessed: 20-02-2019.

[12] L. F. C. Project, Open vSwitch, http://openvswitch.org/, accessed: 20-02-2019.

[13] J. McCauley, POX controller, https://github.com/noxrepo/pox, accessed: 20-02-2019.

[14] M. Casado, T. Koponen, S. Shenker, A. Tootoonchian, Fabric: A retrospective on evolving SDN, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, ACM, 2012, pp. 85–90.

[15] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, O. Koufopavlou, Software-defined networking (SDN): Layers and architecture terminology, Tech. rep. (2015).

[16] M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, A. Rindos, et al., SDSecurity: A software defined security experimental framework, in: IEEE International Conference on Communication Workshop (ICCW), IEEE, 2015, pp. 1871–1876. doi:10.1109/ICCW.2015.7247453.

[17] Snort, Network intrusion detection and prevention system, https://snort.org/, accessed: 20-02-2019.

[18] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, G. Gu, A security enforcement kernel for openflow networks, in: Workshop on Hot Topics in Software Defined Networks, ACM, 2012, pp. 121–126. doi:10.1145/2342441.2342466.

[19] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, M. Tyson, Fresco: Modular composable security services for software-defined networks, in: 20th Annual Network & Distributed System Security Symposium, NDSS, 2013.

[20] H. Kim, N. Feamster, Improving network management with software defined networking, IEEE Communications Magazine 51 (2) (2013) 114–119. doi:10.1109/MCOM.2013.6461195.

[21] S. A. Mehdi, J. Khalid, S. A. Khayam, Revisiting traffic anomaly detection using software defined networking, in: R. Sommer, D. Balzarotti, G. Maier (Eds.), Recent Advances in Intrusion Detection, Springer Berlin Heidelberg, 2011, pp. 161–180. doi:10.1007/978-3-642-23644-0_9.

[22] X. T. Phan, K. Fukuda, Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks, Journal of Information Processing 25 (2017) 182–190.

[23] T. Tang, S. A. R. Zaidi, D. McLernon, L. Mhamdi, M. Ghogho, Deep recurrent neural network for intrusion detection in sdn-based networks, in: IEEE International Conference on Network Softwarization (NetSoft), 2018. doi:10.1109/NETSOFT.2018.8460090.

[24] A. Abubakar, B. Pranggono, Machine learning based intrusion detection system for software defined networks, in: 7th International Conference on Emerging Security Technologies (EST), 2017, pp. 138–143. doi:10.1109/EST.2017.8090413.

[25] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: A comprehensive survey, Proceedings of the IEEE 103 (1) (2015) 14–76. doi:10.1109/JPROC.2014.2371999.

[26] NMAP, The network mapper - a free and open-source utility for network discovery, https://nmap.org/, accessed: 20-02-2019.

[27] Hammer, Denial of Service Python Script, https://github.com/cyweb/hammer, accessed: 20-02-2019.

[28] LOIC, Low Orbit Ion Cannon - Denial of Service Script, https://github.com/NewEraCracker/LOIC/releases, accessed: 20-02-2019.

[29] AppNeta, Tcpreplay, https://github.com/appneta/tcpreplay, accessed: 20-02-2019.

[30] S. Smiler, et al., OpenFlow cookbook, Packt Publ., 2015.

[31] Tcpreplay, bigFlows.pcap, http://tcpreplay.appneta.com/wiki/captures.html, accessed: 20-02-2019.

[32] C. V. Zhou, C. Leckie, S. Karunasekera, A survey of coordinated attacks and collaborative intrusion detection, Computers & Security 29 (1) (2010) 124–140. doi:10.1016/j.cose.2009.06.008.

[33] J. Navarro, A. Deruyver, P. Parrend, A systematic survey on multi-step attack detection, Computers & Security 76 (2018) 214–249. doi:10.1016/j.cose.2018.03.001.

Celyn Birkinshaw achieved a BSc in Physics from the University of Hertfordshire in 2007 and an MSc in Cyber Security from the University of York in 2018. His main research interests are in the field of network security, including software-defined networks, and the Internet of things.

Elpida Rouka was born in 1994 in Athens, Greece. In 2012 she received her BSc/MEng degree in Electrical and Computer Engineering from the National Technical University of Athens, specializing in Computer Science and Networks. In 2018 she received her MSc degree in Cyber Security from the University of York. She is currently employed as Security Engineer at UniSystems S.A. in Athens. Her main research interests include cryptography, network monitoring and security, malware detection and reverse engineering, software-defined networking and penetration testing.

Vassilios G. Vassilakis received his Ph.D. degree in Elec-

trical and Computer Engineering from the University of Patras, Greece in 2011. He is currently a Lecturer (Assistant Professor) in Cyber Security at the University of York, UK. He's been involved in EU, UK, and industry funded RD projects related to the design and analysis of future mobile networks and Internet technologies. His main research interests are in the areas of network security, Internet of things, next-generation wireless and mobile networks, and software-defined networks. He's published over 90 journal/conference papers. He's served as an Associate Editor in IEICE Transactions on Communications, IET Networks, and Elsevier Optical Switching Networking.