VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Enforcing a network-wide security policy using SDN

Jarrod N. Bakker

Supervisors: Dr Ian Welch, Professor Winston Seah

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering (Hons).

Abstract

Traditional firewalls are used to enforce network security policies at boundaries within a network. However, this can leave hosts vulnerable to attacks that originate from within the network they are part of. Software Defined Networking (SDN) is an architecture that separates the control plane from the data plane within switches to provide a logically centralised controller with a global network view and network programmability. Coupling those attributes with SDN's native feature to manage flows of traffic provides the necessary components to implement a SDN firewall. This report presents ACLSwitch, a SDN firewall solution that extends the capabilities of previous SDN firewalls and provides the necessary mechanisms to protect hosts from attacks that originate from within the network they are part of as well as attacks from outside the network as well.

Acknowledgments

I would like to thank my supervisors, Dr Ian Welch and Professor Winston Seah, for their guidance and support throughout the project. Their complementary knowledge bases aided me in building what one might interpret as a foundation and guide to building and testing the next generation of firewalls.

I would also like to thank my parents, Michael and Carol Bakker, for the support that they have provided throughout all of my studies. They encouraged me to push myself and test what I thought was possible to achieve.

Contents

Figures

Chapter 1

Introduction

A firewall is a configuration of hardware, software or a combination of the two, that is used to enforce a network security policy at network boundaries [1]. A network security policy defines access rights which are used by firewalls to filter traffic. Despite decreasing the chances of a host becoming infected from attacks that originate outside of the firewall boundary, hosts are still vulnerable to attacks that originate from the inside. Firewalls would need to be deployed on all network links in order to protect hosts from attacks originating from inside and outside the network. Such a solution however, does not scale up effectively due to the cost associated with purchasing a firewall and the complexity associated with managing numerous firewalls that require separate configuration. Figure 1.1 illustrates how hosts can be protected by threats from the outside but vulnerable to attacks from neighbouring hosts.

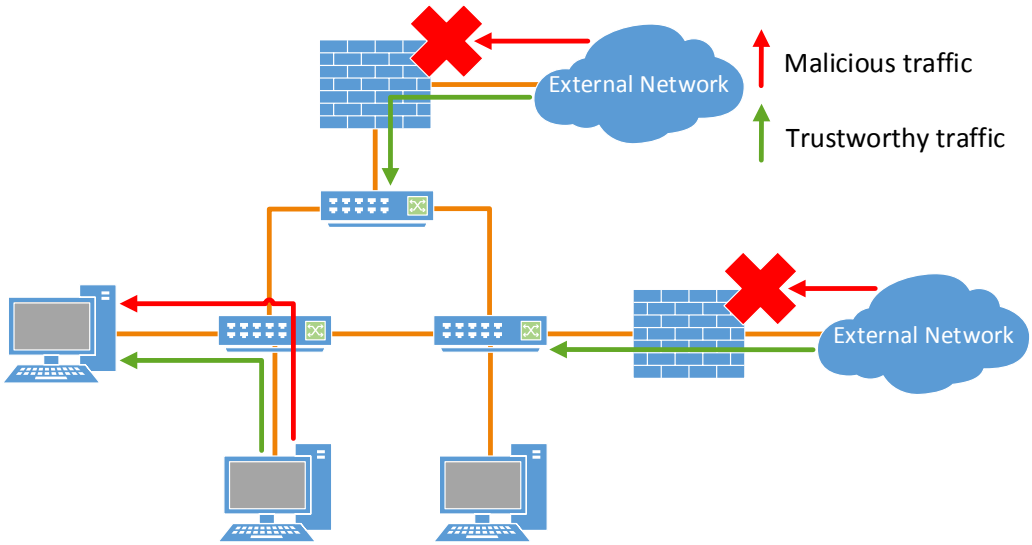

Figure 1.1: Traditional firewall placement results in hosts being compromised by their neighbours.

1.1 Proposed Solution

SDN is an emerging technology that gives a logically centralised controller a global view of a network topology and provides programmability through the separation of the control plane from the data plane [2, 3]. A native feature of SDN is the ability to manage flows of traffic based on source and destination IP addresses and transport layer port numbers. This similarity with firewalls makes it a feasible candidate for providing a scalable network-wide firewall solution [2].

This report presents a solution called ACLSwitch, a SDN firewall capable of protecting hosts from attacks originating from inside and outside of a network. The solution also extends the current capabilities of current SDN firewall solutions.

1.2 Major Contributions

This report presents the following major contributions within the area of SDN firewalls:

- A SDN firewall capable of enforcing both stateless and stateful policies.
- A SDN firewall capable of protecting hosts from attacks that originate from inside and outside of a network.
- A SDN firewall that provides a mechanism for configuring different security policies that can be enforced over different parts of a network topology.
- A SDN firewall that does not require the user to configure each switch separately with the appropriate flow table entries.
- An evaluation methodology for validating the correctness of a SDN firewall's behaviour.

1.3 Report Structure

This report has the following structure. Chapter 2 contains a background survey on firewalls and network security policies, SDN and its relevance to enforcing network security policies, and a selection of SDN firewall case studies. Chapter 3 covers the design choices of the solution. Chapter 4 describes how the solution was implemented, using the information provided in chapters 2 and 3. Chapter 5 presents the method used to validate the functional performance of the solution and assesses it success. Chapter 6 presents the conclusions and future work. Following these chapters are appendices which contain additional resources.

Chapter 2

Background and Literature Review

This chapter defines and explores the concepts relevant to the project by presenting a background survey. SDN firewall case studies are also included.

2.1 Firewalls

This section will provide an overview of firewalls and their operation. The limitations of firewall will also be explored along with a set of criteria that can be used to evaluate them.

2.1.1 Classifying Firewalls

A firewall is a piece of software, hardware, or a combination of the two, that has the ability to filter undesirable flows of network traffic at boundaries within a network [1]. By filtering network traffic, access to a network can be limited and controlled [4]. This functionality can be utilised to prevent malicious software from entering a network and compromising hosts. This scenario is illustrated in Figure 2.1.

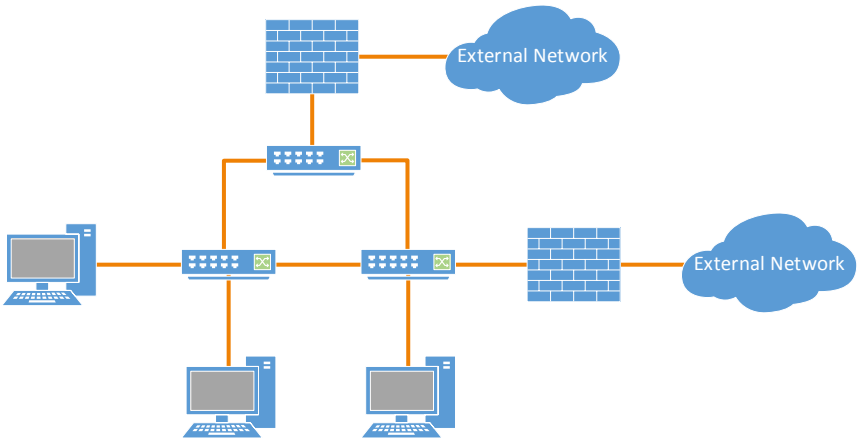

Figure 2.1: Typical placement of firewalls at network gateways.

Firewalls are by no means the only solution to making a network secure and preventing information from being exfiltrated. As a result of being designed to perform the task of filtering traffic, they are unable to detect anomalies on host machines. They instead prevent

suspicious traffic such as Trojan horses from entering a network, prevent port scanners from reaching hosts and other network attacks from compromising a system's integrity [1]. Thus it is wise to think of a firewall as a prevention mechanism and not a detection mechanism.

Firewalls can be classified using two methods. The first method classifies firewalls into three main categories: packet filters, circuit gateways and application gateways [5, 6, 4]. A packet filter drops packets based on source and destination addresses and port numbers. A circuit gateway monitors communications at the session layer of the OSI model, for example the status of a TCP handshake. An application gateway provides services for a particular application, such as email. In the email example, an application gateway can check emails for restricted data. This classification method provides a finer level of granularity than the second method described below, thus making it useful in determining the components of a firewall implementation.

The second method of classification, defined by Whitman et al. [1], classifies firewalls as stateless or stateful. A stateless firewall is one that allows or blocks a connection based on information in the packet header and ignores the state of the connection. In comparison, a stateful firewall uses a connection table to monitor the state and context of active connections. For example a packet that has been sent in response to an internal request can be allowed to pass. The main difference between stateless and stateful firewalls is that stateless firewalls do not examine the relationship between individual packets [7].

This report will primarily use the terms *stateless* and *stateful* to categorise firewall implementations. When more granularity is required (such as when describing components) the first classification method will be used for augmentation.

A firewall has the ability to inspect packets so that malicious traffic can be filtered from legitimate traffic. This feature makes them invaluable as a first-line defence mechanism for enterprise networks. They can exist in hardware or software and can either perform stateless or stateful packet inspection.

2.1.2 Network Security Policy

Firewalls contain a sequence of ordered rules that specify which traffic is allowed into a network [8]. The set of rules enforces a network security policy (or simply a security policy), which is defined as a high-level statement describing a system's protection strategy [5]. The purpose of a policy is to mitigate the risk of a network's resources being compromised.

$$< \textit{predicate} > \rightarrow < \textit{decision} >$$

Figure 2.2: High level view of a firewall rule, reproduced from [8].

Figure 2.2 shows that from a high level, the rules within a firewall exhibit a mapping between a predicate and a decision. The predicate evaluates whether or not the packet is part of a recognisable piece of traffic and the decision states what action should be performed on the packet. When a packet is received by a firewall the predicate is evaluated i.e. *does the firewall recognise the flow?* Based on the output of the predicate, the decision is executed. Firewalls typically provide two decisions, *allow* the traffic or *block/drop* the traffic. However, it is not uncommon for firewalls to provide a third decision, *forward*, which can be used to perform network address translation or port forwarding. The software firewall `iptables` provides this functionality.

Security policies can be restrictive or permissive [7]. The default behaviour of restrictive policies is to drop traffic, therefore rules are used to specify what traffic is allowed. In con-

trast, the default behaviour of restrictive policies is to allow traffic, therefore rules are used to specify what traffic should be dropped. A hybrid approach would allow rich policies to be specified, where restrictive policies dictate who can contact the network and permissive policies fine tune what traffic in particular should be dropped.

2.1.3 Expressing Policies

Security policies can be expressed in different ways. The method of expression used can affect the behaviour of a system as well as the level of granularity of policies. Two methods of expressing policies in the context of firewalls will be explored: access control lists (ACLs) and role-base access control (RBAC).

ACLs express polices by associating a user with a set of approved access decisions. This method is commonly used by operating systems to restrict access decisions (such as read, write and execute) to resources like as files and directories. This method works best when the policy is configured from a central location and where protection is data-oriented. However it does not scale to environments with large populations that change frequently [5]. Firewalls have adapted this method by treating flows of traffic identified by IP addresses and port numbers as the user and the firewall's behaviour (e.g. drop) as the access decision. The adaptation is unsurprising as ACLs follow the predicate-decision mapping illustrated in Figure 2.2.

RBAC is a policy model where access decisions are not based on the identity of a user but on the function they perform within a system [5]. As a result, access decisions for a group of users that share a common function can be managed consistently. Unlike ACLs, RBAC enables the specification of both coarse-grained and fine-grained policies. Using a university as an example, all students that attend the university may be assigned a role that grants access to resources in a common area such a library; this would be a coarse-grained policy. Within the same university environment, a group of students from a faculty X studying a particular major Y may have a role assigned to them which grants access to specialised resources; this would be a fine-grained policy. Fine-grained policies such as the one in the previous example allow segmented networks (such as networks with a demilitarized zone or DMZ) to be configured with more control.

2.1.4 Software and Hardware Implementations

Firewalls can be implemented using both software and hardware. The choice to use software, hardware or both, can effect the performance and capabilities of the solution. It is important that the trade-off between performance and capabilities be understood.

Hardware firewalls are typically deployed in the form of a middlebox. By using application-specific integrated circuits (ASICs), traffic can be quickly evaluated to determine the enforcement decision. Despite being designed for high performance filtering, hardware firewalls suffer from the following disadvantages stated by Collings and Liu [7]:

1. They are often expensive in cost.

2. Maintenance is associated with complicated, vendor-specific instructions.

3. Interoperability between hardware firewalls made by different vendors can not always be achieved.

4. Failures can result in replacing and reconfiguring multiple firewalls in order to ensure a consistent policy across a network.

Software firewalls use regular computing hardware (i.e. not ASICs) to provide the functionality of a firewall. The lack of ASICs allows software implementations to make more intelligent decisions on filtering but at the cost of performance. As software firewalls can be run on standard computers, they have access to more computing resources and memory. This is especially true in the case of products developed by Linewize such as Surfwize[1] and MyLinewize[2] which are filtering services that are hosted in the cloud. Another example of a software firewall is `iptables`, which is present on most machines that have a Linux kernel.

2.1.5 Limitations of Traditional Firewalls

Despite being suitable for filtering undesirable flows of traffic, firewalls in general suffer from some limitations. These limitations can result in a network becoming vulnerable to attack.

Cheswick et al. state that firewalls typically perform filtering at a specific layer of the protocol stack. [4]. As a result, anything malicious that is sitting at different layers will be missed. For instance, firewalls that filter traffic based on data up until the transport layer will be unable to detect issues at the application layer. The solution to this issue is to filter traffic by observing the higher layers. However this can slow down the filtering process as more processing time is required as you move higher up the protocol stack.

Another limitation of firewalls stems from how they are deployed. As firewalls are traditionally deployed at network boundaries, those tend to be the only places where traffic is filtered. As a result, a network becomes vulnerable to attacks that originate from an internal host. As hosts that are within the same network share an implicit trust relationship, all traffic sent between hosts internally is assumed to be trustworthy. One solution would be to place a firewall between each link within the network so that all traffic could be filtered. This solution does not scale however because of management, complexity and monetary cost.

Traditional firewalls are also vulnerable to IP fragmentation attacks. In an IP fragmentation attack, packets carrying malicious payloads utilise the IP fragment feature of IPv4 or IPv6 to split data from layers 4 to 7 into multiple packets. The fragments can then be reassembled at the target where the malicious payload is then executed. As the firewall does not have the packet in its complete form, it will be unable to filter the traffic appropriately which results in its delivery to the target [9].

2.1.6 Evaluating the Functional Performance of a Firewall

Cheswick et al. formally define three design goals that firewalls should follow [6, 4]. Their purpose is to ensure the correct operation of a firewall. The goals are:

1. All traffic, regardless of its source, must pass through the firewall when passing through the network.
2. A security policy defines what traffic is allowed to pass.
3. The firewall itself cannot be subverted.

2.2 Software Defined Networking

Software Defined Networking (SDN) is a network architecture that separates the control plane from the data plane. This separation allows the behaviour of the network to be defined

[1] `https://linewize.atlassian.net/wiki/display/PD/Content+Filtering`
[2] `https://linewize.atlassian.net/wiki/display/PD/My+Classroom`

through the behaviour of software applications [2, 3]. This section will explore SDN as well as OpenFlow, a protocol used to manage software defined networks.

2.2.1 Origin of SDN

SDN as it is known today has stemmed from the development of both *Secure Architecture for the Networked Enterprise* (SANE) [10] and Ethane [11]. Both SANE and Ethane separated the control plane from the data plane, with the purpose of defining the behaviour of the network from a logically centralised controller.

Casado et al. presented the notion that the decentralised control structure of the Internet, despite being necessary, made it insecure [10]. Traditional network architectures make it difficult to manage entire networks. Trust is distributed across mechanisms such VLANs, firewalls and Network Address Translators (NATs) thus it is not managed by a single entity. Such devices can become choke-points as they are typically deployed at network boundaries where they are unable to provide coverage over the entire network.

SANE was a proposed solution that utilised the concept of a clean slate, that is it redefined how networks should operate. The design goals of SANE were:

- Policies should be able to be expressed independent of a network topology.
- Policies should be enforced at the link layer, to prevent it from being circumvented by a lower layer in the protocol stack.
- Information on a network's topology such the location of hosts and other computing resources should be only be accessible by those with the correct privilege.
- Trust should be placed in a single trusted component that can be replicated if necessary.

Given the above design goals, it is clear that the primary focus of SANE was to provide a network security mechanism without the need for building more layers on top of the protocol stack.

Development into similar architectures was continued through the development of Ethane. Using lessons learnt from SANE, Casado et al. sought to expand upon and improve the centralised approach of managing a network [11]. To expand on the work done previously, two areas received focused.

The first area concerned the capabilities of the control plane. SANE provided a control plane that was designed to enforce security policies. This meant that switches were only able to control who could communicate with who and did not provide functionality that could be used to diagnose and fix issues. Ethane aimed to broaden its capabilities so that a wider range of management functions could be carried out yet still provide a mechanism for enforcing security policies.

The second area sought to improve the deployment capabilities of SANE. The implementation of SANE not only required changes to a network's infrastructure but to end hosts as well. Ethane was designed in such a way, that end hosts would did not need to be reconfigured and Ethane-enabled switches could be deployed alongside traditional Ethernet switches.

From the outlines of SANE and Ethane provided, it can be seen that the purpose of SDN shifted from securing a network against threats to enabling more generalised control over the network. As a result of this change, the SDN architecture facilitates the development and deployment of various networked solutions.

2.2.2 OpenFlow

OpenFlow was presented in 2008 as a continuation of the work done through SANE and Ethane. OpenFlow itself is a protocol that exploits common features seen throughout switches and routers. The protocol uses these features to manipulate the flow tables within these devices so their behaviour can be defined from a logically centralised controller [12]. Figure 2.3 illustrates an example SDN topology where an OpenFlow controller communicates with OpenFlow enabled switches using links that are physically separated from the network fabric that connects hosts.

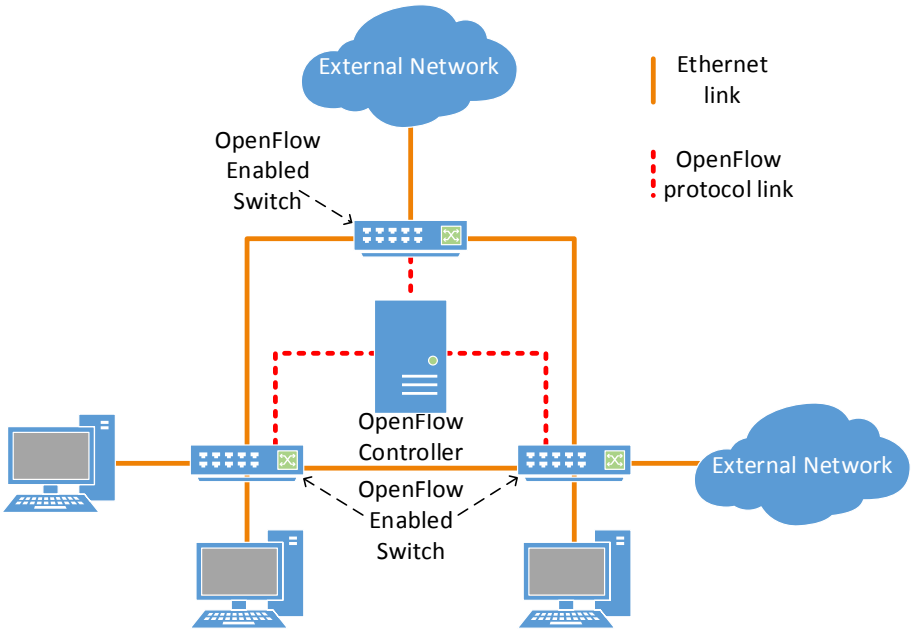

Figure 2.3: Topology illustrating how an OpenFlow enabled software defined network can be deployed.

Protocol Operation

The OpenFlow protocol operates between switches over a secure channel using SSL. It is through this channel that the data plane within an OpenFlow enabled switch is configured. The primary theatre for configuration resides in the flow table pipeline. The flow table pipeline consists of a series of flow tables that can be used to logically separate flow table entries. Flow table modification messages alter the contents of flow tables by facilitating the insertion, modification and removal of flow table entries [13, 14]. Flow table entries do not need to be removed manually either, a timeout value can be configured to facilitate automatic removal. Figure 2.4 illustrates the relationship the OpenFlow protocol forms between an OpenFlow enabled switch and an OpenFlow controller.

The OpenFlow protocol provides a means for configuring each switch separately. This means that different policies (for security, QoS or routing for example) can be enforced at different areas of a network topology. This feature embodies the programmable nature of SDN, as all behaviour can be managed from a logically centralised controller.

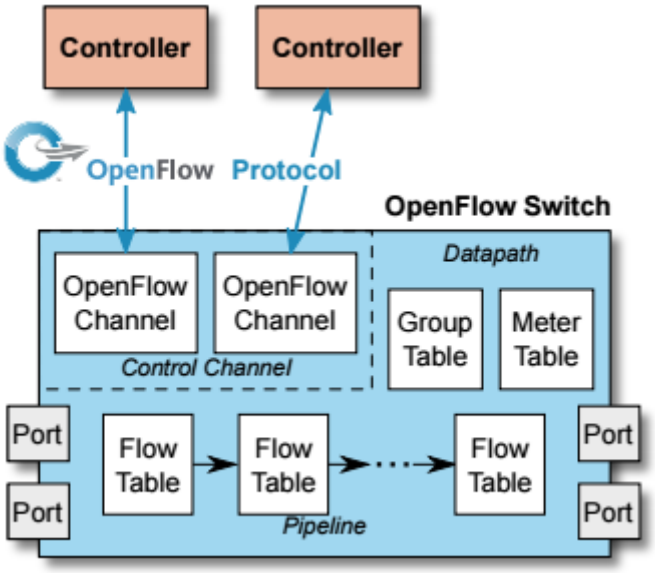

Figure 2.4: The OpenFlow protocol at the switch level [14].

As the protocol has matured since its first release, the variety of traffic that can be matched has been expanded. The 1.5.0 specification presented a match field for table entries that allows TCP flags (such as SYN, ACK and FIN) to be matched [14]. Flow table entries with this field can be used to detect the state of TCP connections. As OpenFlow also provides mechanisms for modifying the content within packet headers, these two features could be combined to reset TCP connections at the switch level.

OpenFlow Controllers

Various OpenFlow controllers facilitate the use of different versions of OpenFlow. As OpenFlow is controller agnostic, controllers can be implemented using different programming languages. This means that developers can utilise the nuances in different languages to change the behaviour of the network. Table 2.1 presents a collection of popular OpenFlow controllers along with the programming language they support and the versions of OpenFlow they support.

Controller	Language	OpenFlow compliance
Ryu	Python	1.0, 1.1, 1.2, 1.3, 1.4, 1.5 [15, 16]
Floodlight	Java	1.0, 1.3 [15, 17]
NOX	C++	1.0 [15]
POX	Python	1.0 [15]

Table 2.1: Table displaying popular OpenFlow controllers, with the language they are developed in along with the versions of OpenFlow that they fully support.

2.3 Firewalls in Software Defined Networking

SDN and OpenFlow in particular, offer mechanisms for enforcing various policies at SDN switches. As such, security policies can be enforced across an entire network topology thus challenging the traditional approach for using firewalls.

2.3.1 Description of a SDN Firewall

As stated earlier, traditional firewalls filter packets at network boundaries. Traditionally the aim is to keep threats that originate from external networks from entering the internal network and compromising an internal resource. This results in firewalls being deployed at gateways. Figure 2.1 described this architecture previously, an abstraction is provided in Figure 2.5.

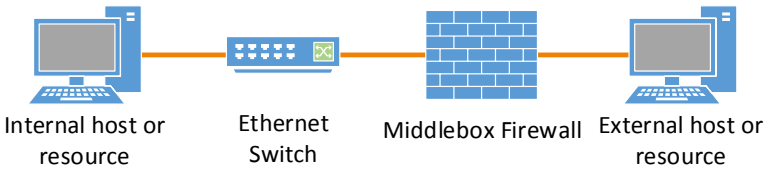

Figure 2.5: The motivation for the placement of a typical firewall is to protect resources.

In comparison, a SDN firewall consists of a SDN application that runs on SDN switches and a SDN controller. The same classification methods from before (packet filter, circuit gateway or application gateway, and stateless or stateful) can be used to classify firewalls to identify their behaviour. SDN firewalls also have the option of distributing rules either proactively or reactively. Proactive distribution will send flow table entries that represent rules to switches as soon as they become available at the controller. Reactive distribution will send a single single flow table entry to a switch only when an undesirable packet has been sent to the controller. The consequences of both methods are explored in depth in Chapter 3 Section 3.3.2. A high-level view of the SDN firewall architecture is illustrated in Figure 2.6 and an abstraction is provided in Figure 2.7.

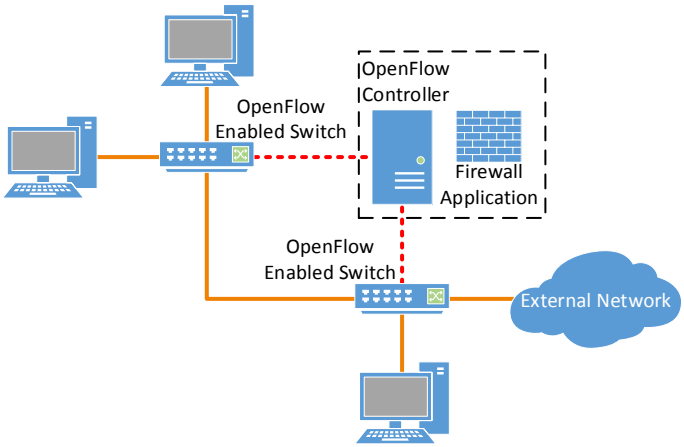

Figure 2.6: A SDN firewall sits at the SDN controller and uses OpenFlow to communicate with switches.

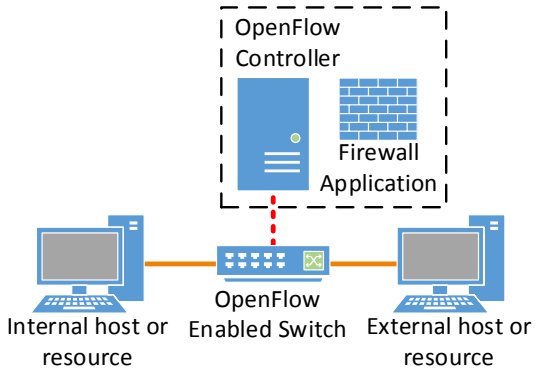

Figure 2.7: A SDN firewall abstraction that illustrates the change in how firewalls are deployed to protect resources.

2.3.2 SDN Firewall Case Studies

Prior research and development has been conducted in the area of SDN firewalls. Solutions that use OpenFlow will be explored below in order to illustrate the similarities and differences between them.

1. SDN Firewall Proposed by Suh et al.

Suh et al. proposed a stateless SDN firewall that extends the capabilities of a regular packet filter by allows users to specify rules that can filter traffic at the link layer [18]. The authors chose to enforce the security policy at the controller as they stated that such architectures enable flexible management. However they identified the issue that this can result in the controller becoming overwhelmed thus making the controller vulnerable to DoS attacks. The solution also allowed users to specify a timeout value for a flow table entry. It is important to note that once the timeout expires and the entry is removed, a user must manually create the rule again if it needs to be enforced at a later date.

2. Ryu rest_firewall.py

Ryu provides an example SDN firewall that allows users to specify stateless policies via a REST API [19]. Rules can be configured to drop the following types of traffic: ARP, IPv4, IPv6, TCP, UDP, ICMP and ICMPv6. The firewall also supports policies on traffic marked with VLAN tags. Policy configurations can only be made on a per-switch basis, therefore if a user wishes to distribute the same rule to multiple switches then separate calls to the REST API must be made. This extra layer of complexity can result in inconsistent security policies.

3. SDN Firewall Proposed by Collings and Liu

Collings and Liu proposed a stateful SDN firewall intended to only be deployed at network gateways [7]. The kind of stateful policies used and their method of enforcement is not stated. Their chosen controller, POX, only supports OpenFlow 1.0 which means that TCP flag matching cannot be performed via a switch's flow table. Therefore it could be assumed that an external application is used to analyse incoming traffic. However such a solution would increase the latency of traffic passing through the switch. Another limitation of the

solution is that it is intended to be deployed at network gateways. This feature makes the solution similar to the traditional method of firewall deployment.

4. Valve

Valve extends the functionality of the Ryu controller specifically for OpenFlow 1.3 [20]. Valve provides support for configuring stateless ACLs based on IP addresses and port numbers. Regardless of the fact that Valve is not designed to be a SDN firewall, it fits the criteria for a stateless packet filter. Similarly to rest_firewall.py provided with Ryu, Valve provides VLAN support.

5. SDN Policy Expression Model Proposed by Sasaki et al.

Sasaki et al. proposed an alternative method of specifying stateless policies in OpenFlow which claimed to reduce the total number of rules. [21]. As with Valve, this solution was not developed as a firewall. However it was applied to the context of security policy enforcement and it demonstrates an alternative design for SDN firewalls. Instead of using ACLs, RBAC is used which to map a user assignment (UA) to a permission assignment (PAs). A user assignment associates a role with an IP address (the user), whereas a permission assignment associates a role with an IP address (a resource such as a server) and an action (such as *allow*). The number of rules required when using ACLs and RBAC (assuming that clients and servers both have one role) respectively, can be described as follows:

$$\begin{aligned} \text{ACL size} &= \text{no. of source IP addresses} \times \text{no. of destination IP addresses} \\ \text{RBAC size} &= \text{no. of UA} + \text{no. of PA} \\ &= \text{no. of source IP addressess} + \text{no. of destination IP addresses} \,. \end{aligned}$$

Given the above equations and the provided assumption, RBAC can be represented in a smaller table. However RBAC is best utilised when users can be clearly mapped to functions.

Case Study Summary

The case studies above reveal that SDN firewalls that have been currently developed can be labeled as stateless packet filters. The lack of stateful SDN firewalls has been explained by Hu et al. [22]. They state that there are challenges surrounding the support of stateful packet inspection with OpenFlow as access to packet-level information is limited. A possible solution to this problem would be to use an external application to analyse TCP header fields and feed the information back into OpenFlow. This solution requires extra layers of complexity and goes against the early SDN philosophy that policies should be enforced at the link layer. As stated earlier however, OpenFlow 1.5 introduces functionality to allow TCP flags to be matched at switches.

It is also clear that SDN and OpenFlow make it possible to implement different methods of specifying polices. Such methods change the way that policies can be allocated to users or possibly even switches. Table 2.2 summarises the given cases studies in terms of the OpenFlow controller used, the version of OpenFlow used, how the firewall can be classified, the kind of policies supported and rule distribution type (explained in more detail in Chapter 3 Section 3.3.2).

Case study	Controller	OF version	Classifier 1	Classifier 2	Policy	Rule dist.
Suh et al.	POX	1.0	Stateless	Packet filter	Restrictive	Reactive
rest_firewall.py	Ryu	1.0, 1.2, 1.3	Stateless	Packet filter	Restrictive	Proactive
Collings and Liu	POX	1.0	Stateful	Packet filter	Restrictive	Reactive
Valve	Ryu	1.3	Stateless	Packet filter	Restrictive	Proactive
Sasaki et al.	-	1.1	Stateless	Packet filter	-	Proactive

Table 2.2: Summary of SDN firewall case studies.

2.4 Chapter Summary

Firewalls are used to filter undesirable flows of traffic to prevent a system from being compromised. Traditionally they are deployed at network boundaries which can leave the machines behind the firewall vulnerable to attacks from the inside but they can also be used to segment networks and create a DMZ. SDN is an architecture that has the inherent feature of performing actions on flows of traffic. As SDN provides a global view of a network topology and allows policies to be enforced without the need for more layers, it is a suitable architecture for providing a solution that can enforce a security policy that can protect hosts from attacks originating from inside and outside the network.

Chapter 3

Solution Design

This chapter explores and discusses the design choices for a SDN firewall that is centred around the features and capabilities of OpenFlow 1.3. This chapter represents the engineering design process that was carried out: after identifying the problem and conducting a background survey, possible design approaches were determined and tested. The lessons learnt from tests were used to improve the design before subjecting it to more tests.

3.1 Design Goals

In this section the design goals for both the stateless and stateful firewall implementations will be listed. The design goals were inspired by the case studies provided in the previous chapter as well as the features of the OpenFlow protocol. Note that the design goals below will be used in conjunction with the firewall design goals established by Cheswick et al. to validate the functional performance of ACLSwitch in Chapter 5.

3.1.1 Stateless SDN Firewall

The design goals for the stateless SDN firewall are as follows:

1. A security policy should be enforced at the switch as to avoid overloading the controller.
2. An ACL rule should block traffic that it explicitly states.
3. An ACL rule should not block traffic that it is not supposed to.
4. Once an ACL rule has been added to the controller, it should be propagated to connected switches.
5. Once an ACL rule has been removed from the controller, the action should be reflected in connected switches.
6. Different policies should be able to be applied across different parts of a network.
7. The solution should be configurable on application start-up.
8. The solution should be configurable via an API during runtime.
9. The solution should not rely on external applications (where possible) to enforce stateless policies in order to reduce the complexity of the system.

3.1.2 Stateful SDN Firewall

The design goals for the stateful SDN firewall are as follows:

1. The stateful implementation should be able to enforce stateless policies in much the same way as the stateless implementation.

2. The solution should not rely on external applications (where possible) to enforce stateful policies in order to reduce the complexity of the system.

3. Time-based stateful rules should be scheduled on a 24-hour cycle.

3.2 Using OpenFlow 1.3 as a Platform for a SDN Firewall

OpenFlow was introduced in the previous chapter as a protocol for managing the behaviour of OpenFlow compliant SDN switches. A version of OpenFlow can be used by a controller to communicate a mapping between a behaviour and a flow of traffic to a switch. This ability to map a behaviour to a flow of traffic is as innate to OpenFlow as it is a Firewall.

3.2.1 Choice of OpenFlow Version and Controller

Chapter 2 Section 2.2.2 illustrated popular OpenFlow controllers along with the versions of OpenFlow that they fully support. Given the information presented in there, it is also worthwhile to note the versions of OpenFlow supported by the current version of Open vSwitch[1]. As of the 9th of May, 2015, the latest version of Open vSwitch (v2.3) fully supported OpenFlow versions 1.0 to 1.3. OpenFlow versions 1.4 and 1.5 were not fully supported as there were one or more missing features [23]. This restricted the choice of controllers to Ryu and Floodlight. Ryu was chosen in the end due to past experience with the controller.

3.2.2 Building an Application for OpenFlow 1.3

The Open Networking Foundation abstracts SDN with OpenFlow into three layers. The infrastructure layer holds what can be classically defined as the network, these are the nodes/switches/routers which forward traffic towards its destination. The control layer sits on top of the infrastructure layer and uses OpenFlow to configure the behaviour of the nodes. The application layer sits on top of the control layer and presents a user with API which can be used to interact with services in the control layer [3]. This is illustrated in Figure 3.1.

[1] An open-source virtual switch commonly used with the Mininet environment http://openvswitch.org/.

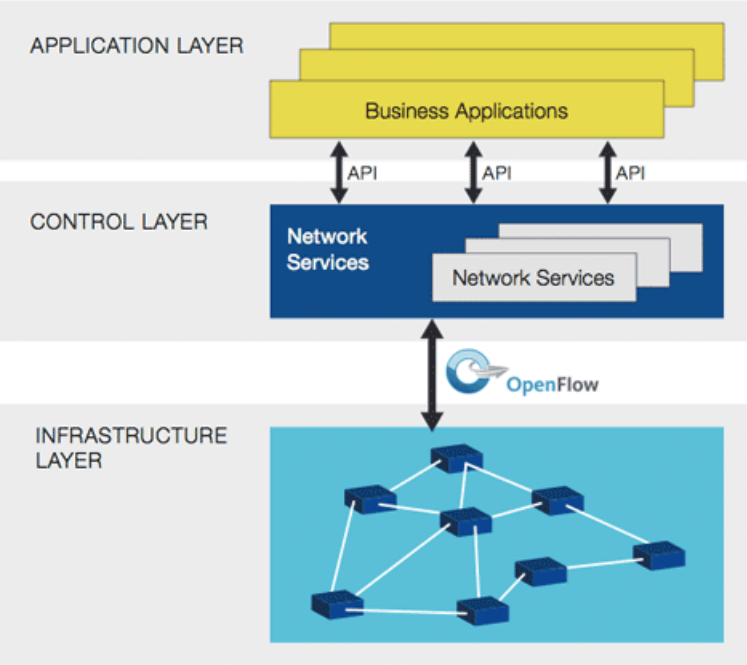

Figure 3.1: A three layer abstraction of a SDN network using OpenFlow [3].

Given the layers presented in Figure 3.1, a SDN firewall solution will need to do the following:

1. Enforce a network security policy at the infrastructure layer.

2. Operate as a service at the control layer by storing network security policies.

3. Provide a user with an API from which to change the state of the firewall service at the application layer.

3.2.3 Relevant Features of OpenFlow 1.3

Features of OpenFlow 1.3 that are relevant to the design of a SDN firewall will be explored below.

Flow Table Pipeline

OpenFlow 1.3 allows each switch to contain multiple flow tables. The tables are implemented as a pipeline where a packet is first matched against entries within table 0 (see Figure 3.2). A flow table entry may direct a packet to another table for further processing but only to a table with a higher identifier. For instance, a flow table entry in table 2 can direct packets to tables 3 or 4, but not to tables 1 or 0.

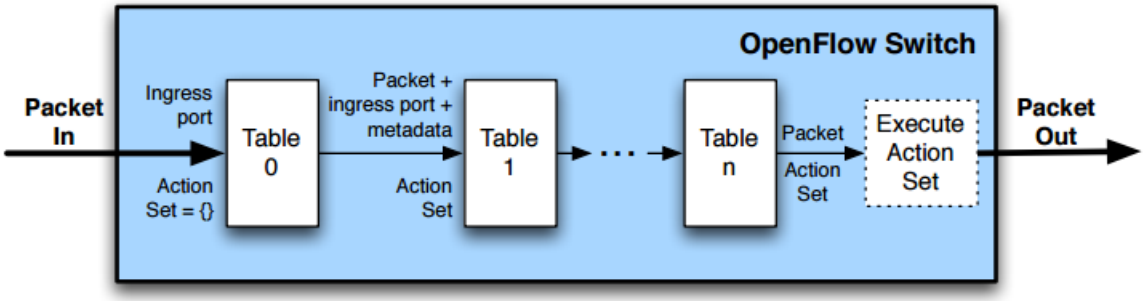

Figure 3.2: The OpenFlow flow table pipeline as of version 1.3.5 [13].

Each flow table must also contain a table-miss flow entry. This is an entry that matches all header fields and it is commonly placed at the bottom of a flow table (by setting its priority to 0) to direct matches to the next table in the pipeline, to the controller for processing or to drop matched packets. The OpenFlow 1.3.5 specification states that a flow table does not have a table-miss flow entry installed by default and it must be installed by the controller. It behaves in much the same way as any other flow table entry: it is created by the controller, it may expire or it may be manually removed by the controller [13].

Flow Matching and Flow Table Entries

Arguably the most relevant feature of OpenFlow which lends itself to performing the role of a firewall is flow matching. When a packet enters an OpenFlow switch, the fields within its headers are compared against entries within the switch's flow table. Matches are evaluated from the top of a table, downwards. A packet follows the action associated with the first flow table entry to match all fields within a packet's headers. If a flow table entry was only matching a source IPv4 address of 10.0.0.1, then any packets with that source IPv4 address will match that entry. As a result of not specifying other fields such as a destination IPv4 address, the example flow table entry would have the IPv4 destination address field implicitly set to *all* or *any*.

Flow table entries can be installed in a flow table at any time during a switch's operation. They can be installed as a result of a request being made from a switch to the controller or be installed pre-emptively to avoid packet buffering during the exchange mentioned before. Therefore flow table entries not only represent active connections but potential connections as well. This is unlike a stateful firewall that uses a connection table to only monitor active connections.

Flow Table Entry Creation and Removal

It is a given that the OpenFlow protocol supports the creation of flow table entries. In the context of enforcing a network security policy, this allows a policy to be applied at OpenFlow enabled switches through the creation and distribution of flow table entries. What is more interesting however is the removal of flow table entries. OpenFlow 1.3 supports two methods of removal, each represented by a different flow table modification message: `OFPFC_DELETE` and `OFPFC_DELETE_STRICT`. When the `OFPFC_DELETE` message is used, all flow table entries matching the fields in the message are deleted. When the `OFPFC_DELETE_STRICT` message is used, the priority of a flow along with its fields are matched for removal.

$e_1 = \{ip\_src = 10.0.0.1, ip\_dst = 10.0.0.2, priority = 2\}$
$e_2 = \{ip\_src = 10.0.0.1, ip\_dst = 10.0.0.2, tp\_proto = tcp, port\_dst = 80, priority = 4\}$

Figure 3.3: Two flow table entries with similar source and destination IP addresses.

Using the entries illustrated in Figure 3.3 as examples, if an `OFPFC_DELETE` flow modification message was issued matching $ip\_src = 10.0.0.1$ and $ip\_dst = 10.0.0.2$ then both e_1 and e_2 would be removed. However if an `OFPFC_DELETE_STRICT` flow modification message was issued matching $ip\_src = 10.0.0.1$, $ip\_dst = 10.0.0.2$ and $priority = 2$ only e_1 will be removed. This distinction may appear subtle at first but in the context of a network security policy the use of `OFPFC_DELETE` can lead to the accidental removal of flow table entries that should otherwise be left in a switch's flow table. Such a situation would put a network into a vulnerable state.

By default, a switch will not notify the controller when a flow table entry is removed by any means. However if an entry has the `OFPFF_SEND_FLOW_REM` flag set then a flow removed message will be sent to the controller upon removal of the flow.

Flow Table Entry Timeout

Flow table entries exhibit two distinct timeout attributes: `idle_timeout` and `hard_timeout`. As the name suggests, the `idle_timeout` attribute indicates when an entry should be removed if no matches are made; this is a period of time expressed in seconds. The `hard_timeout` attribute simply indicates the maximum lifetime of an entry in seconds. If both attributes are set to zero then an entry will remain in a switch indefinitely unless removed manually.

An alternative approach for removing flow table entries after a period of time would involve using the controller to maintain a schedule and manually remove flow table entries from the switch when required. This approach would required the controller to deviate from other tasks that might be more important such as modifying the network security policy. It is recommended that the `hard_timeout` attribute be used however, as the behaviour that would result from using the former approach has already been implemented natively at the switch.

Handling IP Fragments

Chapter 2 Section 2.1.5 described how IP fragments can subvert a traditional firewall. OpenFlow 1.3 provides a mechanism for handling IP fragments at switches through the definition of three explicit behaviours. These are represented as flags within a `ofp_switch_config` message. If the `OFPC_FRAG_NORMAL` flag is set then the switch will treat IP fragments as non-fragmented traffic and attempt to match it against flow table entries. If the `OFPC_FRAG_DROP` flag is set then the switch will drop any packets with IP fragments. If the `OFPC_FRAG_REASM` flag is set then the switch will reassemble IP fragments and then match the reassembled packet against flow table entries.

This functionality was not included in the implementation of ACLSwitch however, as Scapy (a packet manipulation module for Python that was used for testing and validation) can neither create nor send IP fragments. Due to time constraints, other tools for validating this functionality were not investigated. Greater focus was placed on implementing and validating the core stateless implementation of ACLSwitch as the stateful implementation of ACLSwitch extended the stateless implementation.

3.3 Stateless SDN Firewall Design

This section will explore the design of a stateless SDN firewall using OpenFlow 1.3.

3.3.1 Components

A stateless SDN firewall solution can be logically separated into the following components: an API, ACL modification and flow table entry dispatch and removal. For the purposes of the project, the solution should also be able be able to fulfil the task of a layer two switch so that packets may be forwarded towards their destination. From a high-level, the policy enforcement architecture can be expressed using the diagram in Figure 3.4.

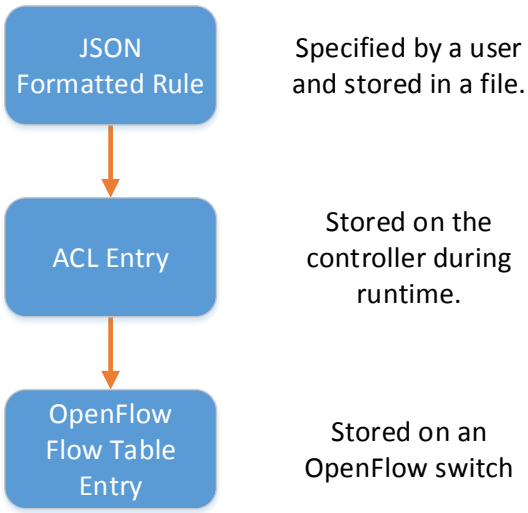

Figure 3.4: Diagram illustrating how rules specified at a high-level are enforced in the proposed SDN firewall solution.

The API must allow rules to be created not only on application start-up but during runtime as well. An API can be accessed by using Ryu's web server function. Ryu uses Python's Web Server Gateway Interface (WSGI) to allow other systems to interact with applications [24]. This allows developers to design an API that uses the REST philosophy. Instructions on utilising this can be found at
`https://osrg.github.io/ryu-book/en/html/rest_api.html`.

3.3.2 Policy Enforcement

As stated in Chapter 2, a firewall enforces a network security policy by filtering network traffic. Such policies are enforced at the firewall, regardless of its configuration (hardware, software or both). Security policies are not generally static either and are subject to change in order to meet the current environment of a system. To address this, a stateless SDN firewall must be able to add and remove rules such that different flows of traffic can be filtered at different times.

Enforcing a Stateless Policy using OpenFlow

As stated earlier OpenFlow allows actions to be applied to flows of traffic. A flow can be matched using an OpenFlow `ofp_match` structure (`OFPMatch` in Ryu) and the set of actions to be applied is wrapped inside an OpenFlow `ofp_action_header` structure (`OFPInstructionActions` in Ryu). The match fields relevant to a stateless SDN firewall are shown in Table 3.1. The complete list of header match fields can be found in Section 7.2.3.8 of the OpenFlow 1.3.5 specification [13].

OXM Header Match Field	Packet header field
OXM_OF_ETH_TYPE	Specifies the network layer protocol
OXM_OF_IPV4_SRC	IPv4 source address
OXM_OF_IPV4_DST	IPv4 destination address
OXM_OF_IPV6_SRC	IPv6 source address
OXM_OF_IPV6_DST	IPv6 destination address
OXM_OF_IP_PROTO	Specifies the transport layer protocol
OXM_OF_TCP_SRC	TCP source port number
OXM_OF_TCP_DST	TCP destination port number
OXM_OF_UDP_SRC	UDP source port number
OXM_OF_UDP_DST	UDP destination port number

Table 3.1: Table describing the meaning of relevant Header Match Fields.

Dispatching Flow Entries

As the control and data planes are logically separated in a SDN environment, a network security policy could be initially enforced at either a SDN switch or at the controller. When initially enforced at the switch, the policy enforcement mechanism can be classed as proactive. When initially enforced at the controller, the policy enforcement mechanism can be classed as reactive.

In a SDN firewall which uses reactive enforcement, the controller will only send flow table entries to a switch when traffic cannot be matched within the switch's flow table. When a controller receives a packet from a switch it must evaluate whether or not it should be dropped by checking the packet against the ACL. The flow table entry generated as a result of the evaluation can then forwarded to the requesting switch. In a SDN firewall which uses proactive enforcement however, flow table entries which block traffic are pre-emptively sent to connected OpenFlow switches as the rules are made available. A consequence of this approach is that undesirable traffic will not be forwarded to the controller when it first enters a network. This can prevent the controller from being flooded. Figures 3.5 and 3.6 illustrate the steps carried out in reactive and proactive policy enforcement respectively. Note that h1 refers to host 1 and h2 refers to host 2.

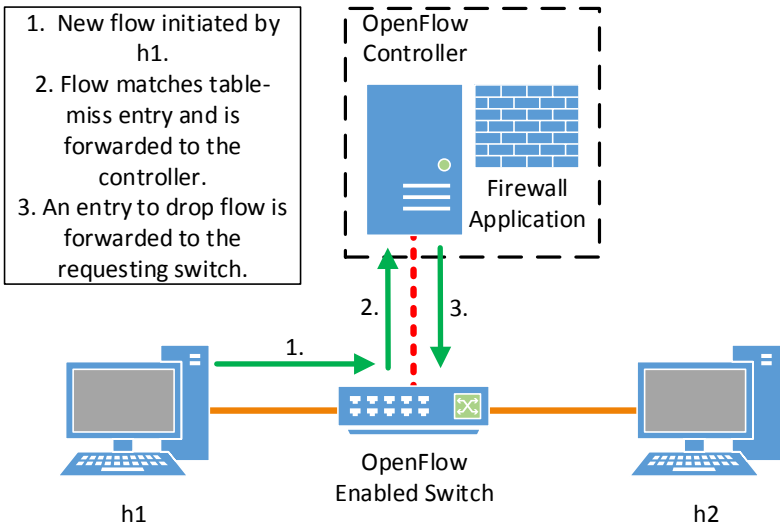

Figure 3.5: Reactive policy enforcement operation.

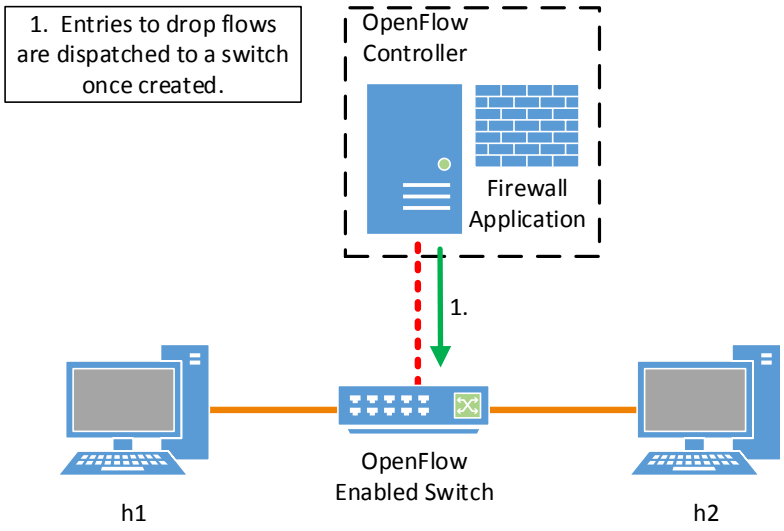

Figure 3.6: Proactive policy enforcement operation.

By pre-emptively sending rules to connected switches, flow tables may be filled with entries that may never received matches. However, this scheme can reduce the communication overhead that results from having to process traffic at the controller and buffer it at the switch. In comparison, reactive policy enforcement can result in undesirable traffic entering a network. The scenario below describes this.

Consider a SDN firewall that uses reactive policy enforcement that has been configured to block TCP traffic arriving on port 80 from h1. Upon network start-up, the only flow table entry in each switch's flow table will be a table-miss flow entry. If a switch then receives a packet from h1 it will be forwarded to the controller for evaluation. If the received packet

does not have a TCP source port of 80 then the controller will inform the switch that traffic matching the Ethernet address of the packet should be forwarded towards its destination. In any other situation this action would be acceptable, however if the switch were to receive a packet from host with a TCP source port of 80 then it would be allowed through instead of being blocked. The switch had not been told to block the traffic because the undesirable traffic had not been received first. As such behaviour is undesirable, proactive policy enforcement is more preferable over reactive policy enforcement.

The above problem could be avoided by installing flow matches for each flow received by a switch. However, this compromise is expensive and is not scalable. Therefore it can be concluded that proactive enforcement is more effective. A hybrid approach could also be implemented but it would be difficult to guarantee that the policy could be successfully enforced due to the issues regarding reactive enforcement.

Restrictive and Permissive Policies

A restrictive SDN firewall would have a default behaviour of blocking all traffic and would only allow flows of traffic through if specified. This design approach would result in a SDN firewall that could clearly define what traffic is allowed but not what traffic should be blocked. For example, one rule would be needed to allow UDP traffic to and from a host with an IPv4 address of 8.8.8.8. However, a security policy that only required UDP traffic to and from a host with an IPv4 address of 8.8.8.8 to be blocked would result in numerous rules defining all other traffic that is allowed.

A permissive SDN firewall would have a default behaviour of allowing all traffic through and would only block flows of traffic if specified. This design approach would result in a SDN firewall that could clearly define what traffic should be blocked but not what traffic should be allowed. This optimistic approach to filtering traffic is not suitable when more flows of traffic need to blocked than allowed through.

3.3.3 Policy Domains

A SDN controller's knowledge of a network topology allows it to not only enforce a single security policy across a network but also enforce smaller security policies or *policy domains* across different parts of the network as needed. A switch's purpose can define the policy domain it is assigned and by extension the set of rules that it will enforce. The advantage of using SDN is the notion of a centralised controller. This feature can be utilised to ensure that policy domains do not conflict with one another and avoid the creation of redundant rules. Figure 3.7 provides an illustration of this functionality.

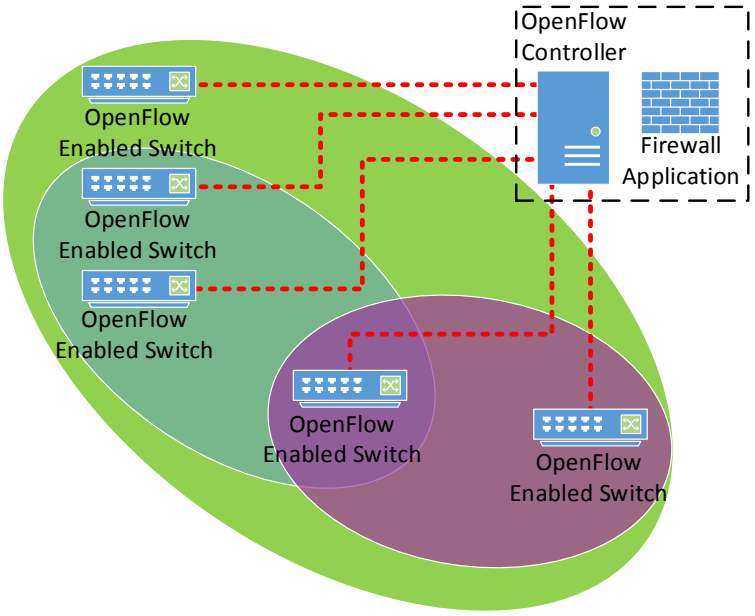

Figure 3.7: OpenFlow switches can be assigned different set rules to enforce different policies (represented by different coloured ellipses).

As policy domains could be assigned to switches based on their purpose (i.e. the function that it performs), the best suited model for expressing policy domains is RBAC. In this context, a switch is treated as a user and can be assigned different policy domains depending on what traffic needs to be filtered. As a network topology is subject to change, the policies assigned to switches are not assumed to be permanent.

3.4 Stateful SDN Firewall Design

This section will explore the design of a stateful SDN firewall using OpenFlow 1.3. As described in Chapter 2, a stateful firewall enforces policy by using a packet's state and context. With this in mind, a stateful firewall must also able to enforce stateless policies and utilise the mechanisms from such a configuration. Therefore the design of a stateful SDN firewall will build upon the design of a stateless SDN firewall. This means that it must have a mechanism for configuring stateless policies as well as supporting an API, ACL modification, flow table entry dispatch and removal, and policy domains.

3.4.1 Enforcing a Stateful Policy using OpenFlow

A typical stateful firewall uses TCP flags to restrict traffic that has not been initiated from an internal host. OpenFlow 1.3 however does not support the matching of TCP flags. The earliest version of OpenFlow to support the matching of TCP flags is 1.5[2]. Another issue regarding this concerns Open vSwitch's compatibility with OpenFlow 1.5. As mentioned before Open vSwitch only partially supports 1.5, with no indication given as to what features are missing. As this was the case, it was deemed to be risky to develop a stateful SDN firewall using OpenFlow 1.5 as the behaviour of the solution would not be guaranteed.

[2]The specification for OpenFlow 1.5.1 can be found at `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf`

A possible solution that would enable the inspection of TCP flags would involve the use of a separate program to disassemble packets. However the overhead incurred from sending the traffic to another machine to be inspected and waiting for a response would be greater than inspecting the traffic at a software switch itself. This would also violate the second design goal for the stateful SDN firewall.

Another area of packet context that can be enforced using stateful policies is the arrival time of a packet. Such functionality can be seen in products developed by Linewize, namely MyLinewize and Surfwize, that were describe in Chapter 2. To facilitate time bound rules, the `hard_timeout` field for flow table entries should be used. The limiting factor of this field is that the value is stored as an unsigned 16-bit integer which restricts the maximum timeout value to 65535 seconds or approximately 18.2 hours.

3.4.2 Scheduling and Dispatching Rules

Time bound rules need to be handled by a scheduling mechanism. Such a mechanism will need to ensure that rules are inserted in order correctly by using a data structure such as a queue. Queue insertion will need to be aware of the fact that rules will be scheduled on a 24 hour cycle. Consequently, the scheduler will have no concept of dates, only hours and minutes. A trivial case for scheduling a rule would be when the queue is empty and the time of a new rule comes after the current time. In this case we can say that the current time is less than time of the new rule i.e. $t_{cur} < t_{new}$. A less trivial case would be as follows: the head of queue contains a rule scheduled for 0030, the current time is 2300 and a new rule needs to be scheduled for 0000. Unlike the previous example, t_{cur} in this case is greater than t_{new}. To handle such situations, the scheduler will need to consider: the current time, the time of the rule scheduled at the head of the queue and the time of the new rule.

To dispatch a rule at the appropriate time, a mechanism has be to used to wait till that time. One option was to implement a signal handler for the **SIGALRM** signal. With this method, the difference in time (t_{wait}, in seconds) is calculated between the current time and the time of the next rule to be schedules. The **SIGALRM** signal can then be set to expire after t_{wait} seconds. Once expired the rule can be dispatched to the appropriate switches. This method cannot be used however as signal handlers require applications to be multi-threaded and applications in Ryu are not. When a signal expires the following assertion is broken ("AssertionError: Cannot switch to MAINLOOP from MAINLOOP") and the application crashes.

Another option that is fully supported and used by Ryu involves the use of green threads. A green thread (or Eventlet) gives the illusion of a program being multi-threaded when the environment being used does not support true multi-threading. Only a single green thread can run at any one time but they can be scheduled to sleep for periods of time. This behaviour makes green threads a feasible solution in the context of scheduling actions for specific moment. Another advantage of green threads is that data structures that are shared do not need locks to maintain consistency. The reason for this is that green threads cooperatively yield to one another instead of being pre-empted. Therefore access to a data structure can only be granted if a yield is called explicitly [25].

3.5 User Interface

To satisfy the requirements of design goal four for the stateless SDN firewall solution, a user interface was necessary. A user interface satisfies the requirements by allowing a user to change the ACL configuration during application runtime. Two design options will be explored in this section: a multi-threading approach and a RESTful API approach. It should

be noted that the purpose of the project was not to design a interface, however it is necessary for a user to configure a firewall.

3.5.1 Multi-threaded Ryu Interface

In the multi-threading approach, a user interacts directly with the SDN application. They are presented with an interface (text-based for instance) and all interactions are sent directly to the application. Such an interface could be implemented by a loop that reads input from a user's terminal screen. However the presence of the loop means that algorithms which deal with messages sent by switch will never run. Therefore the interface could be allocated it own thread to operate within. However such a solution does not behave as expected because Ryu only permits single-threaded processes to run. As a result a multi-threaded solution behaves similarly to a solution where the interface and SDN flow table request handler are in the same thread.

3.5.2 REST API

In the RESTful approach the user interface and SDN application are logically separated. From a separate program, a user makes API calls via a network connection to change the ACL configuration. Unlike multi-threading, Ryu supports REST linkage by using WSGI from Python. By registering a WSGI-compatible web server class, an external application can interact with a SDN application run on a Ryu controller. As the RESTful approach is supported by Ryu, it was the recommended solution. It should be noted that this mechanism does not have any security measures implemented on top of it. As a result it is possible for anyone to utilise such an interface if they know the correct HTTP verbs and URL.

3.6 Chapter Summary

OpenFlow 1.3 is a suitable protocol for use in a SDN firewall. It is fully supported by the latest versions of Open vSwitch and Ryu and can facilitate the enforcement of stateless firewall policies. However it is not suitable for enforcing stateful policies that observe the state of TCP connections. Other types of stateful policies can enforced natively by OpenFlow such as packet arrival time. Such a stateful policy can be used to filter traffic at specific times of the day. Scheduling such policies is challenging as comparisons made between rules only consider hours and minutes. This can be overcome by observing the time that a new rule is being scheduled for, the time that the rule at the head of the queue has been scheduled for and the time at which the new rule is being created (i.e. the current system time). Besides supporting the enforcement of stateful policies, the design above proposes a new method of security policy enforcement where different segments of a network can have policies domains enforced there.

Chapter 4

ACLSwitch Implementation

This chapter will cover implementation specific details of the solution using the options described in the previous chapter. The implemented SDN firewall solution was named ACLSwitch.

4.1 Development Environment

Both implementations of ACLSwitch were developed using a virtual machine image available from `http://sdnhub.org/tutorials/sdn-tutorial-vm/` (still accessible as of October 12, 2015). The image contained a 64-bit version of Ubuntu 14.04 that supports numerous SDN controllers such as Ryu, Floodlight and POX. The version of Ryu on the image was 3.22[1], meaning that it had full support for OpenFlow 1.3.

The version of Open vSwitch present on the image (2.3.90[2]) also had full support for OpenFlow1.3. The other tool present on the image that was instrumental for development was Mininet (version 2.2.1[3]).

The Python module `prettytable` was installed separately for use with an implemented command-line interface. The module allows for the creation of cleanly formatted tables which can be printed to a terminal. A public repository was created and hosted on GitHub[4] to contain implementation and test code along with the results of tests.

4.2 Stateless SDN Firewall Implementation

The stateless implementation of ACLSwitch was built to filter flows based on IPv4 addresses, IPv6 addresses and transport layer (TCP and UPD specifically) source and destination ports. An example Ryu application called `simple_switch_13.py` was used as a foundation for building the firewall. This Ryu application was included with the Git repository for Ryu and is freely available under the Apache 2.0 licence [26].

When executed, *simple_switch_13.py* operates as a learning switch by creating flows within a switch's flow table which map a switch's port to an Ethernet destination address. Whenever an OpenFlow switch receives a packet which cannot be matched in a flow table and given that a table miss entry has been provided, the packet gets forwarded to the controller for evaluation. At that point, *simple_switch_13.py* will create a flow table entry to allow flows of traffic matching the input port and the Ethernet destination address of the forwarded

[1] This can be determined by executing the `$ /ryu/bin/ryu-manager --version` command.
[2] This can be determined by executing the `$ ovs-vswitchd --version` command.
[3] This can be determined by executing the `$ mn --version` command.
[4] `https://github.com/bakkerjarr/ENGR489_2015_JarrodBakker`

packet. For the purposes of this project *simple_switch_13.py* was used as a template to be extended. This choice was made as the packet forwarding functionality provided by *simple_switch_13.py* is necessary for a network to operate. However it should be noted that it is not intelligent. If a machine were to change its switch port but remain on the same switch then it would longer receive traffic as *simple_switch_13.py* creates static mappings between MAC addresses and switch ports. As hosts cannot change their switch ports in Mininet this was not an issue and finding a solution to this particular issue was out of the project's scope. The solution to this would be to remove the primitive layer two switching functionality from the implementation and implement another application to handle that functionality. Figure 4.1 illustrates the high-level architecture for the stateless implementation of ACLSwitch.

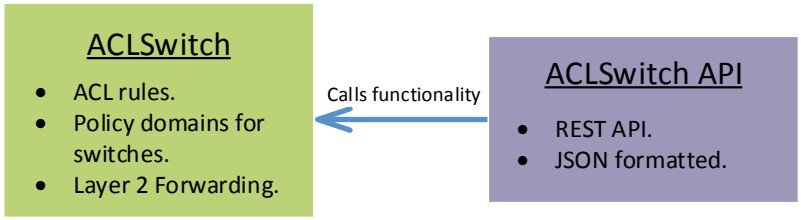

Figure 4.1: High-level illustratration of the stateless ACLSwitch application.

4.2.1 Data Structures

ACLSwitch uses various data structures to store the state of the system. Two of the most important data structures are the namedtuple `ACL_ENTRY` and the dictionary `_access_control_list`. For the stateless implementation of ACLSwitch, `ACL_ENTRY` represents a single entry within the ACL (also known as a rule). It stores the source and destination IP address, the type of transport protocol being used, the source and destination transport layer ports, and the policy that the rule is part of. IP addresses stored in the source and destination fields can be either be IPv4 or IPv6, it is the responsibility of the application to determine the IP version of those values if they need to be used.

An asterisk (i.e. "*") can be used as a field value to indicate a wild-card. If the source port field was set to "*" for example, then this would indicate that all source ports would match against this rule. Another useful example of using a "*" wild-card is where it is necessary to filter all traffic between two particular hosts. In this case the source and destination IP fields would be set accordingly and the transport layer protocol field would be set to "*".

The values that can be stored within an `ACL_ENTRY` object also follows the restrictions that the OpenFlow specification states. For instance, if the transport protocol field is set to "*" within an `ACL_ENTRY` object then the source and destination port fields must also be set to "*". This is because the source and destination port fields within a flow table entry require the `ip_proto` field to be set as a prerequisite. The same rule applies to IP addresses. If IP addresses are not provided then no fields can be specified at higher layers. It is worth noting that at least one of the IP address fields must be set i.e. not "*".

The namedtuple collection was used as it allows values to be addressed using a key (such as "ip_src") and the fields are immutable. Values in regular Python tuples cannot be addressed using a key and values within regular Python dictionaries are mutable. Addressing values within a data structure results in a program that is easier to read and immutability means that a rule's attributes cannot be altered once created. To ensure correctness, the syntax of rules are verified before being created. The `ACL_ENTRY` data structure was configured

as follows: `namedtuple("ACL_ENTRY", "ip_src ip_dst tp_proto port_src port_dst policy")`.

The dictionary `_access_control_list` contains the set of valid `ACL_ENTRY` objects. The choice was made to use a dictionary so that each object could be accessed by using a key. The key for an `ACL_ENTRY` object is a unique ID number. This allows rules to referred to by the other data structures that will be mentioned below.

The dictionary `_policy_to_rules` maps the name of policy to a list of rules ID numbers associated with that policy. The specifics regarding the use of policy domains is covered in Section 4.2.4. The other data structure that enables the use of policies is the dictionary `_connected_switches`. This dictionary maps the datapath ID of a switch to a list of policies associated with the switch.

The last data structure that was used comes from `simple_switch_13.py`. The dictionary `mac_to_port` is used to remember what MAC address is associated with a switch port. This facilitates layer two forwarding.

4.2.2 Table Pipeline

Multiple tables within the table pipeline were used to logically separate flow table entries associated with ACL rules from flow table entries associated with layer two forwarding. The first table in the pipeline, table 0, was designated as the one to hold flow table entries associated with ACL rules. It was important for ACL flow table entries to be placed in the first table as undesirable traffic should be filtered out before it can be forwarded to victim hosts. The second table in the pipeline, table 1, was designated as the one to hold flow table entries associated with layer two forwarding. Figure 4.2 illustrates the described table pipeline and behaviours.

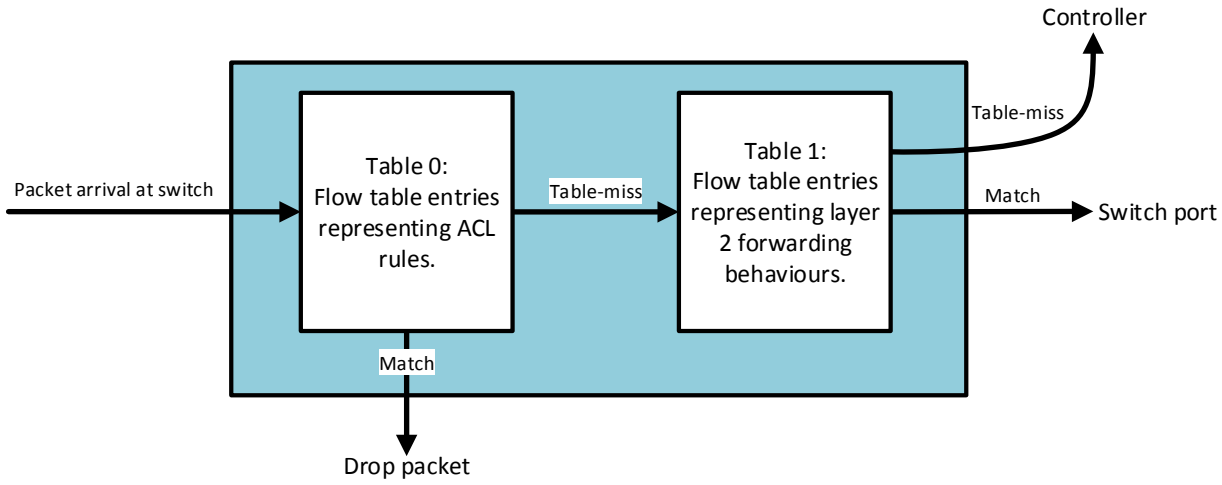

Figure 4.2: Table pipeline specific to ACLSwitch.

4.2.3 Flow Table Entry Creation and Removal

As mentioned above, an ACL_ENTRY object represents a single rule. Expanding on this, a rule makes a one-to-one mapping with a flow table entry which is then stored in switches. This is a relationship that was illustrated in Figure 3.4. As a one-to-one mapping exists, all rules must be unique in regards to IP source and destination addresses, the transport layer protocol and the source and destination ports. Rules are allowed to be added to and

removed from the _access_control_list data structure and as a result this behaviour needs to reflected in the management of flow table entries.

Creating OFPMatch Instances

For flow table entries to be added and removed from a switch, an OFPMatch instance must be created. Ryu provides an easy method for creating OFPMatch instances by using the datapath.ofproto_parser.OFPMatch() method. To use this method, any number of header match fields can be provided. For instance if no fields are specified then it is implied that all traffic will be matched. Note that the wild-cards used in ACL_ENTRY objects cannot be used here.

As rules can have wild-carded fields, the aforementioned datapath.ofproto_parser.OFPMatch() method is not suitable. Use of the method would require multiple conditional statements to match each case. For example one case would need to check for when only an IPv4 source address has specified, another case would need to check for when IPv4 source and destination addresses are both specified, another case would need to check for when IPv4 source and destination addresses are both specified along with a TCP source port. It is clear that this method would not scale if more protocols were allowed to be matched.

The solution to the above problem was to use the OFPMatch.append_field method. This method allows header match fields to be appended to an OFPMatch instance as required. Therefore whenever a field within an ACL_ENTRY object is not "*", the method can be called and the appropriate value appended to the OFPMatch instance.

Creating Flow Table Entries

The creation of flow table entries follows the proactive enforcement approach. This means that flow table entries are distributed to switches as the rules representing them become available. The proactive policy enforcement algorithm used in the solution can be summarised by the steps listed below. Note that *appropriate switches* refers to switches that are part of the rule's policy domain.

1. SDN firewall application receives a new ACL rule from an input source.
2. An OFPMatch instance is created which matches the traffic referred to in the new ACL entry.
3. The appropriate switches are sent a flow table entry with the OFPMatch instance created in the previous step along with a OFPACTION instance to drop the traffic.

It is interesting to note that the order of rules within table 0 does not matter from the perspective of enforcing a security policy as the outcome will be the same. The only requirement is that the table-miss entry must be at the bottom on the table. Regardless, the lack of ordering and structure can lead to redundant rules within the flow table. One way to handle this could be to assign different priorities to flow table entries based on the presence of a wild card. For example an entry matching a source IPv4 address of 10.0.0.1 and a destination IPv4 address of 10.0.0.2 should have a lower priority than a rule that only matches a source IP address of 10.0.0.1.

In the above example, the former results in the same behaviour as the latter, however the converse does not hold. In this case, the latter rule should be given a higher priority so that it gets matched sooner thus reducing the number of comparisons. However this does not solve the issue of redundant rules existing within the flow table. Optimising the placement of flow table entries within a flow table was outside the scope of the project, therefore such functionality was not implemented.

Removing Flow Table Entries

Flow table entry removal used the `OFPFC_DELETE_STRICT` modification message. As with the creation of flow table entries, only switches that are part of a rule's policy domain are sent the `OFPFC_DELETE_STRICT` modification message.

4.2.4 Using Policy Domains for Switches

The policy functionality utilises the controller's global view of the network to allow network security policies to vary across the network. This can also be used to reduce the number of flow table entries within a switch or define network segments such as a DMZ. Flow table entries that block traffic at switches located on network boundaries may drop traffic that will not be seen in the internal network. Therefore this can lead to flow table entries that will never receive matches. By assigning a different policy to such switches, the unused rules can be removed the switch. Policies are able to be created and deleted. They can also be assigned and unassigned from switches.

In the solution, a switch was identified by its datapath ID: a positive integer used by OpenFlow to identify a switch. This method was well suited to the Mininet development environment as a switch's number was equal to its datapath ID. For example a switch s1 would be given a datapath ID of 1.

A feature of policy domains is that they from a many-to-one mapping with ACL rules. This enables fine grained policies to be written and enforced. On top of this, redundant flow table entries will be prevented from being dispatched and overriding flow statistics. Overriding flow statistics was not an issue for stateless policies in this solution but it would prevent extra functionality from being developed that required accurate flow statistics.

4.2.5 Configuring ACLSwitch to Enforce Policies

Two methods of configuring ACLSwitch were implemented. Upon application start-up, policies and rules can be declared from a file named `config.json`. In this file policies must be declared before rules so that the policies exist within ACLSwitch. Policy assignments cannot be made from this file either as it cannot be assumed that a specified switch is known to the controller when the file is read.

AClSwitch also utilised Ryu's support for Python WSGI to create a RESTful interface. This allows an external application to observe and modify the state of the data structures within ACLSwitch. HTTP verbs are used to specify the kind of action being performed (creating a resource, modifying a resource, removing a resource or viewing a resource) and the URL specifies what the action is being performed on. JSON was used as a method for packaging information in an agreed upon format. The combination of a HTTP verb, URL and JSON payload allowed a grammar to be defined that could facilitate the transfer of data using a RESTful approach. The complete API can be found in Appendix A.

As an example, the following would be used by an external application to assign a policy to a switch:

- HTTP verb - PUT
- URL - `http://127.0.0.1:8080/acl_switch/switch_policies/assignment`
- JSON payload - {"switch_id":<switch ID encoded as a string>, "new_policy":<policy name encoded as a string>}

4.3 Stateful SDN Firewall Implementation

The stateful implementation of ACLSwitch extended the stateless implementation to allow stateful filtering of traffic. The chosen form of state to filter traffic by was packet arrival time. Packet arrival time policies are enforced at the switches as before, however the key difference is that flow table entries exist in the switch temporarily. To facilitate this behaviour, the controller must keep track of when flow table entries need to be dispatched and such entries need to utilise the `hard_timeout` field.

Figure 4.3 illustrates the high-level architecture for the stateful implementation of ACLSwitch. Note that the differences between this implementation and the stateless implementation use a red font colour.

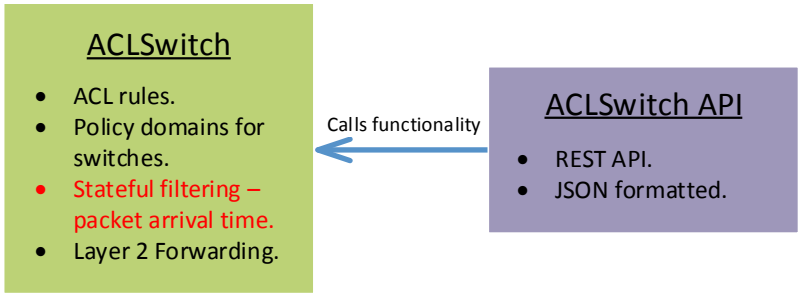

Figure 4.3: High-level illustratration of the stateful ACLSwitch application.

4.3.1 Changes to the Stateless Implementation

The stateful version of ACLSwitch appends extra fields to the `ACL_ENTRY` data structure in order to enable the enforcement of time bound policies. The two fields appended were time_start and time_duration. This resulted in the data structure declaration being change to `namedtuple("ACL_ENTRY", "ip_src ip_dst tp_proto port_src port_dst policy time_start time_duration")`.

A new data structure was created to maintain a schedule of rules that need to be dispatched. The list `_rule_time_queue` is used as a queue where rule IDs are ordered based on the respective rule's time to be dispatched. The head of the queue is the next rule to be dispatched. To allow multiple rules to be scheduled for dispatch at the same time, the data structure was designed at a high level to be a queue of lists.

The REST API for ACLSwitch was modified to facilitate the creation and removal of time bound rules. Users can also see the current state of the schedule.

4.3.2 Rule Scheduling and Dispatch

After time bound rules have been created and inserted into the `_access_control_list` data structure, they are inserted into `_rule_time_queue`. A single rule can only to scheduled for a single time period. This restriction cannot be avoided by duplicating an `ACL_ENTRY` object and scheduling it for a different time as all rules are unique based on IP source and destination addresses, the transport layer protocol and the source and destination ports. The process of insertion comprises of three steps:

1. Check if the queue is empty and if so insert rule ID at the head of the queue.

2. Check if the head of the queue needs to be pre-empted and if so insert rule ID at the head of the queue.

3. If the rule ID does not need to be inserted at the head of the queue, then insert it while maintaining the correct order.

Once on the queue, a green thread is created to execute the `_distribute_rules_time` function to dispatch rules when necessary. If the head of the queue is pre-empted then the currently running green thread is killed and a new one is created. The `_distribute_rules_time` function sets the green thread to sleep until the time indicated by the rule at the head of the queue. Once the time has passed, any rules that need to be dispatched are sent to their appropriate switches and the queue to rearranged to reflect the new order. This process is repeated in a `while(True)` loop.

After rules have been dispatched it is necessary to sleep the green thread for a second. As it takes less that a second to dispatch rules, when the function inside the green thread loops back around, the function still believes that it is necessary to dispatch rules and so this cycle repeats until a second has passed. Dispatching a rule multiple times does not in duplicate flow table entries being present with the flow table but it will overwrite the fields associated with flow statistics. Besides overwriting flow statistics that might be needed by other applications, a flow table entry will remain in a switch for a second longer than intended.

The flow table entry attribute `hard_timeout` was used to implement the behaviour of a rule being applied over a period of time. Remembering that the upper limit for this attribute is roughly 18 hours, the decision was made to only allow rules to be created if they lasted no more than 18.2 hours (1092 minutes). There would typically be few cases where an 18 hour rule would be needed.

4.4 Command-line Interface

A command-line interface was developed to utilise the ACLSwitch API. The interface was created as a means to easily use the API and so was not built to look visually pleasing. The interface performs syntax checking of rules before sending them to the ACLSwitch application so that a user can receive feedback promptly. A user can use the interface to:

- View the state of the data structures.

- Create and remove rules (time bound rules included).

- Create, remove and change the assignment of policies.

As stated in Chapter 3, the purpose of the project was not to design an interface. As a result the design choices made concerning the interface's operation will not be discussed. However, Figure 4.4 illustrates the high-level operation of the interface through a finite state machine.

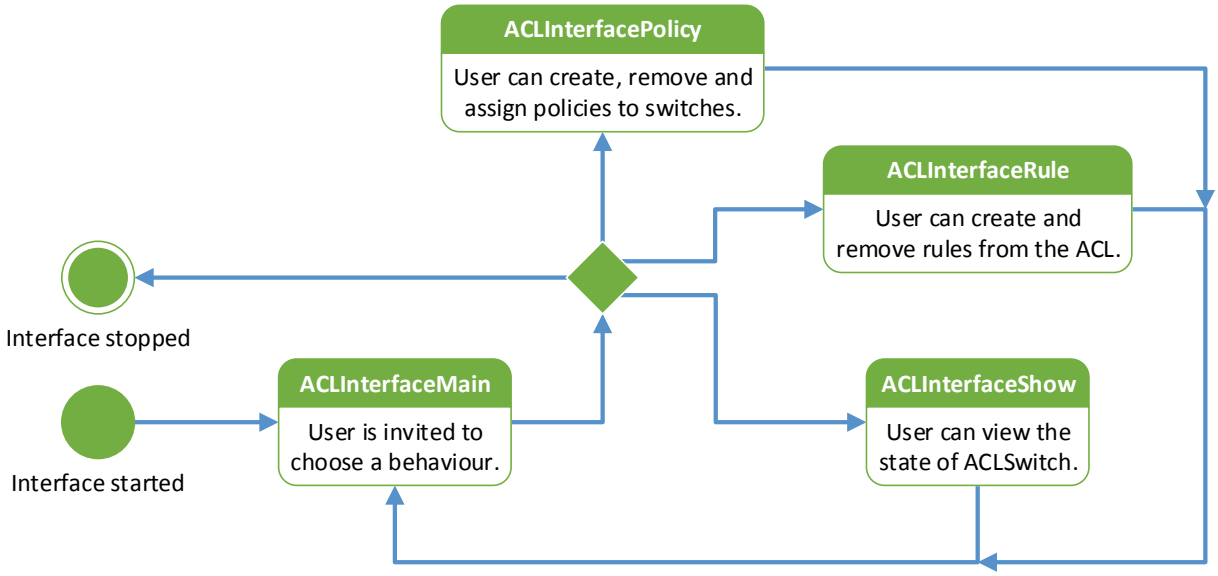

Figure 4.4: Finite state machine illustrating the operation of a command-line based interface for ACLSwitch.

4.5 Chapter Summary

The proposed solution utilises different features of OpenFlow to enforce network security policies. It has the ability to enforce both stateless and stateful policies by blocking traffic based on IP addresses and transport layer port numbers. The stateless part of the firewall can proactively distribute flow table entries to switches as the respective rules are added to the ACL. The stateful part of the firewall can dispatch flow table entries at preconfigured times that will timeout in accordance with their respective ACL entries. An entire network topology can be covered by a single security polices or switches can be assigned different policy domains in order to filter different traffic at different parts of the network.

Chapter 5

Evaluation

This chapter presents the functional evaluation of ACLSwitch. To ensure the correct operation of the implementation, several testing scenarios were devised to verify the different aspects of ACLSwitch. The functional tests can be viewed as a framework for evaluating other firewall configurations. The stateful implementation of ACLSwitch was used for evaluation as it contains both the functionality for enforcing stateless and stateful policies.

5.1 Evaluation Method

Functional tests were used to validate the correct operation of ACLSwitch and ensure that the correct behaviours were exhibited. The functional tests also provided a method of confirming that the design goals stated in Chapter 3 Section 3.1 were met. The design goals for firewalls as defined by Cheswick et al. were also used to evaluate the functional fitness of ACLSwitch. Finally, comparisons with the SDN firewall case studies were made in order to evaluate the contributions made through the development of ACLSwitch.

5.1.1 Test Environment

The testing environment used was an extension of the development environment specified in Chapter 4 Section 4.1. The extra Python modules used were `netifaces` and `Scapy`. `netifaces` provides the programmer with an easy method of obtaining a machine's IP addresses. `Scapy` is a powerful packet manipulation module that allows a programmer to quickly configure port scans. In the context of the evaluation, different flows of traffic can be defined and sent through the firewall programmatically and the results can be gathered by functions defined by `Scapy`.

`Scapy` enabled the testing of all IPv4 flows and IPv6 flows that contained an ICMPv6 Echo Request. TCP and UDP flows sent over IPv6 did not receive replies from the target host in the same way when sent over IPv4. Even with no rules blocking such traffic, the expected behaviour is for the destination host to reset the connection but this was not the case with IPv6. See Appendix B Section B.1 for more information.

This behaviour was not exhibited when initiating a SSH connection over IPv6. As SSH uses TCP over IPv6 and `Scapy` can successfully send IPv6 traffic, I concluded that the issue was with the protocol stack on the destination host and how it handles unexpected IPv6 TCP connections. In order to test other kinds of IPv6, the Python module `Paramiko` was used to establish SSH connections over IPv6. The limiting factor of using this method is that only TCP flows can be tested with the destination port can be specified.

5.1.2 Test Case Topologies

The figures below illustrate four different topologies that were used for testing. Note that in all cases c refers to an OpenFlow enabled SDN controller, s refers to an OpenFlow enabled switch and h refers to a host machine. The integer following each letter is used as an identifier. Each host had an IPv4 address of 10.0.0.*i*, an IPv6 address of fe80::200:ff:fe00:*i* and a MAC address of 00:00:00:00:00:0*i* (or 00:00:00:00:00:*i* if *i* is larger than 15). Note that for IPv6 and MAC addresses *i* is expressed as a hexadecimal integer.

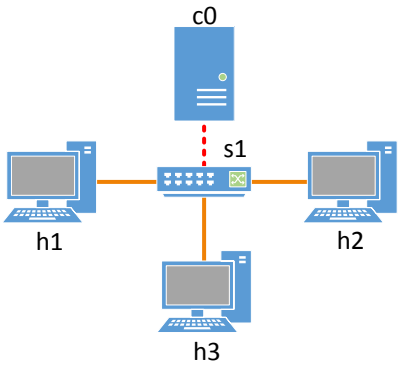

Figure 5.1: A star topology with a controller (c0), an OpenFlow enabled switch (s1) and three hosts (h1, h2 and h3).

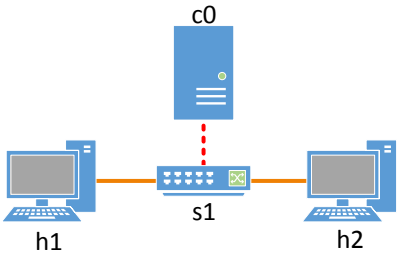

Figure 5.2: A star topology with a controller (c0), an OpenFlow enabled switch (s1) and two hosts (h1 and h2).

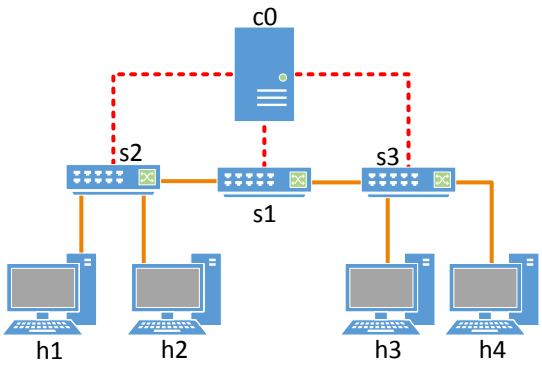

Figure 5.3: A tree topology with a controller (c0), three OpenFlow enabled switches (s1, s2 and s3) and four hosts (h1, h2, h3 and h4).

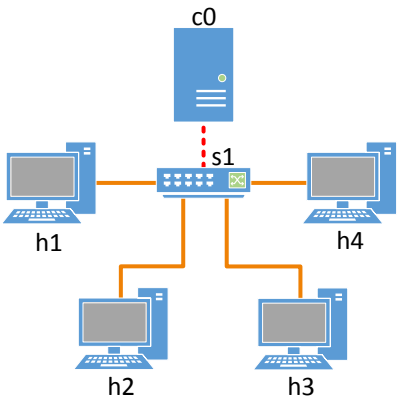

Figure 5.4: A star topology with a controller (c0), an OpenFlow enabled switch (s1) and four hosts (h1, h2, h3 and h4).

5.1.3 Test Process

All but one of the tests (Stateful Scheduling) used at least one of the topologies shown above in Mininet. The appropriate `config.json` file was used for each test to configure the necessary policies before traffic was sent through the network. Scripts particular to the test would then be used to send flow of traffic through the network topology so that the behaviour could be observed. The behaviour could be observed and verified at the host running the

test and by executing the `dpctl dump-flows -O OpenFlow13` command in Mininet to dump the contents of the flow tables of every connected switch to the terminal window.

5.2 Test Cases

These tests were used to verify the behaviour of the ACLSwitch. In this section, the test cases will be outlined and their results described. Appendix C contains the complete set of test details, including specific information on individual test files. The first test suite mentioned in 5.2.1 has all details included in the main body of the report in order to demonstrate the evaluation methodology and process. The following details should be noted as well:

- The Mininet network and the SDN controller were restarted between each test.
- A *range of traffic* refers to ICMP Echo Requests over IPv4, TCP over IPv4, UDP over IPv4 and ICMPv6 Echo Requests over IPv6 except where noted.
- *All traffic* refers to all flows of traffic that can be filtered by ACLSwitch.

When blocking traffic, each test assumes that traffic blocked the switch does not make it to the destination host. Appendix B Section B.2 outlines a test that shows that the assumption is correct.

5.2.1 Scapy Test Suites

The tests contained within each Scapy test suite were configured programmatically using the Python module of the same name (`Scapy`). They were run without the need for any other scripts or intervention besides the insertion of a `config.json` file for the ACLSwitch application. These tests verify that a single network security policy can be applied across a network topology.

Suite 1 - NoDrop

This suite aimed to verify that traffic does not blocked unintentionally at a switch when there are no rules stating that it should be blocked. The network topology used was the one depicted in Figure 5.1. Below is the list of test files used along with an elaboration:

1. NoDrop_EmptyACL - With no rules present to drop any traffic, send a range of traffic through the switch from h1 to h2 and h3.
2. NoDrop_IPv4Ping - With rules to drop all TCP and UDP over IPv4 traffic between the hosts, send an ICMP Echo Request over IPv4 from h1 to h2 and h3.
3. NoDrop_IPv4TCP - With rules to drop all UDP over IPv4 traffic between the hosts, send TCP traffic from h1 to h2 and h3 with varying source and destination ports.
4. NoDrop_IPv4UDP - With rules to drop all TCP over IPv4 traffic between the hosts, send UDP traffic from h1 to h2 and h3 with varying source and destination ports.
5. NoDrop_IPv6Ping - With rules to drop all TCP and UDP over IPv6 traffic between the hosts, send an ICMPv6 Echo Request over IPv6 from h1 to h2 and h3.

In the case of test file NoDrop_IPv4TCP the choice to block all UDP over IPv4 was made because the only other kind of traffic that can be blocked with halting communication is UDP. The same reasoning was used with the test file NoDrop_IPv4UDP and the choice to block all TCP over IPv4. The term *varying source and destination ports* refers to preconfigured selection of source and destination ports.

The tests within this suite confirmed that traffic does not get blocked accidentally at a switch. In the cases where rules were inserted into the ACL the table dumps revealed that none of the flow table entries associated with blocking traffic were matched by any packets. Therefore it was concluded that traffic cannot be blocked accidentally.

Suite 2 - Drop All H1

This suite aimed to verify that when a range of traffic is sent from a host, it can get blocked at a switch. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a source IP address of h1.

The tests within this suite confirmed that traffic was blocked at switch when the source IP address and transport layer protocol fields were specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that an entire type of traffic (TCP, UDP, etc) can be blocked when it originates from a specific host.

Suite 3 - Drop Select H1

This suite aimed to verify that rules could be used to block specific TCP and UDP flows at a switch. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a source IP address of h1.

The test files that use IPv4 first sent a series of flows where both the source and destination ports are specified. Following this, flows were sent where the destination port was specified but the source as not. The source port in this case was assigned a random (seeded) destination port between 32768 and 61000 to mimic what an operating system kernel typically does with outgoing connections - it assigns an ephemeral port. The last flows sent had the source port specified but the destination not. This ensured that a range of cases was covered. Care was taken with the latter flows as Scapy automatically sets the destination port to 80 if it is not specified. However this results in the flows matching a rule to block port 80 traffic from h1. This is an interesting result as the behaviour is the same as the flow is still blocked but it was not being matched by the expected flow table entry. Therefore the latter flows were assigned a random destination port in a similar fashion to the flows with random source ports. All ACL rules filtered traffic with h1's source IP address.

The tests within this suite confirmed that traffic was blocked at a switch when the source IP address and transport layer port number fields are specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that specific TCP (over IPv4 and IPv6 when destination port is specified) and UDP (over IPv4 only) flows can be blocked when required.

Suite 4 - Drop All H1 Destination

This suite aimed to verify that when a range of traffic is sent to a specific host, it can get blocked at a switch. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a destination IP address of h1.

When a UDP port is scanned and found to closed, an ICMP packet carrying a *Destination port unreachable* message is sent back to the source. Therefore it was not adequate for the test file DropAllH1Dest_IPv4UDP to follow the same pattern used for other tests where h1 sends to h2 and h3. To complete the test, the test file was run on h2 with traffic being directed at h1.

The tests within this suite confirmed that traffic was blocked at a switch when the destination IP address and transport layer protocol fields were specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that an entire type of traffic (TCP, UDP, etc) can be blocked when it is destined for a specific host.

Suite 5 - Drop Select H1 Destination

This suite aimed to verify that rules could be used to block specific TCP and UDP flows targeted for a specific host. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a destination IP address of h1.

The test file DropSelectH1Dest_IPv4UDP used the same method as DropAllH1Dest_IPv4UDP where traffic was sent h2 to h1. The test file DropSelectH1Dest_IPv6TCP_destPort had an important difference with its counterpart DropSelectH1_IPv6TCP_destPort. Because h1 initiated the request and the intention was to block the reply, the TCP source port coudl be filtered.

The tests within this suite confirmed that traffic was blocked at a switch when the destination IP address and transport layer port number fields are specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that specific TCP (over IPv4 and IPv6 when the source port is specified) and UDP (over IPv4 only) flows can be blocked when required.

Suite 6 - Drop All Two Host

This suite aimed to verify that rules could be used to block all traffic between two hosts when both the source and destination IP addresses are specified in a rule. The network topology used was the one depicted in Figure 5.2. All ACL rules filtered traffic with a source IP address of h1 and destination IP address of h2.

The tests within this suite confirmed that traffic was blocked at a switch when the source and destination IP addresses were specified in a rule. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore was be concluded that all traffic can be blocked between two specific hosts.

Suite 7 - Drop Select Two Host

This suite aimed to verify that rules could be used to specific TCP and UDP flows between two hosts when both the source and destination IP addresses are specified in a rule. The network topology used was the one depicted in Figure 5.2. All ACL rules filtered traffic with a source IP address of h1 and destination IP address of h2.

The tests within this suite confirmed that traffic was blocked at a switch when the source and destination IP addresses, and source and destination transport layer port numbers were specified in a rule. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times.

Therefore it was concluded that specific TCP (over IPv4 and IPv6 when the source port is specified) and UDP (over IPv4 only) flows can be blocked between two specific hosts.

Suite 8 - Drop H1 Same Rule Different Switch

This suite aimed to verify that a single network security policy could be enforced over an entire network topology. The network topology used was the one depicted in Figure 5.3. To test the coverage of a policy yet still allow h1 to send traffic, the source address field for a rule was wild-carded. Therefore all ACL rules filtered traffic with a destination IP address of h1. Note that UDP cannot be tested in this way as the destination host replies with ICMP packets and not with UDP.

The tests within this suite confirmed that traffic was blocked at all switches when the destination IP address was specified in a rule. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. As the previous suites showed that a range of different rules could be enforced at a switch, it holds that the same range of rules could also be enforced on multiple switches. Therefore it was concluded that a single network security policy could be enforced over an entire network topology.

5.2.2 Rule Removal Test

The purpose of this test was to verify that rules could be removed from the ACL and the changes could be observed in the network. This is an important feature to test as it is not uncommon for network security policies to change over time. The network topology used was the one depicted in Figure 5.1.

The test found found that rules could be removed from the ACL, resulting in network security policy that changed over time. As the test progressed, h1's connectivity changed between being able to contact its neighbours to not being able to at all. Producing table dumps before and after each communication attempt revealed that the changes in the policy were being reflected in the switch's flow table. Therefore it was concluded that rules can be successfully removed from the ACL with the changes taking effect in the network.

5.2.3 Policy domain tests

The purposes of these tests were to verify that different policy domains could be enforced at different switches. The network topology used for both tests was the one depicted in Figure 5.3. In the context of the tests below, s1 is a switch that connects two networks with one another.

Policy Assignment

The purpose of this test was to verify that a policy could be assigned to s1 that would stop a host from communicating with hosts outside of its network. The test did not send traffic with various source and destination port numbers as the previous tests proved that variation could be enforced at a single switch.

The test found that h1 could only communicate with h2 in tests 1 and 2 and h3 could only communicate with h4 in tests 3 and 4. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that switches can enforce different policies when they are assigned to them.

Policy Removal

The purpose of this test was to verify that a policy could be removed from a switch and the changes could be observed in the network. The test did not send traffic with various source and destination port numbers as the previous tests proved that variation could be enforced at a single switch.

The test found that h1's connectivity with other hosts matched the policy domains assigned to s1 at different points in time. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that a policy assignment can be removed from a switch.

5.2.4 Stateful policy tests

The purpose of these tests were to verify the the correct operation of stateful policies. The tests did not use a variety of traffic types (i.e. different source and destination port combinations) as the stateful implementation builds upon what has already been tested and verified in the stateless implementation.

Scheduling

A series of tests were used to ensure that rules can be scheduled in the correct position within the queue. Each test file tested a different aspect, such as in what order are rules added and the time relative to the current system time. The script inserts rules relative to the current system time that the test file was run on. Therefore the tests can be executed at any time of the day and the ordering of the rules should remain consistent. Note that *before the current system time* means that the rule should be scheduled for the next day, however the hours and minutes of the new rule alone place it in front of the current time. For example, if the current time wass 1400 and a rule needed to be scheduled for 1300 then it should be scheduled for the next day.

The test found that rules can be scheduled the correct order. The tests covered a range of different scenarios to ensure correctness. Therefore it was concluded that stateful policies rules can be correctly scheduled.

Dispatch

Two scenarios were used to verify that stateful policies can be dispatched at their correct times and be enforced for their designated period of time. The first test aimed to confirm that a single stateful policy could be dispatched correctly for a given period of time. The second aimed to confirm that multiple stateful policies could be scheduled for dispatch at the same time as well as at different times. The network topologies used for the first and second tests are depicted in figures 5.2 and 5.4 respectively. Below is the list of test files used along with an elaboration:

The test found that the stateful policies dropped packets within the correct time frames. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that stateful policies can be correctly enforced.

5.2.5 Test Result Summary

The success of all tests above verify the premise that ACLSwitch can operate as a stateful SDN firewall. Even though the IPv6 aspects were not as thoroughly tested as their IPv4 counterparts, some assumptions can be made. As it was confirmed that traffic can be blocked based on a source IPv6 address and a TCP destination port, and a destination IPv6 address and a TCP source port, it can be assumed that IPv6 traffic carrying a TCP connection can be blocked similar to IPv4 traffic carrying a TCP connection. As none of the tests were able to generate IPv6 traffic carrying a UDP datagram, successful matching of IPv6 traffic using the UDP match field in flow table entries cannot be determined.

5.3 Design Goal Assessment

The functional tests described in Section 5.2 covered the design goals that were established earlier, this is illustrated in Table 5.1. From looking at the table, it is clear that certain goals were addressed multiple times by different tests. Different tests may be nuanced by a particular case being tested, such as filtering traffic destined for a host and filtering traffic originating from a host. In other cases it, such as with stateless design goal 1, it was necessary for it to be met in all tests.

Test	Stateless Design Goals									Stateful Design Goals			Topology
	1	2	3	4	5	6	7	8	9	1	2	3	(Figure)
Scapy Suite 1	×		×	×			×		×	×			5.1
Scapy Suite 2	×	×		×			×		×	×			5.1
Scapy Suite 3	×	×		×			×		×	×			5.1
Scapy Suite 4	×	×		×			×		×	×			5.1
Scapy Suite 5	×	×		×			×		×	×			5.1
Scapy Suite 6	×	×		×			×		×	×			5.2
Scapy Suite 7	×	×		×			×		×	×			5.2
Scapy Suite 8	×	×		×			×		×	×			5.3
Rule removal	×	×		×	×		×	×	×	×			5.1
Policy Assignment	×	×		×		×	×		×	×			5.3
Policy Removal	×	×		×		×	×	×	×	×			5.3
Stateful Scheduling												×	N/A
Stateful Dispatch	×										×		5.2 and 5.4

Table 5.1: Table illustrating the design goal coverage from the functional tests.

The design goals for firewalls defined by Cheswick et al. can also be used to evaluate ACLSwitch. ACLSwitch addresses the first goal as it allows policies to be enforced at all switches thus covering the entire network. As all traffic would flow through at least one of the switches with a software defined network, it follows that all traffic passes through the firewall. This is a goal that traditional firewalls fail to meet they are not deployed between all links. The second goal is addressed by ACLSwitch's permissive operation; by default all traffic is allowed to pass through the network. Traffic is only blocked with an ACL rule is created that matches the respective flow.

The third goal can only be addressed if the machine running the controller is part of a network that is separate from the hosts and out-of-band control is being utilised. Out-of-band control ensures that OpenFlow control plane traffic does not use the same link as regular traffic. As a result, hosts will be unable to communicate with the controller and alter

the state of the firewall. However if hosts can contact the controller over the network then the state of the firewall could be altered. This is the case because the framework provided by Ryu to configure REST APIs does not secure connections made with applications. Regardless, any attacker that could access the machine that the controller runs on (through physical access for example) could also alter the ACLSwitch configuration.

5.4 Comparison with the SDN Firewall Case Studies

ACLSwitch, like the case studies presented in Chapter 2 Section 2.3.2, fits both methods of classification for a firewall. Specifically, its a stateful packet filter but it can be categorised further. ACLSwitch is different compared to the case studies in that it is permissive (it only blocks traffic that is explicitly specific in the ACL). However unlike the solutions proposed by Suh et al. and Collings and Liu, ACLSwitch proactively distributes flow table entries to the switches as the rules are created. Table 5.2 summarises these details.

Case study	Controller	OF version	Classifier 1	Classifier 2	Policy	Rule dist.
ACLSwitch	Ryu	1.3	Stateful	Packet filter	Permissive	Proactive
Suh et al.	POX	1.0	Stateless	Packet filter	Restrictive	Reactive
rest_firewall.py	Ryu	1.0, 1.2, 1.3	Stateless	Packet filter	Restrictive	Proactive
Collings and Liu	POX	1.0	Stateful	Packet filter	Restrictive	Reactive
Valve	Ryu	1.3	Stateless	Packet filter	Restrictive	Proactive
Sasaki et al.	-	1.1	Stateless	Packet filter	-	Proactive

Table 5.2: Comparison of ACLSwitch with the SDN firewall case studies.

As noted in Chapter 2 Section 2.3.2, the Ryu rest_firewall.py solution allows separate rules to be distributed to different switches. However the implementation does not allow for a single rule to be distributed to multiple switches at the same time. Multiple API calls have to made in order to carry out that functionality which can lead to a user introducing errors in issued commands. To maintain consistency with `rest_firewall.py`, a user must have knowledge of the topology and the purpose of each switch. ACLSwitch does not share the same dependency, a single API call can be made to enforce a rule across the network.

5.5 Chapter Summary

The evaluation process showed that ACLSwitch can successfully enforce security policies in a test environment. Not only that, but different policies can be assigned to different switches and stateful policies can enforced as well. As a result of passing the functional tests, the design goals that were established were met. The design goals for firewalls as defined by Cheswick et al. were met as well. ACLSwitch also sets itself apart from the SDN firewall case studies as it can enforce stateful policies and it provides a mechanism for enforcing security policies across multiple switches without the need to configure each switch separately.

Chapter 6

Conclusions and future work

This chapter will discuss conclusions and their relevance. Future work will be described following that.

6.1 Conclusions

The traditional method of deploying firewalls at network boundaries results in hosts within such boundaries to become vulnerable to attacks from neighbouring hosts. This report presented ACLSwitch, a SDN firewall solution that utilises the advantages of a logically centralised controller with global knowledge of a network topology that can enforce policies across an entire network. The solution is flexible enough that different policy domains can be enforced at different points in the network as well. Such functionality can be used to easily create a DMZ within a network, thus adding an extra layer of protection. Case studies provide similar functionality but they fail to apply policy models that allow groups of switches to be configured simultaneously.

The ability to enforce different policies at different points within a network can be used by a network administrator to easily filter suspicious traffic from a subverted host inside an enterprise network. This can also be scaled up to an ISP scenario where DDoS traffic can be filtered from a group of edge routers with a single policy. Therefore offending traffic can be contained within a particular area of the network without having to burden other nodes that might never see the attacking flows.

As a result of this investigation, it is apparent that algorithms for enforcing stateless policies can be implemented using OpenFlow 1.3. However their functionality is restricted by what the OpenFlow 1.3 specification can offer. A traditional approach to facilitate the enforcement of stateful policies involves accessing TCP flag information, this information can not be accessed by versions of OpenFlow earlier than 1.5. Stateful policies can be implemented given this constraint however. This project demonstrated this by implementing algorithms which use packet arrival time as packet state. In turn, this facilitates the blocking of traffic for periods of time throughout a day.

As the OpenFlow protocol matures and more functionality is supported by switches and controllers, the set of possible policies that are enforcible by a SDN firewall will increase in number. Just as ACLSwitch has changed the way that policies can be enforced across groups of switches through an established version of OpenFlow, future solutions will be able to take advantage of new features that offer new possibilities in the area of policy enforcement. The work that has been presented offers an insight into the future of firewalls.

6.2 Future Work

This section outlines potential future work. The items that will be mentioned below can either be classed as improvements or extensions to the current implementation.

OpenFlow allows flow statistics to be obtained from switches. Statistics include: the number of packets matched by a flow table entry (the `n_packets` field) and the total number of bytes matched by a flow table entry (the `n_bytes` field). By fetching these values and presenting them to a user, it could be possible to determine whether or not a policy is placed appropriately within a network by using the `n_packets` field, or predict the attack payload by comparing the `n_bytes` field with the payload sizes of other attacks.

Once OpenFlow 1.5 is fully supported by Open vSwitch, the functionality for enforcing stateful policies using TCP flags can be implemented. Flow table entries can be used to match traffic and update a connection table. Updates to the connection table will require switches to communicate with the controller. This architecture may not scale however as the number of TCP connections increase. MacVittie suggests that stateful solutions that use TCP flags should be separate from stateless solutions due to the load that would be placed on the controller [27]. Investigation would need to be conducted in order to determine the performance impact for the controller, the switches and connected hosts when using such a configuration.

In the Chapter 5, I identified an issue concerning the REST API framework provided by Ryu. The API should require an application to authenticate themselves, this would ensure that only privileged users can alter the state of the firewall. It would also be necessary to use a technology such as SSL to establish encrypted links in order to mitigate the risk of man-in-the-middle attacks.

Bibliography

[1] M. E. Whitman, H. J. Mattord, R. D. Austin, and G. Holden, *Guide to Firewalls and Network Security: With Intrusion Detection and VPNs*. Course Technology, second ed., 2009.

[2] D. Kreutz, F. M. V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, pp. 14–76, Jan. 2015.

[3] Open Networking Foundation, "Software-defined networking (sdn) definition," 2015. Retrieved 6 October, 2015, from `https://www.opennetworking.org/sdn-resources/sdn-definition`.

[4] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security*. Addison-Wesley, second ed., 2003.

[5] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing Inc., 2 ed., 2008. pp. 640.

[6] W. R. Cheswick and S. M. Bellovin, *Firewalls and Internet Security*. Addison-Wesley, first ed., 1994.

[7] J. Collings and J. Liu, "An openflow-based prototype of sdn-oriented stateful hardware firewalls," in *2014 IEEE 22nd International Conference on Network Protocols (ICNP)*, pp. 525–528, IEEE, Oct. 2014.

[8] H. B. Acharya, A. Joshi, and M. G. Gouda, "Firewall modules and modular firewalls," in *2010 18th IEEE International Conference on Network Protocols (ICNP)*, pp. 174–182, IEEE, Oct. 2010.

[9] G. Ziemba, D. Reeda, and P. Traina, ""Security Considerations for IP Fragment Filtering", *RFC 1858*," October 1995.

[10] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "Sane: A protection architecture for enterprise networks," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

[11] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 1–12, Aug. 2007.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[13] Open Networking Foundation, "Openflow switch specification - version 1.3.5," March 2015. Retrieved 9 May, 2015, from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf.

[14] Open Networking Foundation, "Openflow switch specification - version 1.5.1," March 2015. Retrieved 9 May, 2015, from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[15] Georgia Tech, "Ryu, floodlight, nox, and pox - georgia tech - software defined networking," February 2015. Retrieved 20 March, 2015, from https://www.youtube.com/watch?t=6&v=MGyToL4ZWLY.

[16] Open Source Software Computing Group, "Getting started - ryu 3.26 documentation," 2014. Retrieved 6 October, 2015, from http://ryu.readthedocs.org/en/latest/getting_started.html\#support.

[17] R. Izard, "Announcing floodlight v1.0," January 2015. http://www.projectfloodlight.org/blog/2015/01/02/announcing-floodlight-v1-0/.

[18] M. Suh, S. H. Park, B. Lee, and S. Yang, "Building firewall over the software-defined network controller," in *2014 16th International Conference on Advanced Communication Technology (ICACT)*, pp. 744–748, IEEE, Feb. 2014.

[19] Open Source Software Computing Group, "Firewall - ryubook 1.0 documentation," 2015. Retrieved 8 May, 2015, from http://osrg.github.io/ryu-book/en/html/rest_firewall.html.

[20] WAND, "Not so simple switching," February 2015. Retrieved 16 May, 2015, from https://ecs.victoria.ac.nz/foswiki/pub/Events/SDNWorkshop/WSW-Valve.pdf.

[21] T. Sasaki, Y. Hatano, K. Sonoda, Y. Morita, H. Shimonishi, and T. Okamura, "Load distribution of an openflow controller for role-based network access control," in *2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 1–6, IEEE, 2013.

[22] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "Flowguard: Building robust firewalls for software-defined networks," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 97–102, ACM, 2014.

[23] Open vSwitch, "Ovs faq," 2015.

[24] RYU project team, "Rest linkage," 2014. Retrieved 10 July, 2015, from http://osrg.github.io/ryu-book/en/html/rest_api.html.

[25] Eventlet Contributors, "Basic usage - eventlet 0.17.4 documentation," 2010. Retrieved 4 September, 2015, http://eventlet.net/doc/basic_usage.html.

[26] Open Source Software Computing Group, "Ryu component-based software defined networking framework," 2015. Retrieved 7 July, 2015, from https://github.com/osrg/ryu.

[27] L. MacVittie, "Sdn prerequisite: Stateful versus stateless," April 2014. Retrieved 30 April, 2015, from https://devcentral.f5.com/articles/sdn-prerequisite-stateful-versus-stateless.

Appendix A

ACLSwitch REST API

This appendix describes the REST API for ACLSwitch.

Return information about ACLSwitch

HTTP verb: GET
URL: `http://127.0.0.1:8080/acl_switch`
JSON: N/A

Return information on what policies are assigned to each switch

HTTP verb: GET
URL: `http://127.0.0.1:8080/acl_switch/switches`
JSON: N/A

Return a list of of the currently available policy domains

HTTP verb: GET
URL: `http://127.0.0.1:8080/acl_switch/switch_policies`
JSON: N/A

Return the current contents of the ACL

HTTP verb: GET
URL: `http://127.0.0.1:8080/acl_switch/acl_rules`
JSON: N/A

Return a list representing the queue of scheduled stateful rules

HTTP verb: GET
URL: `http://127.0.0.1:8080/acl_switch/acl_rules/time`
JSON: N/A

Create a new policy domain

HTTP verb: POST
URL: `http://127.0.0.1:8080/acl_switch/switch_policies`
JSON: {"policy":<policy name encoded as a string>}

Delete a policy domain

HTTP verb: DELETE
URL: `http://127.0.0.1:8080/acl_switch/switch_policies`
JSON: {"policy":<policy name encoded as a string>}

Assign a policy domain to a switch

HTTP verb: PUT
URL: `http://127.0.0.1:8080/acl_switch/switch_policies/assignment`
JSON: {"switch_id":<switch ID encoded as a string>, "new_policy":<policy name encoded as a string>}

Assign a policy domain to a switch

HTTP verb: DELETE
URL: `http://127.0.0.1:8080/acl_switch/switch_policies/assignment`
JSON: {"switch_id":<switch ID encoded as a string>, "old_policy":<policy name encoded as a string>}

Add a stateless rule to the ACL

HTTP verb: POST
URL: `http://127.0.0.1:8080/acl_switch/acl_rules`
JSON: {"ip_src":<IPv4 or IPv6 address encoded as a string>, "ip_dst":<IPv4 or IPv6 address encoded as a string>, "tp_proto":<Name of transport protocol encoded as a string>, "port_src":<Transport layer port number encoded as a string>, "port_dst":<Transport layer port number encoded as a string>, "policy":<policy name encoded as a string>}

Add a stateful rule to the ACL

HTTP verb: POST
URL: `http://127.0.0.1:8080/acl_switch/acl_rules/time`
JSON: {"ip_src":<IPv4 or IPv6 address encoded as a string>, "ip_dst":<IPv4 or IPv6 address encoded as a string>, "tp_proto":<Name of transport protocol encoded as a string>, "port_src":<Transport layer port number encoded as a string>, "port_dst":<Transport layer port number encoded as a string>, "policy":<policy name encoded as a string>, "time_start":<Time specified in hours and minutes in 24-hour format encoded as a string>, "time_duraction":<Time in seconds encoded as a string>}

Remove a rule from the ACL

HTTP verb: DELETE
URL: `http://127.0.0.1:8080/acl_switch/acl_rules`
JSON: {"rule_id":<rule ID encoded as a string>}

Appendix B

Screenshots of tcpdump packet captures

B.1 Sending IPv6 traffic using Scapy

Figure B.1 illustrates what happens when sending an ICMPv6 Echo Request over IPv6 (highlighted in yellow) compared to when an TCP connection is initiated using IPv6 (highlighted in red) when using `Scapy` and the `tcpdump` utility. The destination host received the TCP over IPv6 traffic (the bottom terminal window in the figure) but it did not send a TCP header back to reset the connection.

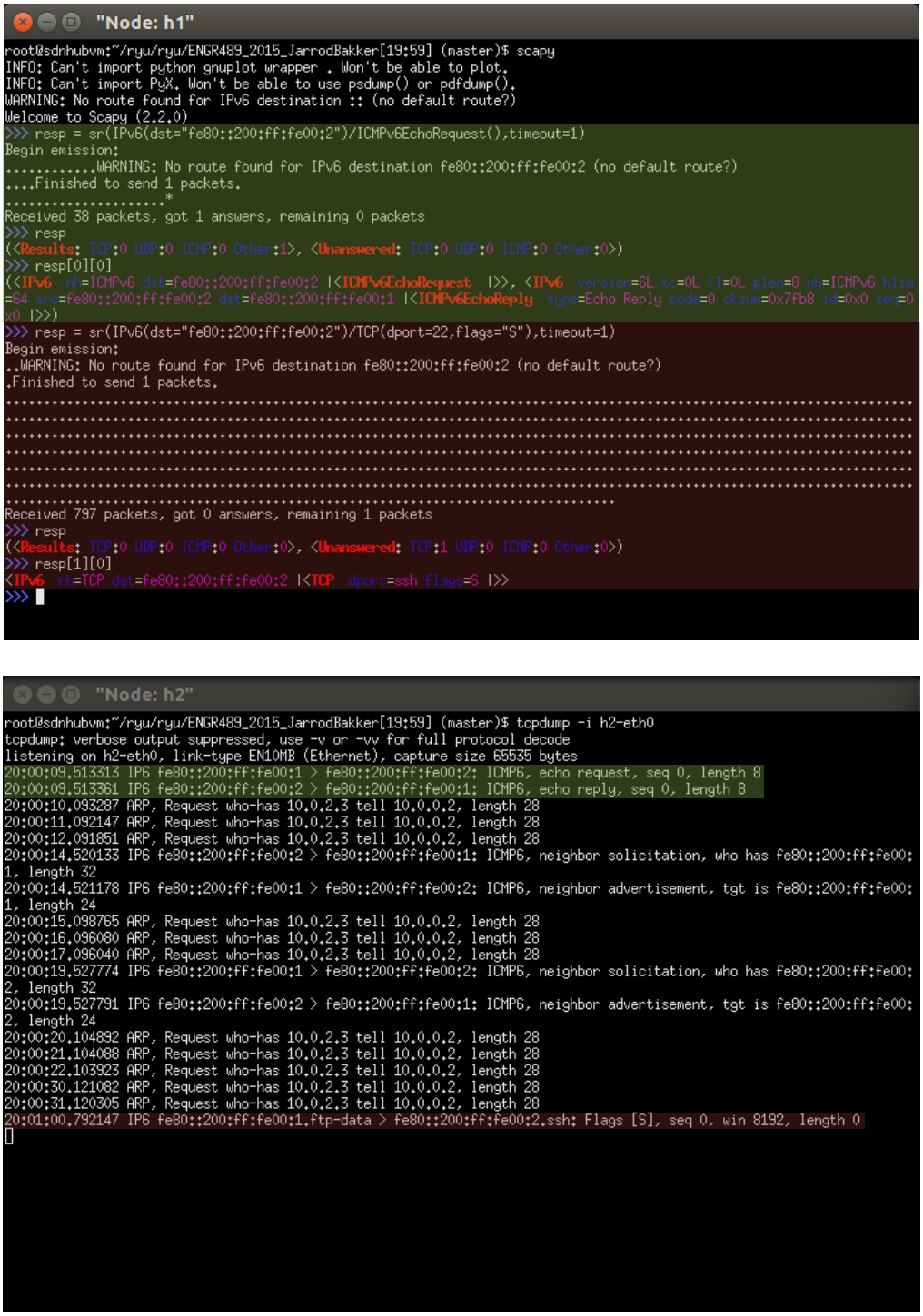

Figure B.1: tcpdump packet capture of ICMPv6 and TCP traffic sent over IPv6 using Scapy.

B.2 Verifying that traffic blocked at a switch does not reach the destination host

It may be taken for granted that flows that are configured to be blocked at the switch never reach the destination host. To verify that a blocked flow never reaches its destination a small test was run. ACLSwitch was configured with one rule to block traffic with an IPv4 source address of 10.0.0.1 and IPv4 destination address of 10.0.0.2. The network topology used was the one depicted in Figure 5.2. h1 ran `$ ping 10.0.0.2 -c 5` to send five ICMP echo requests to h2 over IPv4 while h2 ran `tcpdump -i h2-eth0` to monitor traffic.

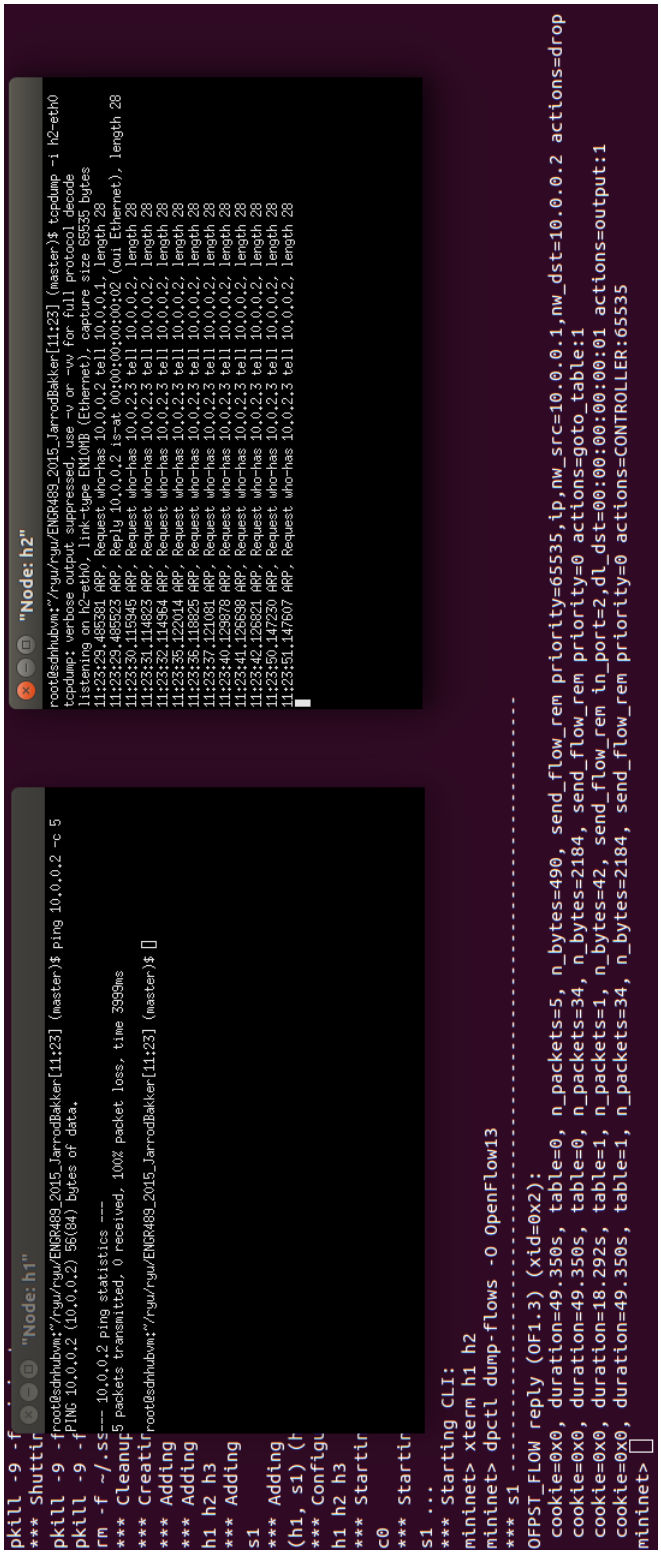

Figure B.2: tcpdump shows that traffic blocked at a switch does not reach the destination host h2.

The following observations can be draw from Figure B.2:

1. The ping output on h1 shows that none of the five ICMP echo requests sent by h1 reached h2 (100% packet loss).

2. The tcpdump output on h2 shows that no ICMP echo requests were received by h2.
3. The flow table entry that was blocking the flow (the only table entry with actions set to drop), matched five packets (see the n_packets field).

Given observations 1, 2 and 3 above, it can be concluded that the traffic sent by h1 was indeed blocked at s1.

Appendix C

Scapy test suite notes

This appendix contains details on all of the functional tests. Note that the first test was displayed in its entirety in Chapter 5. It has also been included for completeness.

C.1 Scapy Test Suites

The tests contained within each Scapy test suite were configured programmatically using the Python module of the same name (`Scapy`). They were run without the need for any other scripts or intervention besides the insertion of a `config.json` file for the ACLSwitch application. These tests verify that a single network security policy can be applied across a network topology.

Suite 1 - NoDrop

This suite aimed to verify that traffic does not blocked unintentionally at a switch when there are no rules stating that it should be blocked. The network topology used was the one depicted in Figure 5.1. Below is the list of test files used along with an elaboration:

1. NoDrop_EmptyACL - With no rules present to drop any traffic, send a range of traffic through the switch from h1 to h2 and h3.
2. NoDrop_IPv4Ping - With rules to drop all TCP and UDP over IPv4 traffic between the hosts, send an ICMP Echo Request over IPv4 from h1 to h2 and h3.
3. NoDrop_IPv4TCP - With rules to drop all UDP over IPv4 traffic between the hosts, send TCP traffic from h1 to h2 and h3 with varying source and destination ports.
4. NoDrop_IPv4UDP - With rules to drop all TCP over IPv4 traffic between the hosts, send UDP traffic from h1 to h2 and h3 with varying source and destination ports.
5. NoDrop_IPv6Ping - With rules to drop all TCP and UDP over IPv6 traffic between the hosts, send an ICMPv6 Echo Request over IPv6 from h1 to h2 and h3.

In the case of test file NoDrop_IPv4TCP the choice to block all UDP over IPv4 was made because the only other kind of traffic that can be blocked with halting communication is UDP. The same reasoning was used with the test file NoDrop_IPv4UDP and the choice to block all TCP over IPv4. The term *varying source and destination ports* refers to preconfigured selection of source and destination ports.

The tests within this suite confirmed that traffic does not get blocked accidentally at a switch. In the cases where rules were inserted into the ACL the table dumps revealed that

none of the flow table entries associated with blocking traffic were matched by any packets. Therefore it was concluded that traffic cannot be blocked accidentally.

Suite 2 - Drop All H1

This suite aimed to verify that when a range of traffic is sent from a host, it can get blocked at a switch. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a source IP address of h1. Below is the list of test files used along with an elaboration:

1. DropAllH1_IPv4All - With a rule to block all IPv4 traffic originating from h1 send ICMP Echo Requests, TCP and UDP traffic from h1 to h2 and h3.
2. DropAllH1_IPv4TCP - With a rule to block all TCP traffic originating from h1, send TCP traffic from h1 to h2 and h3 with varying source and destination ports.
3. DropAllH1_IPv4UDP - With a rule to block all UDP traffic originating from h1, send UDP traffic from h1 to h2 and h3 with varying source and destination ports.
4. DropAllH1_IPv6Ping - With a rule to block all IPv6 traffic originating from h1, send ICMPv6 Echo Requests from h1 to h2 and h3.

The tests within this suite confirmed that traffic was blocked at switch when the source IP address and transport layer protocol fields were specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that an entire type of traffic (TCP, UDP, etc) can be blocked when it originates from a specific host.

Suite 3 - Drop Select H1

This suite aimed to verify that rules could be used to block specific TCP and UDP flows at a switch. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a source IP address of h1. Below is the list of test files used along with an elaboration:

1. DropSelectH1_IPv4TCP - With rules to block specific TCP flows originating from h1, send matching TCP flows from h1 to h2 and h3.
2. DropSelectH1_IPv4UDP - With rules to block specific UDP flows originating from h1, send matching UDP flows from h1 to h2 and h3.
3. DropSelectH1_IPv6TCP_destPort - With rules to block specific TCP flows originating from h1 sent over IPv6, send matching TCP flows from h1 to h2 and h3. Note that as `Paramiko` has to be used for these tests, only the destination port can be specified in a rule.

The test files that use IPv4 first sent a series of flows where both the source and destination ports are specified. Following this, flows were sent where the destination port was specified but the source as not. The source port in this case was assigned a random (seeded) destination port between 32768 and 61000 to mimic what an Operating System Kernel typically does with outgoing connections - it assigns an ephemeral port. The last flows sent had the source port specified but the destination not. This ensured that a range of cases was covered. Care was taken with the latter flows as Scapy automatically sets the destination port to 80 if it is not specified. However this results in the flows matching a rule to block

port 80 traffic from h1. This is an interesting result as the behaviour is the same as the flow is still blocked but it was not being matched by the expected flow table entry. Therefore the latter flows were assigned a random destination port in a similar fashion to the flows with random source ports. All ACL rules filtered traffic with h1's source IP address.

The tests within this suite confirmed that traffic was blocked at a switch when the source IP address and transport layer port number fields are specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that specific TCP (over IPv4 and IPv6 when destination port is specified) and UDP (over IPv4 only) flows can be blocked when required.

Suite 4 - Drop All H1 Destination

This suite aimed to verify that when a range of traffic is sent to a specific host, it can get blocked at a switch. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a destination IP address of h1. Below is the list of test files used along with an elaboration:

1. DropAllH1Dest_IPv4All - With a rule to block all IPv4 traffic destined for h1 send ICMP Echo Requests, TCP and UDP traffic from h1 to h2 and h3.

2. DropAllH1Dest_IPv4TCP - With a rule to block all TCP traffic destined for h1, send TCP traffic from h1 to h2 and h3 with varying source and destination ports.

3. DropAllH1Dest_IPv4UDP - With a rule to block all UDP traffic destined for h1, send UDP traffic from h2 to h1 with varying source and destination ports.

4. DropAllH1Dest_IPv6Ping - With a rule to block all IPv6 traffic destined from h1, send ICMPv6 Echo Requests from h1 to h2 and h3.

When a UDP port is scanned and found to closed, an ICMP packet carrying a *Destination port unreachable* message is sent back to the source. Therefore it was not adequate for the test file DropAllH1Dest_IPv4UDP to follow the same pattern used for other tests where h1 sends to h2 and h3. To complete the test, the test file was run on h2 with traffic being directed at h1.

The tests within this suite confirmed that traffic was blocked at a switch when the destination IP address and transport layer protocol fields were specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that an entire type of traffic (TCP, UDP, etc) can be blocked when it is destined for a specific host.

Suite 5 - Drop Select H1 Destination

This suite aimed to verify that rules could be used to block specific TCP and UDP flows targeted for a specific host. The network topology used was the one depicted in Figure 5.1. All ACL rules filtered traffic with a destination IP address of h1. Below is the list of test files used along with an elaboration:

1. DropSelectH1Dest_IPv4TCP - With rules to block specific TCP flows destined for h1, send matching TCP flows from h1 to h2 and h3.

2. DropSelectH1Dest_IPv4UDP - With rules to block specific UDP flows destined for h1, send matching UDP flows from h1 to h2 and h3.

3. DropSelectH1Dest_IPv6TCP_srcPort - With rules to block specific TCP flows originating from h1 sent over IPv6, send matching TCP flows from h1 to h2 and h3.

The test file DropSelectH1Dest_IPv4UDP used the same method as DropAllH1Dest_IPv4UDP where traffic was sent h2 to h1. The test file DropSelectH1Dest_IPv6TCP_destPort had an important difference with its counterpart DropSelectH1_IPv6TCP_destPort. Because h1 initiated the request and the intention was to block the reply, the TCP source port coudl be filtered.

The tests within this suite confirmed that traffic was blocked at a switch when the destination IP address and transport layer port number fields are specified. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that specific TCP (over IPv4 and IPv6 when the source port is specified) and UDP (over IPv4 only) flows can be blocked when required.

Suite 6 - Drop All Two Host

This suite aimed to verify that rules could be used to block all traffic between two hosts when both the source and destination IP addresses are specified in a rule. The network topology used was the one depicted in Figure 5.2. All ACL rules filtered traffic with a source IP address of h1 and destination IP address of h2. Below is the list of test files used along with an elaboration:

1. DropAllTwoHostNetwork_IPv4All - With a rule to block all IPv4 traffic from h1 and destined for h2, send ICMP Echo Requests, TCP and UDP traffic to h2 from h1.

2. DropAllTwoHostNetwork_IPv4TCP - With a rule to block all TCP traffic from h1 and destined for h2, send TCP traffic to h2 from h1 with varying source and destination ports.

3. DropAllTwoHostNetwork_IPv4UDP - With a rule to block all UDP traffic from h1 and destined for h2, send UDP traffic to h2 from h1 with varying source and destination ports.

4. DropAllTwoHostNetwork_IPv6Ping - With a rule to block all IPv46traffic from h1 and destined for h2, send ICMPv6 Echo Requests to h2 from h1.

The tests within this suite confirmed that traffic was blocked at a switch when the source and destination IP addresses were specified in a rule. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore was be concluded that all traffic can be blocked between two specific hosts.

Suite 7 - Drop Select Two Host

This suite aimed to verify that rules could be used to specific TCP and UDP flows between two hosts when both the source and destination IP addresses are specified in a rule. The network topology used was the one depicted in Figure 5.2. All ACL rules filtered traffic with a source IP address of h1 and destination IP address of h2. Below is the list of test files used along with an elaboration:

1. DropSelectTwoHostNetwork_IPv4TCP - With rules to block specific TCP flows when sent from h1 and destined for h2, send matching TCP flows from h1 to h2.

2. DropSelectTwoHostNetwork_IPv4UDP - With rules to block specific UDP flows when sent from h1 and destined for h2, send matching UDP flows from h1 to h2.

3. DropSelectTwoHostNetwork_IPv6TCP_destPort - With rules to block specific TCP flows when sent from h1 and destined for h2 over IPv6, send matching TCP flows from h1 to h2.

The tests within this suite confirmed that traffic was blocked at a switch when the source and destination IP addresses, and source and destination transport layer port numbers were specified in a rule. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that specific TCP (over IPv4 and IPv6 when the source port is specified) and UDP (over IPv4 only) flows can be blocked between two specific hosts.

Suite 8 - Drop H1 Same Rule Different Switch

This suite aimed to verify that a single network security policy could be enforced over an entire network topology. The network topology used was the one depicted in Figure 5.3. To test the coverage of a policy yet still allow h1 to send traffic, the source address field for a rule was wild-carded. Therefore all ACL rules filtered traffic with a destination IP address of h1. Note that UDP cannot be tested in this way as the destination host replies with ICMP packets and not with UDP. Below is the list of test files used along with an elaboration:

1. DropH1SameRuleDiffSwitch_IPv4Ping - With a rule to block all IPv4 traffic destined for h1, send an ICMP Echo Request from h1 to h2, h3 and h4.

2. DropH1SameRuleDiffSwitch_IPv4TCP - With rules to block specific TCP traffic destined for h1, send matching TCP flows from h1 to h2, h3 and h4.

3. DropH1SameRuleDiffSwitch_IPv6Ping - With a rule to block all IPv6 traffic destined for h1, send an ICMPv6 Echo Request from h1 to h2, h3 and h4.

4. DropH1SameRuleDiffSwitch_IPv6TCP_destPort - With rules to block specific TCP traffic destined for h1 over IPv6, send matching TCP flows over IPv6 from h1 to h2, h3 and h4.

The tests within this suite confirmed that traffic was blocked at all switches when the destination IP address was specified in a rule. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. As the previous suites showed that a range of different rules could be enforced at a switch, it holds that the same range of rules could also be enforced on multiple switches. Therefore it was concluded that a single network security policy could be enforced over an entire network topology.

C.2 Rule Removal Test

The purpose of this test was to verify that rules could be removed from the ACL and the changes could be observed in the network. This is an important feature to test as it is not uncommon for network security policies to change over time. The network topology used was the one depicted in Figure 5.1.

The test was performed as follows:

1. It was ensured that that the `config.json` file was blank before starting ACLSwitch. ACLSwitch was started following this.

2. Mininet was started with the topology set to the one depicted in Figure 5.1.

3. A series of TCP over IPv4 flows were sent from h1 to h2 and h3 to ensure that all traffic could flow uninhibited.

4. A set of rules matching the TCP flows from before were created using the REST API.

5. A series of TCP over IPv4 flows were sent from h1 to h2 and h3 to ensure that all traffic was being blocked.

6. The rules that were created before were removed using the REST API.

7. A series of TCP over IPv4 flows were sent from h1 to h2 and h3 to ensure that all traffic could flow uninhibited again.

The test found found that rules could be removed from the ACL, resulting in network security policy that changed over time. As the test progressed, h1's connectivity changed between being able to contact its neighbours to not being able to at all. Producing table dumps before and after each communication attempt revealed that the changes in the policy were being reflected in the switch's flow table. Therefore it was concluded that rules can be successfully removed from the ACL with the changes taking effect in the network.

C.3 Policy domain tests

The purposes of these tests were to verify that different policy domains could be enforced at different switches. The network topology used for both tests was the one depicted in Figure 5.3. In the context of the tests below, s1 is a switch that connects two networks with one another.

Policy Assignment

The purpose of this test was to verify that a policy could be assigned to s1 that would stop a host from communicating with hosts outside of its network. The test did not send traffic with various source and destination port numbers as the previous tests proved that variation could be enforced at a single switch. Below is the list of test files used along with an elaboration:

1. PoliciesDropAllH1_IPv4All - With a rule to block all IPv4 traffic from h1 deployed on s1, send ICMP Echo Requests, TCP and UDP traffic from h1 to h2, h3 and h4.

2. PoliciesDropAllH1_IPv6Ping - With a rule to block all IPv6 traffic from h1 deployed on s1, send ICMPv6 Echo Requests from h1 to h2, h3 and h4.

3. PoliciesDropAllH3_IPv4All - With a rule to block all IPv4 traffic from h3 deployed on s1, send ICMP Echo Requests, TCP and UDP traffic from h3 to h1, h2 and h4.

4. PoliciesDropAllH3_IPv6Ping - With a rule to block all IPv6 traffic from h3 deployed on s1, send ICMPv6 Echo Requests from h3 to h1, h2 and h4.

The test found that h1 could only communicate with h2 in tests 1 and 2 and h3 could only communicate with h4 in tests 3 and 4. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that switches can enforce different policies when they are assigned to them.

Policy Removal

The purpose of this test was to verify that a policy could be removed from a switch and the changes could be observed in the network. The test did not send traffic with various source and destination port numbers as the previous tests proved that variation could be enforced at a single switch.

The test was performed as follows:

1. The `config.json` file was set to contain a policy that would later be assigned to and removed from s1, along with a rule to block IPv4 traffic from h1 that is part of the new policy. ACLSwitch was started following this.
2. Mininet was started with the topology set to the one depicted in Figure 5.3.
3. A series of IPv4 flows were sent from h1 to h2, h3 and h4 to ensure that all traffic could flow uninhibited.
4. The new policy was assigned to s1 using the REST API.
5. A series of IPv4 flows were sent from h1 to h2, h3 and h4. The expected result was that h1 should only be able to communicate with h2.
6. The new policy was removed from s1 using the REST API.
7. A series of IPv4 flows were sent from h1 to h2, h3 and h4 to ensure that all traffic could flow uninhibited again.

The test found that h1's connectivity with other hosts matched the policy domains assigned to s1 at different points in time. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that a policy assignment can be removed from a switch.

C.4 Stateful policy tests

The purpose of these tests were to verify the the correct operation of stateful policies. The tests did not use a variety of traffic types (i.e. different source and destination port combinations) as the stateful implementation builds upon what has already been tested and verified in the stateless implementation.

Scheduling

A series of tests were used to ensure that rules can be scheduled in the correct position within the queue. Each test file tested a different aspect, such as in what order are rules added and the time relative to the current system time. The script inserts rules relative to the current system time that the test file was run on. Therefore the tests can be executed at any time of the day and the ordering of the rules should remain consistent. Note that *before*

the current system time means that the rule should be scheduled for the next day, however the hours and minutes of the new rule alone place it in front of the current time. For example, if the current time wass 1400 and a rule needed to be scheduled for 1300 then it should be scheduled for the next day. Below is the list of test files used along with an elaboration:

1. TimeSchedule_InOrder_AfterCurTime - Schedule a set of rules in order to be dispatched after the current system time.
2. TimeSchedule_InOrder_BeforeCurTime - Schedule a set of rules in order to be dispatched before the current system time.
3. TimeSchedule_OutOrder_AfterCurTime - Schedule a set of rules out of order to be dispatched after the current system time.
4. TimeSchedule_OutOrder_BeforeCurTime - Schedule a set of rules out of order to be dispatched before the current system time.
5. TimeSchedule_ReverseOrder_AfterCurTime - Schedule a set of in reverse order to be dispatched after the current system time.
6. TimeSchedule_ReverseOrder_BeforeCurTime - Schedule a set of in reverse order to be dispatched before the current system time.
7. TimeSchedule_NoOrder - Schedule a set of rules in no particular order to be dispatched before and after the current system time.

The test found that rules can be scheduled the correct order. The tests covered a range of different scenarios to ensure correctness. Therefore it was concluded that stateful policies rules can be correctly scheduled.

Dispatch

Two scenarios were used to verify that stateful policies can be dispatched at their correct times and be enforced for their designated period of time. The first test aimed to confirm that a single stateful policy could be dispatched correctly for a given period of time. The second aimed to confirm that multiple stateful policies could be scheduled for dispatch at the same time as well as at different times. The network topologies used for the first and second tests are depicted in figures 5.2 and 5.4 respectively. Below is the list of test files used along with an elaboration:

1. TimeDispatch_TwoHost_DropH1 - Schedule a rule to block traffic between h1 and h2 for 1 minute.
2. TimeDispatch_FourHost_DropH1 - Schedule one rule each to block traffic sent from h1 to h2 and h3 for 1 minute starting at the same time, and schedule another rule to block traffic sent from h1 to h4 for 1 minute starting immediately after the first two rules expire.

The test found that the stateful policies dropped packets within the correct time frames. The table dumps for each test file confirmed this as each flow table entry associated with blocking a flow of traffic was matched the expected number of times. Therefore it was concluded that stateful policies can be correctly enforced.