Towards Smarter SDN Switches: Revisiting the Balance of Intelligence in SDN Networks

Jonathan Weekes

School of Computing and Communications

Lancaster University, UK

Thesis submitted for the degree of Doctor of Philosophy

September 2019

Abstract

Software Defined Networks (SDNs) represent a new model for building networks, in which the control plane is separated from the forwarding plane, allowing for centralised, fine grained control of traffic in the network. The benefits of SDN range widely from reducing operational costs of networks to providing better Quality of Service guarantees to its users. Its application has been shown to increase the efficiency of large networks such as data centers and improve security through Denial of Service mitigation systems and other traffic monitoring efforts.

While SDN has been shown to be highly beneficial, some of its core features (e.g separation of control and data planes and limited memory) allow malicious users to carry out Denial of Service (DoS) attacks against the network, reducing its availability and performance. Denial of Service attacks are explicit attempts to prevent legitimate users from accessing a service or resource. Such attacks can take many forms but are almost always costly to its victims, both financially and reputationally. SDN applications have been developed to mitigate some forms of DoS attacks aimed at traditional networks however, its intrinsic properties facilitate new attacks.

We investigate in this thesis, the opportunity for such Denial of Service attacks in more recent versions of SDN and extensively evaluate its effect on a legitimate user's throughput. In light of the potential for such DoS attacks which specifically target the SDN infrastructure (controller, switch flow table etc), we propose that increasing the intelligence of SDN switches can increase the resilience of the SDN network by preventing attack traffic from entering the network at its source. To demonstrate this, we put forward in this thesis, designs for an intelligent SDN Switch and implement two additional functionalities towards realising this design into a software version of the SDN switch. These modules allow the switch to efficiently handle high control plane loads, both malicious and legitimate, to ensure the network continues to provide good service even under such circumstances. Evaluation of these modules indicate they effectively preserve the performance of the network under under high control plane loads far better than unmodified switches, with no notable drawbacks.

Declaration

I declare that the work in this thesis has not been submitted for a degree at any other university, and that the work is entirely my own.

Jonathan Weekes
September 2019

Acknowledgments

I would like to express my deepest thanks to my supervisor Dr Matthew Broadbent. Over the past year his guidance, patience and motivational speeches have been invaluable. A special thanks as well to Dr Shishir Nagaraja for working with me particularly during the initial stages of my degree and to Dr Nicholas Race for his support and guidance. I would also like to thank my family (Dad, Mom, Niks, Rachel and Josef) and those friends who have been especially supportive of me through this time: Abigail, Anysha and Nastassia. Most importantly, I thank God for getting me through this.

Contributing Publications

Weekes, Jonathan, and Shishir Nagaraja. “Controlling Your Neighbour’s Bandwidth for Fun and for Profit.” Cambridge International Workshop on Security Protocols. Springer, Cham, 2017.

Contents

1 Introduction **24**

1.1 Denial of Service 25

1.2 Software Defined Networks 26

1.3 DoS Attacks in SDN 27

1.4 Switch Based Flow Rule Eviction 28

1.5 Thesis Aims and Contributions 28

1.5.1 Analysis of initial steps toward increasing switch intelligence . 28

1.5.2 Further Switch Intelligence Augmentation 29

1.6 Thesis Structure 30

2 Thesis Background **32**

2.1 Introduction 32

2.2 The Rise of Networking and Packet Routing 33

2.3 Programmable Networks 35

2.4 Software Defined Networking and OpenFlow 37

2.4.1 The OpenFlow SDN Architecture 38

2.4.2 OpenFlow- A Protocol for Programmable Networks 42

2.4.3 Beyond Openflow: Stateful SDN Data Planes 45

2.5 Denial of Service Attacks and Network Security 47

2.6 Software Defined Networking Vulnerabilities 49

2.7 Denial of Service Attacks in Software Defined Networks 50

2.7.1 SDN DoS Attack Definition: 51

2.7.2 Reconnaissance . . . 52
2.7.3 Controller . . . 52
2.7.4 Controller Switch Communication Channel . . . 53
2.7.5 Switch OpenFlow Agent . . . 53
2.7.6 Switch Flow table . . . 54
2.8 Denial of Service Controller Based Solutions . . . 55
2.8.1 Monitoring . . . 55
2.8.2 Machine Learning . . . 56
2.8.3 Thresholds . . . 57
2.8.4 Queues . . . 58
2.8.5 Other Mechanisms . . . 59
2.9 Denial of Service Switch Based Solutions . . . 60
2.9.1 TCP Proxies . . . 61
2.9.2 Duplicate Flow Requests . . . 62
2.9.3 Entropy Based Detection . . . 62
2.10 Distributed Control Plane and Load Balancing . . . 65
2.11 Summary . . . 67
2.11.1 Moving Forward . . . 68

3 An Analysis of Flow Eviction Strategies 70

3.1 Introduction . . . 70
3.1.1 Table Overflow . . . 71
3.1.2 Switch Based Flow Rule Eviction . . . 72
3.1.3 "First Packet" Delays . . . 73
3.2 Analysis of the Switch Flow Rule Eviction Attack Vector . . . 74
3.2.1 Analysis Of The Effects Of Delayed Packets . . . 75
3.2.2 Attacker Model . . . 79
3.2.3 Attacking the FIFO Policy: The Spray Attack . . . 80
3.2.4 Spray Attack Proof of Concept . . . 81

3.2.5 Limitations . . . 82
3.3 Alternative Policies for Eviction . . . 83
3.3.1 Least Recently Used . . . 83
3.3.2 Least Frequently Used . . . 84
3.3.3 Random . . . 84
3.4 Attacking Volume Based Eviction Policies: The Clog Attack . . . 85
3.5 Resilience Analysis of various eviction policies . . . 86
3.5.1 Attack Power . . . 89
3.5.2 Number of Attackers . . . 93
3.5.3 Flow Table Size . . . 96
3.5.4 Background Traffic . . . 100
3.6 Implications . . . 104
3.6.1 QoS of Streaming Applications . . . 104
3.6.2 Critical Infrastructure . . . 105
3.7 Summary . . . 107

4 Increasing SDN Switch Intelligence 109

4.1 Introduction . . . 109
4.1.1 Benefits of Switch Intelligence . . . 110
4.2 Intelligent Switch Design . . . 110
4.2.1 Switch Flow Rule Eviction . . . 113
4.2.2 Intelligent Flow Request Forwarding . . . 113
4.2.3 Controller Malfunction Detection . . . 114
4.2.4 Implementation . . . 115
4.3 Load Distribution . . . 115
4.3.1 Taxonomy . . . 116
4.3.2 OpenFlow Support . . . 118
4.3.3 Switch Based Load Distribution Design . . . 120
4.3.4 Summary . . . 126

4.4 Malicious Packet_in Classification and Filtering 126
4.4.1 Adversarial Model 128
4.4.2 Taxonomy 129
4.4.3 Random Forest Classification 135
4.4.4 Attribute Gathering 136
4.4.5 Switch Module Design 139
4.4.6 Switch Implementation 142
4.4.7 Summary 143
4.5 Conclusion 144

5 Evaluation of Additional Switch Intelligence 145

5.1 Introduction 145
5.2 Load Distribution 146
5.2.1 Controller Performance 147
5.2.2 CPU Load Tests 152
5.2.3 Latency Tests 155
5.2.4 Processing Time 159
5.2.5 Summary 160
5.3 Malicious Packet_in Filter 160
5.3.1 Passive Evaluation 161
5.3.2 Active/Live evaluation 164
5.3.3 Summary 174
5.4 Conclusion 174

6 Conclusion 176

6.1 Thesis Contributions 176
6.1.1 Analysis of initial steps toward increasing switch intelligence . 176
6.1.2 Further Switch Intelligence Augmentation 178
6.1.3 Research Impacts 180
6.1.4 Summary 181

6.2 Future Work 181
6.2.1 Switch Flow Rule Eviction 182
6.2.2 Load Distribution 183
6.2.3 Malicious Packet_in Filter 183
6.2.4 Further Switch Intelligence 185
6.3 Concluding Remarks 186

Bibliography **187**

Appendix **205**

A Results of varying the attack parameters 205
A.1 Results of Varying the number of flushes in the Spray Attack . 205
A.2 Results of Varying the number of malicious flow rules in the Clog Attack 206
B Results of varying the number of attackers 208
B.1 Results of Varying the number of Attackers in the Spray Attack 208
B.2 Results of Varying the number of attackers in the Clog Attack 209
C Results of varying the flow table size 211
C.1 Results of Varying the size of the flow table in the Spray Attack 211
C.2 Results of Varying the size of the flow table in the Clog Attack 212
D Results of varying the surrounding network traffic during the attack . 213
D.1 Results of Varying the surrounding network traffic in the Spray Attack 213
D.2 Results of Varying the surrounding network traffic in the Clog Attack 215

List of Figures

2.1 SDN Architecture 39

2.2 SDN Flow Table Example 41

3.1 Network Setup 75

3.2 Bandwidth With and Without Matching Flow Rules 76

3.3 Frequent Evictions with varying controllers 78

3.4 Spray attack effects 82

3.5 Simulation of the Effects of Frequent Flow Rule Evictions 88

3.6 Bandwidth Changes under Increasing Spray Attack Power 90

3.7 Bandwidth Changes under Increasing Clog Attack Power 92

3.8 Bandwidth Changes with increasing number of Spray Attackers . . . 95

3.9 Bandwidth Changes with increasing number of Clog Attackers 96

3.10 Bandwidth Changes with varying Flow Table size: Spray Attack . . . 98

3.11 Bandwidth Changes with varying Flow Table size: Clog Attack . . . 99

3.12 Bandwidth Changes under varying Background Traffic: Spray Attack 102

3.13 Bandwidth Changes under varying Background Traffic: Clog Attack . 103

4.1 Intelligent SDN Switch Design 112

4.2 Switch Based Flow Request Distribution 121

4.3 Malicious Packet_in Filter 128

4.4 Flow Distribution of various datasets 138

5.1 Network Setup 147

5.2 Percentage of Packet_ins processed per controller (Homogeneous Controller Pool) . . . 149
5.3 Percentage of Packet_ins processed per controller (Non Homogeneous Controller Pool) . . . 150
5.4 CPU Load per controller (Homogeneous Controller Pool) . . . 153
5.5 CPU Load per controller (Non Homogeneous Controller Pool) . . . 154
5.6 Latency of Pings with multiple controllers (Homogeneous Controller Pool) . . . 156
5.7 Latency of Pings with multiple controllers (Non Homogeneous Controller Pool) . . . 157
5.8 Network Setup . . . 165
5.9 Network Connectivity under attack . . . 166
5.10 Network Bandwidth under attack . . . 167
5.11 Attack Patterns . . . 169
5.12 Attack Flow Requests Serviced by the controller . . . 169
5.13 Connectivity of the Attacking Host around attacks . . . 171
5.14 Switch and Controller CPU Usage . . . 172

List of Tables

1 Table of Acronyms 23

3.1 Bandwidth changes with variable rule 93

3.2 Background Traffic Datasets 101

4.1 Dataset Attributes 137

5.1 Round Robin Load Balancer Processing Times 159

5.2 Round Robin Load Balancer Processing Times 159

5.3 Load Balancer Processing Times 159

5.4 Random Forest Accuracy Evaluation 163

5.5 Effect of the Number of Decision Trees on Random Forest Accuracy . 163

5.6 Weighting of Attributes in Classification 164

5.7 Times taken for packet_in classification 173

1a Spray Attack Power under **FIFO** eviction 205

1b Spray Attack Power under **Random** eviction 205

1c Spray Attack Power under **LFU** eviction 206

1d Spray Attack Power under **LRU** eviction 206

2a Clog Attack Power under **FIFO** eviction 206

2b Clog Attack Power under **Random** eviction 207

2c Clog Attack Power under **LFU** eviction 207

2d Clog Attack Power under **LRU** eviction 207

2e Clog Attack Power under **LFU** eviction with last rule packet rates varied . . . 208
3a Increasing Spray Attackers under **FIFO** eviction . . . 208
3b Increasing Spray Attackers under **Random** eviction . . . 208
3c Increasing Spray Attackers under **LFU** eviction . . . 209
3d Increasing Spray Attackers under **LRU** eviction . . . 209
4a Increasing Clog Attackers under **FIFO** eviction . . . 209
4b Increasing Clog Attackers under **Random** eviction . . . 210
4c Increasing Clog Attackers under **LFU** eviction . . . 210
4d Increasing Clog Attackers under **LRU** eviction . . . 210
5a Spray Attack on Varying Flow Table Sizes: **FIFO** eviction . . . 211
5b Spray Attack on Varying Flow Table Sizes: **Random** eviction . . . 211
5c Spray Attack on Varying Flow Table Sizes: **LFU** eviction . . . 211
5d Spray Attack on Varying Flow Table Sizes: **LRU** eviction . . . 212
6a Clog Attack on Varying Flow Table Sizes: **FIFO** eviction . . . 212
6b Clog Attack on Varying Flow Table Sizes: **Random** eviction . . . 212
6c Clog Attack on Varying Flow Table Sizes: **LFU** eviction . . . 213
6d Clog Attack on Varying Flow Table Sizes: **LRU** eviction . . . 213
7a Spray Attack with varying BG traffic: **FIFO** eviction . . . 213
7b Spray Attack with varying BG traffic: **Random** eviction . . . 214
7c Spray Attack with varying BG traffic: **LFU** eviction . . . 214
7d Spray Attack with varying BG traffic: **LRU** eviction . . . 214
8a Clog Attack with varying BG traffic: **FIFO** eviction . . . 215
8b Clog Attack with varying BG traffic: **Random** eviction . . . 215
8c Clog Attack with varying BG traffic: **LFU** eviction . . . 215
8d Clog Attack with varying BG traffic: **LRU** eviction . . . 216

Table of Acronyms

Acronym	Meaning
AS	Autonomous System
ASIC	Application-Specific Integrated Circuits
BGP	Border Gateway Protocol
CPS	Cyber Physical Systems
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency (US)
DDoS	Distributed Denial of Service
DoS	Denial of Service
EGP	Exterior Gateway Protocol
EIGRP	Enhanced Interior Gateway Routing Protocol
FIFO	First In First Out
ForCES	Forwarding and Control Element Separation
FPS	Flushes Per Second
GB	Giga Bytes
GSMP	General Switch Management Protocol
ICS	Internet Connection Sharing
ICMP	Internet Control Message Protocol
IGRP	Interior Gateway Routing Protocol
IP Address	Internet Protocol Address
IS-IS	Intermediate System to Intermediate System
ISP	Internet Service Provider
KB	Kilo Bytes

LFU	Least Frequently Used
LRU	Least Recently Used
MAC	Address Media Access Control Address
MB	Mega Bytes
MDA	Mean Decrease in Accuracy
ML	Machine Learning
NETCONF	Network Configuration Protocol
OF	OpenFlow
OFA	OpenFlow Agent
OS	Operating System
OSPF	Open Shortest Path First
OvS	Open VSwitch
POW	Proof of Work
PPS	Packet ins Per Second
QoS	Quality of Service
RAM	Random Access Memory
NIB	Network Information Base
REST API	Representational State Transfer Application Program Interface
RFE (Switch)	Random Forest Enabled (Switch)
RIP	Routing Information Protocol
RTT	Round Trip Time
SDN	Software Defined Network
SNMP	Simple Network Management Protocol
SOM	Self-Organizing Map

STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege
SVM	Support Vector Machine
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TLRU	Time AwareLeast Recently Used
UCLA	University of California, Los Angeles
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
WSAN	Wireless Sensor and Actuator Network

Table 1: Table of Acronyms

Chapter 1

Introduction

Computer networks fundamentally enable the transfer of information. They have become integral in both business and leisure in our modern society. From small home or office networks to the Internet, they allow us to access information and entertainment and allow us to communicate from any location.

From a business standpoint, computer networks (in particular the Internet) have become essential to remaining competitive in any industry. Employees everywhere require the ability to instantly contact suppliers and customers. Management requires the latest information on market status and company performance at their fingertips to make informed decisions. Businesses are also able to reduce costs and expand their reach by replacing face to face meetings with communication over the internet for both purchases and sales.

A myriad of industries are built on computer networks. The rise of E-commerce, an industry estimated to be worth around 500bn pounds per year from 2011 to present day [1], is built entirely on the concept of allowing remote access to desired items or services through the Internet. Connectivity has become intrinsic to the healthcare industry which must ensure patient records are transferred between sites in a timely manner and much of academia's research and collaboration would be significantly more difficult without the Internet, just to mention a few. Computer networks enable people to work and play with an unparalleled amount of convenience and efficiency.

The quality of service provided by a network is a large factor considered in im-

plementing a network. The network must increase convenience or provide some business edge to its users. The users therefore specify their network requirements with the expectation that it will provide them with a reasonable level of service enabling them to carry out their tasks. [2] discusses the necessity of concise and accurate specification of requirements and outlines metrics for reasonable Quality of Service (QoS) in some popular network applications.

Network performance can be measured in a number of different ways. One of the primary measurements used is the throughput/bandwidth [3]. This provides an idea of the amount of data per second that can be sent across the network or a network connection (e.g 10MB/s or 1GB/s). It is the most understandable measurement to general consumers and can provide a good idea of the service that can be expected (other measurements such as Jitter or Error Rate [2] are also used among the more technically minded however, for the purposes of this thesis, we focus on the throughput measurement). As networks evolve, the same metrics of performance and convenience continue to be used and these “evolved” forms of networks still have a fundamental requirement to provide good service.

1.1 Denial of Service

Information Systems security is built around the core principles of Confidentiality (only authourized users have access to the Information System), Integrity (the information in the system is accurate and can only be changed by authourized individuals) and Availability (the system remains available for use to authourized users) [4]. Denial of Service (DoS) attacks seek to violate the Availability principle of security by preventing legitimate users from accessing to the system. DoS attacks are a major hindrance to network performance and involve an attacker performing some action with the express purpose of reducing the quality of service received by other users of the network. Motivations for this may range from a desire to earn respect to financial gain or political reasons [5]. In each case, the aim is to inflict damage on the victim and the ability to refuse access to an opponent's resource at a critical time can be a powerful tool. Attacks can be performed through a range of methods from simply bombarding a host or link with packets in an attempt to use up as much of the resource as possible or more sophisticated attacks involving holding onto a resource (e.g a TCP connection) to ensure other network users are

unable to connect to it.

DoS attacks have been prevalent in networks since the late 1990s and continue to bring severe consequences for their victims today. As time progresses, the attacks have grown in number and sophistication. Attackers now often use a number of hosts under their control to perform the attack in what is called a Distributed DoS (DDoS) attack and over a six month period starting at the end of 2017 until early 2018, there were 7,822 DDoS attacks recorded by Akamai, a security company monitoring the internet [6]. These attacks in particular have been facilitated by the rise of insecure smart devices connected to the internet. Attackers take over such devices and use them to generate traffic in their attacks.

In whatever form it takes, an effective DoS attack can prove costly to its victims as whichever service they relied on within their network is temporarily unavailable. This inevitably costs business large amounts of money as one of their core functionalities has failed. In other institutions this can cost critical time or even lives. In 2014, the average cost to a business of a DDoS attack was found to be $40000 per hour and 49% of the attacks reported there lasted between 6 and 24 hours [7]. Such attacks also cause significant reputational damage to a business as the slow service caused by DDoS attacks makes their website (and by extension their brand) appear unreliable.

1.2 Software Defined Networks

Software Defined Networks (SDNs) are a relatively new paradigm within the field of Computer Networking in which the forwarding plane and control plane are separate. Traditional computer networks have both planes tightly coupled in the network switches. The switches forward the packets to their appropriate destinations using various protocols within the switches to determine where the packets should be forwarded. By contrast, SDNs allow switches to retain their forwarding capabilities but place the decision-making functionalities of the network (which decide how packets get from their source to destination) into a separate entity called the Controller.

Since its conception, SDNs, along with its defacto protocol OpenFlow (OF), have been shown to provide many benefits through their ease of use, fine grained ability

to control and monitor flows and the ability to automate control through the use of a diverse range of controller applications. This has given rise to many security and performance enhancing applications which allowed for a more dynamic and versatile network than traditional deployments. However, due to the lack of autonomy (defined here as the ability to make decisions independent of other entities) of the switches, any packet for which the switch does not have a matching flow rule must be forwarded to the controller for instruction (a flow request).

1.3 DoS Attacks in SDN

Despite these benefits, this new network paradigm presents its own challenges which threaten to compromise the network security. We define security as the ability to preserve the Confidentiality, Integrity and Availability security principles within the network. Denial of Service attacks challenge the Availability principle in particular, making the network unavailable or available at a severely reduced Quality of Service to its users. SDNs intrinsically bring about new and unique vectors of attack.

The centralised controller has been identified as a central point of failure which could cripple the network. This central controller can be subjected to a range of attacks, particularly DoS. A high number of flow requests can quickly overload the controller (Controller Saturation) and failure of the controller means no further traffic can be routed with the network brought to a standstill until the controller is brought back online or the attack ceases. Additionally, switches have limited memory for its flow tables, which store information (flow rules) about where to forward packets. These tables can therefore easily be filled (Table Overflow). If no new rules can be added to the switch flow table, no new traffic can be routed through the network. Research has highlighted how a malicious user can attack both these components to degrade service in the network.

Within the SDN research community several solutions have been put forth to mitigate Controller Saturation and Switch Flow Table Overflow including controller distribution, queuing and rate limitations. We explore these solutions later in Sections 2.8-2.10 and look more critically at some of their shortcomings in Chapter 4.

1.4 Switch Based Flow Rule Eviction

To help mitigate the Table Overflow issue, SDN developers have implemented a configuration which allows the switch to evict a flow rule to allow a new one to be inserted if the flow table is full. Previously, the switch would respond to the controller with a "Table Full" error message if the controller attempted to insert a new flow rule into a full table and the switch would drop the flow. The controller could then remove a rule from the switch and then install the new flow rule, however this put additional strain on the controller requiring it to perform a number of other actions. Allowing the switch to automatically evict flow rules marks a change in the SDN paradigm. By increasing the intelligence of the switch, it is no longer a "dumb forwarding device" but is able to make decisions itself and contribute to the performance of the network.

1.5 Thesis Aims and Contributions

This thesis proposes that increasing the intelligence of the SDN switch can improve the resilience of the network. We define "switch intelligence" as the ability of the switch to perform tasks autonomously without dependence on the controller. "Resiliency" is defined in this thesis as the ability of the network to provide and maintain an acceptable level of service in the face of various faults and challenges to normal operation [8]. To demonstrate this, this thesis first analyses initial steps towards increasing the switch intelligence and then proposes further augmentation to the switch's intelligence geared towards achieving the goal of a resilient SDN network. To give evidence for the merit of this proposal, extensive evaluations are performed on both the initial analysis and further augmentations. The thesis contributions are summarised as follows:

1.5.1 Analysis of initial steps toward increasing switch intelligence

1.5.1.1 Analysis of flow rule eviction vulnerabilities

Newer SDN implementations reduce the switch's dependency on the control plane by allowing it to autonomously select a flow rule in its flow table for eviction in favor

of a new flow rule. In this thesis, we propose that while enabling the switch to evict a rule can be beneficial, its current implementation actually opens a new vector for attacking the network. This thesis provides an analysis of this vector and the effects of potential DoS attacks aimed at the SDN switch through this vulnerability.

1.5.1.2 Evaluation of alternative flow rule eviction implementations

Having demonstrated the potential threats in the current implementations of switch based flow rule eviction, this thesis aims to examine the resiliency of alternative flow rule eviction policies which could potentially be implemented in place of the current implementation. Taking cues from the cache-replacement problem of CPUs (Central Processing Unit), the thesis will provide an evaluation of several other potential flow eviction policies to determine which, if any, are more resistant to the attacks presented, enabling the switch to retain its newly introduced autonomy while closing the vector for attack.

1.5.2 Further Switch Intelligence Augmentation

Switch based flow rule eviction represents an increase of the switch intelligence, defining intelligence as the ability to autonomously perform tasks. By allowing it to make decisions and perform actions based on these decisions, it is no longer a "dumb forwarding device". This thesis aims to build on this concept of switch intelligence. We propose that instead of making the network intelligence mutually exclusive, concentrating it in the control plane, increasing the intelligence of the switch can improve the resilience of the network, helping to protect the network core from attacks. In keeping with this, we propose high-level designs for an Intelligent SDN switch. In support of this design, we implement the following features which aim to address the problem of high control plane loads in SDN networks.

1.5.2.1 Controller Load Distribution

The thesis proposes to enable the switch to distribute its control plane load among multiple controllers in the control plane, ensuring that no single controller is overwhelmed with requests from the switch while others are underutilised. This thesis will describe, implement and evaluate the effects of two load balancer designs built directly into the switch that will allow the switch to select which controller it sends requests to in order to optimise the service received.

1.5.2.2 Filtration of DoS attack packets

Instead of blindly forwarding every request to the controller, this thesis proposes that the switch should differentiate and drop malicious requests aiming to overload the controller. This thesis will describe, implement and evaluate the effects of a filter in the switch which enables it to protect the control plane from such malicious requests.

1.6 Thesis Structure

This thesis is structured into six individual chapters. Following this introduction (Chapter 1), we describe in Chapter 2 the rise of SDN from the first discussions on a separated forwarding and control planes to its modern implementation today. We discuss the concept and history of Denial of Service attacks on computer networks and how they have been adapted to cause damage to the various components of SDN networks as well as some solutions which have been proposed to mitigate these attacks.

The following chapter, Chapter 3, describes an in depth analysis performed on initial attempts to restore some autonomy to the switch by way of switch flow rule eviction. We also evaluate alternative implementations for switch flow rule eviction to determine which provides most resiliency for the SDN network.

In Chapter 4, we continue to explore the potential benefits of increased intelligence in the switch, proposing further tasks it can perform autonomously. We outline a design for an intelligent switch which offers the network increased resilience against DoS. We also implement modules in the switch aimed at realising some of these high level designs- providing better performance and security in the SDN network. These modules are implemented directly into the SDN switch allowing it to autonomously distribute legitimate flow requests among the available controllers and filter malicious flow requests to protect the control plane.

In Chapter 5, we present a detailed evaluation of the implemented modules. We evaluate both the load distribution and the flow request filtering, with respect to the increase in network performance and security they provide, comparing their outputs

to an unmodified version of the SDN switch. We use this evaluation to provide concrete evidence of the benefits of increasing the switch intelligence to aid in the protection and performance of the network.

Finally, we outline future work which could be done in the area of switch intelligence and present the main contributions and impacts of our work in this thesis in Chapter 6.

Chapter 2

Thesis Background

2.1 Introduction

Computer Networking is a vast and complex topic with many challenges which have been visited and revisited over several decades. In this chapter we provide a brief history of computer networking and how it has led to the current paradigm. We also look at previous work surrounding the networking challenges we focus on in this thesis.

In section 2.2 we look briefly at the rise of networking and early routing protocols. As we focus on the aspect of packet routing in the network, we transition from traditional networks to what has been hailed as the future of computer networking in Programmable Networks in section 2.3. We look at its progression from early stages up to the current SDN architecture. In section 2.4 we explore in detail the SDN Architecture and OpenFlow.

Since the birth of networks, one of the most prevalent attacks has been Denial of Service in which an attacker attempts to deny legitimate users access to some service (e.g access to a webserver). We briefly explore in Section 2.5 the Denial of Service concept and look at various forms it can take as well as some of the mitigations which have been proposed for it. This is by no means a holistic coverage of Denial of Service, but helps add context to the challenges we work on in this thesis.

Finally, in section 2.6 to 2.8 we relate the problem of Denial of Service to Software

Defined Networking. We first look at general vulnerabilities in SDNs in section 2.6 and in 2.7 we look specifically at how Denial of Service attacks can be carried out against this type of network and their potential effects. In section 2.8 we explore some solutions put forth for this problem. We review work done in this space which has shown that this is a significant challenge to the SDN paradigm which must be resolved before it can be considered complete and production ready to industry standards.

2.2 The Rise of Networking and Packet Routing

In 1969, DARPA (Defense Advanced Research Projects Agency) created the first packet switching computer [9]. The first message was sent on October 29th of the same year from a computer in UCLA (University of California, Los Angeles) to another in the Stanford Research institute. The programmers there proved it was possible for two computers to communicate over a large distance and it was the birth of the information sharing age. By 1976 the network had grown to more than 60 computers [10] and as networks grew, it became necessary to determine where messages should be sent in order to arrive at their destination. This introduced the need for routing protocols.

In 1982, the Routing Information Protocol (RIP) was introduced. This quickly became the industry standard [11]. It was designed for use within small to medium IP based, internal networks. It used distance-vector algorithms to calculate the best path through a network for traffic between hosts, focusing on the number of hops between the source and destination. However, the RIP protocol had several limitations in that it was restricted to a maximum of 15 hops and so its scalability was low and convergence time, slow. Several newer versions were implemented over the years introducing features such as capability for subnet routing in 1996 and IPv6 capability in the final version in 1997.

In 1986, CISCO introduced a proprietary routing algorithm for its routers called the Interior Gateway Routing Protocol (IGRP) [12]. This protocol also used distance vector algorithms to calculate routes through the network, however a major limitation was its lack of subnet support (as with early RIP versions). It was succeeded in 1993 by the Enhanced IGRP protocol (EIGRP) [13] which was able to

handle classless IP addresses, providing these subnetting capabilities. In EIGRP, a router sent incremental updates to the other routers in the network about its connectivity, with each router using a Routing Table, Neighbour Table and Topology Table to store the information it received. Both the IGRP and EIGRP were much better suited for larger networks than RIP, however CISCO maintained tight control over it.

Open Shortest Path First (OSPF) [14] was originally proposed in 1989 as a replacement for RIP. Unlike IGRP, this was not a proprietary protocol confined to use only in certain devices. Also, instead of using common Distance Vector algorithms, it made use of Dijkstra's Shortest Path First algorithm. It not only considered the number of hops, but the path calculation for packets between a particular source and destination took into consideration the bandwidth and current load on a link with administrators allowed to specify the weight of a link. It was also significantly better for larger more complex networks than RIP. While this improved on several RIP issues, it came at the cost of simplicity. RIP was noted for its ease of deployment, while OSPF required of its administrators some more technical understanding. The latest version of OSPF was unveiled in 2008 and included support for IPv6.

Finally, in 1990, Intermediate System to Intermediate System (IS-IS) protocol was released [15]. This protocol was similar to OSPF in that it used Dijkstra's Algorithm instead of Distance Vector. They were developed by different organisations and IS-IS was originally designed to perform service on OSI's Layer 2. Due to the widespread adoption of the IP protocol, later versions of IS-IS provided support for the IP protocol like OSPF. IS-IS is widely used by ISPs (Internet Service Providers) with heavy backbones. Routing information is spread around the network by way of flooding and each router keeps a database of the network topology.

All of these protocols were known as Interior Gateway protocols and they performed routing within a closed network. Routing between Autonomous Systems (AS) was- and still is- performed by Exterior Gateway protocols such as EGP (Exterior Gateway Protocol) [16] and the more modern and widely accepted BGP (Border Gateway Protocol) [17]. These protocols connected the smaller networks that the other protocols govern, and are widely used on the Internet to exchange information among ASes and ISPs today. In all of these protocols (both Exterior and Interior Gateway protocols), a key characteristic was that routers made their own

decisions about where to send traffic. That is to say, the control plane that made decisions about how and where packets were forwarded, and the data plane that performed the actual forwarding were both intrinsically woven together within the same device; each individual device had its own control and data plane functions. The protocols were implemented to enable the routers to exchange information to ensure that each held an accurate view of the network and could use this information to conclude individually where packets it received should be sent.

2.3 Programmable Networks

Among the first to propose the idea of a separated control and data plane was the Open Signalling Group in 1995 [18] in the hopes of making a network "as programmable as the PC" which would allow new network services to be easily deployed. They held a series of workshops over several years providing a forum for those interested in this new paradigm to collaborate and discuss ideas for implementation. They noted at the time that such a concept would be difficult to implement given that those who provide the networking service cannot easily reprogram the routers or switches. Nevertheless, they set out to realize the idea of easy network programmability.

Out of these workshops came one of the earliest forefathers of today's SDN implementations: the General Switch Management Protocol (GSMP). Among other things, the protocol allowed network managers to query a switch for port, connection or QoS (Quality of Service) statistics, delete connections on specific ports and send configuration messages to learn the capabilities of the switch. Like todays SDN architecture, it used one or more controllers (external to the switch) to establish and maintain the state of a switch under its control. Starting in 1998, the final version of the protocol was completed in 2002 [19].

Also in the late 1990s, the concept of Active Networking was introduced. Active Networking, as put forth by Tennenhouse and Wetherall [20] proposed the idea of switches with out-of-band management channels through which a network admin could program the network. In this concept, "capsules" which were network packets containing code, could be interpreted and executed by routers/switches. However, concerns around the practicality, security, safety and performance of this concept

were raised and Active Networks never gained the traction in industry it hoped for [21].

Transitioning into the early 2000s, the 4D Project put forth a number of concepts within the separated architecture. As part of the project, they introduced several new and key principles in networking which adhered to the new architecture and boosted the control planes capabilities. The principles gave the control plane a global view of the network and full control in decision making for packet forwarding [22]. Additionally, the concept of a router whose only control plane functions consist of information dissemination and executing instructions for packet forwarding configurations was put forth [23]. This is very similar to the OpenFlow Agent found on most OpenFlow capable SDN switches today. Members of the team also advocated a logically centralized control plane for switches within an internal network which performed the route selection function for packets within the network [24].

In the late 2000s, the NETCONF (Network Configuration) [25] protocol was introduced by the IETF which allowed for easier configuration modification within network devices through an API. The SNMP (Simple Network Management Protocol) [26] protocol proposed in the late 80s was also used in conjunction with the NETCONF protocol for configuration. However, these protocols by design lacked the separation of data and control plane, as with the previous works mentioned. This combination is worth mentioning however, as they continued to push for easier configuration of network devices which was a key point of programmable networks.

The final step before the full fledged implementation of what we now know as Software Defined Networking came in the form of Ethane [27]. The epitome of programmable networking capability at its time, it gave network managers fine grained control of the flows on a network switch by use of a centralized controller, completely detached from the switches. The separated control plane and fine-grained policy management meant that managers were now able to easily decide the path a flow should take between two hosts on a network yet efficiently manage several hundred flows at the same time. Its implementation made use of a centralized controller, an ETHANE switch and a secure channel between switch and controller, much like what we see in SDN today. While the controller offered limited functionality compared to current offerings, it was the birth of the modern control plane.

Through this brief history, we see that what we know as SDN today is by no means a new paradigm. While the OpenFlow protocol which has become the defacto SDN protocol is fairly recent (as we discuss in the next section), the separation of network control from the forwarding devices is something that has been in development for several decades.

2.4 Software Defined Networking and OpenFlow

While the ideas surrounding programmable networks were novel and exciting, showing many benefits and huge promise to revolutionize the networking field, for the most part they remained consigned to the research sphere. Most commercial networks continued to employ traditional switches whose control and forwarding elements were tightly coupled together. This was partly due to the lack of widespread protocols which facilitated the programmable networks. Additionally, vendors who had spent years perfecting their individual switches' proprietary internal circuits and algorithms were uncomfortable opening up the interfaces to the experimentation and external tampering necessary to bring about programmability.

In 2008, McKeown et al put forth the OpenFlow proposal [28] to allow easy and efficient separation of experimental and production traffic in the same network which enabled researchers to conduct experiments on realistic networks. This protocol was proposed and implemented to provide a standard for communication between switches and controllers and over the past decade has become the de-facto protocol of SDN [29]. The OpenFlow creators noted that a key part of programmability is the ability to control and direct flows. They also noted that most ethernet switches contain a flow-table which stores these flows. Combining these two principles, they created the OpenFlow protocol to allow an external entity (controller) to directly manipulate and configure the flow table within Ethernet switches. They specified the external representation of flows with a set of associated attributes for each and the commands used to query and configure the flow table. With the protocol in place, vendors were now free to keep the inner workings of their products hidden while offering a well known external interface for configuration.

Previously in 2003 the IETF proposed and implemented the ForCES (Forward-

ing and Control Element Separation) protocol for programmable networks [30]. One of the key differences between the ForCES and OpenFlow architectures is that while there exist two logically separate entities in the forwarding and control elements, the ForCES architecture keeps both entities very close together (on the same device or in the same room) while the OpenFlow architecture allows for any amount of distance between the two. The ForCES protocol uses Logical Function Blocks which takes the place of OpenFlow Flow Tables and is programmed by a control element to instruct the forwarding element on how to process packets.

The Open Network Foundation defines SDN as a network architecture in which, "...the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications." [31]. While ForCES was proposed first and continues to be an active protocol for SDN, OpenFlow has become the most widely adopted implementation in the SDN networks we see today. Implementation of the OpenFlow protocol opened up the field for testing of new protocols and services on the network. Network administrators could easily partition experimental and production traffic ensuring that one does not affect the other [32]. This allowed researchers to begin experimenting with new ideas on production networks, attaining credible results. OpenFlow has also seen industrial deployment in Microsoft's public cloud [33] and Google's backbone network connecting its datacenters around the world [34] among others. In light of that, we focus our efforts primarily on OpenFlow in this thesis and refer to the Openflow version when the term "SDN" is used.

2.4.1 The OpenFlow SDN Architecture

The OpenFlow protocol provided a specification for communication between an SDN switch and Controller. Due to its proximity specifications (or lack thereof) between the forwarding device (switch) and the control element (controller), the OpenFlow architecture additionally specifies a secure channel between the controller and the switch. Thus, the OpenFlow SDN architecture actually has three key elements, as illustrated in Figure 2.1:

- Controller
- Controller- Switch communication channel

- Switch

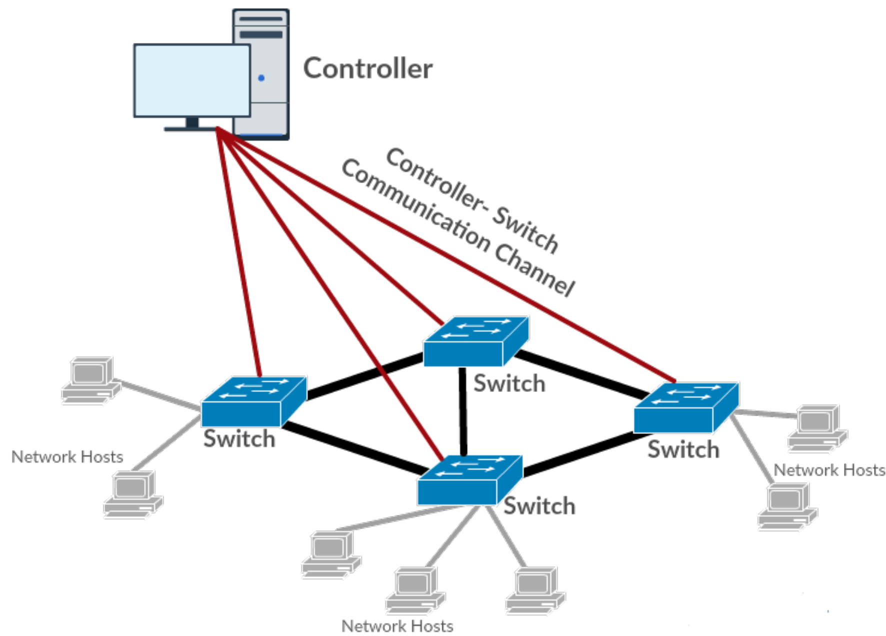


Figure 2.1: SDN Architecture

We provide a general discussion of each of these here and explore security issues with each later on in this chapter.

2.4.1.1 Controller

One of the key characteristics of SDN is the logically centralized controller. The modern controller not only provides a central control point for the flows in the network but also provides a platform for launching other network services. Essentially the controller functions as a network operating system [35] upon which various applications can be run which provide the network services. Some of the more currently widely used controllers include RYU [36], OpenDaylight [37] and Floodlight [38].

The applications typically interact with the controller via a Northbound, REST API (Representational State Transfer Application Program Interface). The controller provides hooks which allow applications to perform queries and issue commands to the switch. These applications provide services such as traffic routing (e.g a mac learning switch application) and QoS guarantees. Many provide security services such as DDoS or scanning detection and protection by monitoring and filtering

the traffic coming into and moving around the network. We explore several such applications later in this chapter.

The controller interacts with the switch via the southbound interface [39]. In OpenFlow networks, it interacts with the OpenFlow Agent residing on the switch. As mentioned before in the discussion on programmable networks, the switch itself contains a very limited control plane which exists for the purpose of interacting with the controller. This is the OpenFlow Agent. Over time, the evolution of the OpenFlow protocol has created more functionality between the controller and the switch, however the basic interactions have remained the same. The controller can issue commands to install flows, remove flows and query the status of the flow table which gives it the flows existing in the switch along with various statistics associated with them (such as the number of packets that have passed through them). Using these commands, the controller exercises complete control over the network and through it, the applications perform their functions.

2.4.1.2 Controller-Switch Communication Channel

The communication channel is described as a secure channel which connects the OpenFlow switch to the controller [40]. It provides a medium over which the controller configures the switch and the switch provides information to the controller. The protocol leaves the implementation to the discretion of the network administrators. Conventionally, wired Ethernet is used, however wireless [41] and even cellular links [42] have been proposed.

2.4.1.3 Switch

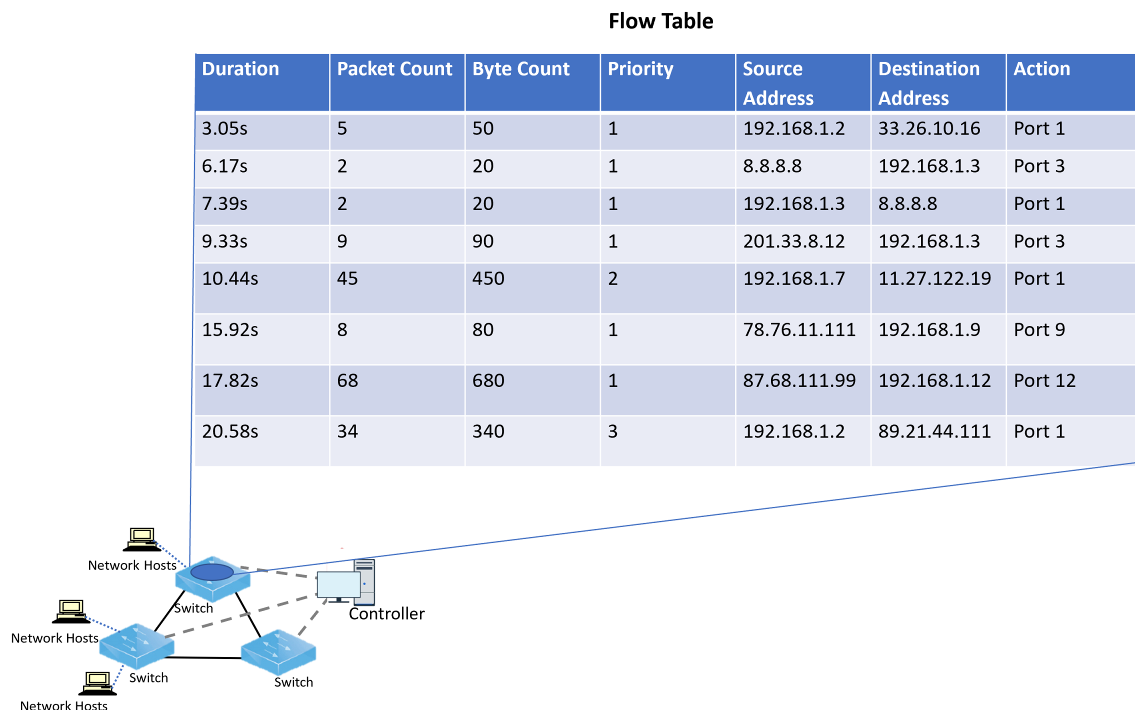
The Switch is responsible for forwarding packets around the network. In a pure SDN architecture, the switch does not make its own decisions about how to route packets but receives such decisions from the controller. In this regard, the switch has often been referred to as a "dumb" device. The SDN switch has several fundamental attributes:

OpenFlow Agent:
The OpenFlow Agent (OFA) is a piece of software on the switch that interacts with

the controller using the OpenFlow protocol [43]. The OFA provides the interface between the switch and the control plane. It is responsible for crafting packet-in messages to request flow rules from the controller and inserting the flow rule issued by the controller into the flow table. On a more general note, it is responsible for responding to controller instructions on the switch. It provides the minimal control plane functions on the switch as specified above.

Flow Table:

The switch Flow Table stores rules providing information on how to route packets which arrive at the switch [40]. Since the switch in itself makes no decisions, the controller is responsible for populating the switch Flow Table by installing or removing rules from it. Each rule contained in the switch flow table consists of several attributes (Figure 2.2 shows a sample). These include the packet header field values with which the rule is concerned, the number of packets and bytes which have used the rule, the length of time it has been in the switch and the actions the switch must take for any packet that matches said rule.



Flow Table

Duration	Packet Count	Byte Count	Priority	Source Address	Destination Address	Action
3.05s	5	50	1	192.168.1.2	33.26.10.16	Port 1
6.17s	2	20	1	8.8.8.8	192.168.1.3	Port 3
7.39s	2	20	1	192.168.1.3	8.8.8.8	Port 1
9.33s	9	90	1	201.33.8.12	192.168.1.3	Port 3
10.44s	45	450	2	192.168.1.7	11.27.122.19	Port 1
15.92s	8	80	1	78.76.11.111	192.168.1.9	Port 9
17.82s	68	680	1	87.68.111.99	192.168.1.12	Port 12
20.58s	34	340	3	192.168.1.2	89.21.44.111	Port 1

Figure 2.2: SDN Flow Table Example

TCAM (Ternary Content Addressable Memory):
TCAM is used to support the flow tables in the switch for fast lookup of flow rules when forwarding packets [44]. TCAM is unfortunately very expensive and this limits the amount of memory the switches hold. TCAM flow tables are often restricted to holding a few thousand flows which is less than ideal in large networks. We explore further why this is a problem later in this chapter.

Data Plane:
The Data Plane is located within the switch and handles the forwarding of packets to the next hop in their path. When a packet arrives at a switch, the switch's data plane consults the flow table for instructions on how to handle the packet. If the packet's header fields match a flow rule in the flow table, the switch performs the action associated with that flow (e.g flood, drop or forward out of a specific port). In the event that the packet's header field values do not match any flow rules in the Flow Table, the packet is passed to the OpenFlow Agent in the switch which prepares a query for the controller on how to handle this packet. Application-Specific Integrated Circuits (ASICs) within the switch provide high performance packet switching within SDN switches. Some well known ASIC providers include Cisco, Huawei, Juniper [39].

2.4.2 OpenFlow- A Protocol for Programmable Networks

The OpenFlow protocol provides a description of the requirements of an SDN Switch and specifications for communication between SDN switches and controllers. Since its conception, OpenFlow has had several versions, each upgrading its functionality and improving on previous versions. OpenFlow 1.0 was the first official version implemented in switches however, several previous (unused) drafts of the protocol were proposed.

2.4.2.1 OpenFlow 1.0

OpenFlow compatible switches were required to have at least two elements: A Flow Table and a secure channel to the controller. The first version of OpenFlow [40] defines a single flow table within the switch which held instructions for all the flows passing through the switch. Any packet arriving at the switch has their header fields compared with the flow rules in this flow table. If the packet matches a rule, the

instructions associated with the rule are applied to it. Any packet not matching a rule in the switch is either dropped or forwarded to the controller for instructions.

As mentioned in Section 2.4.1.3, OpenFlow 1.0 introduced Flow rules or Flow Entries in the switch Flow Table which consist of three elements: Header Fields, Counters, and Actions. The Header Fields are used to match incoming packets to a rule. They include attributes such as Source and Destination IP addresses, Incoming port and Protocol Type. Counters keep track of various flow statistics such as the number of packets and bytes that have used that entry. Actions provide instructions to the switch on how to handle packets matching a given flow. Actions include Forward, Drop, and Modify-Field which allows the switch to modify the header fields of packets. OpenFlow 1.0 lists a set of actions, a subset of which are "Required" which a switch must include to be considered "OpenFlow Compatible".

The protocol additionally enables a set of well defined messages between the switch and the controller. These messages firstly enable the switch to connect to the controller and register itself to the network. The protocol also includes messages to allow the controller to add, modify and remove flows from the flow table, request switch features or the status of the flow table from the switch, and for the switch to notify the controller of an unmatched packet, a flow that has been removed or a port state change. Messages can be Symmetric or Asynchronous.

2.4.2.2 OpenFlow 1.3

Because OpenFlow was hardwired into switches hardware, updating the protocol between versions was not as easy as updating software on a computer. Vendors therefore complained that versions were being released faster than was practical to deploy and for the most part, did not widely implement versions between 1.0 and 1.3. Nevertheless, the subsequent versions continued to build on each other, each adding functionality not previously seen. For brief examples, Group Tables and Group processing were added in OpenFlow 1.1 [45] which allowed for multiple actions to be performed on packets. Openflow 1.2 [46] further built on this by adding IPv6 support and support for extensible matches.

OpenFlow 1.3 [47] was the second version of OpenFlow to be widely adopted by vendors. Unless otherwise specified, this is the version of OpenFlow to which

we refer in this chapter. By OpenFlow 1.3 several significant improvements had been made to OpenFlow 1.0. One major addition was the support for multiple flow tables (while 1.0 had only one) [48]. Multiple flow table brought several benefits to the network. For one, network admins were able to perform packet matching using different header field sets. Additionally, OpenFlow 1.3 made MAC Address Learning and Reverse Path Forwarding Checks significantly easier. These carried many implications for security solutions arising from SDN. It carried support for IPv6, tunneling and gave more fine grained statistics for the flows in the switch.

An upgrade in the protocol also meant the controllers which interacted with the new switches needed to be updated as well. Many developers used this opportunity to create entirely new controllers such as RYU [36]- a single threaded python controller realeased in 2012, and the OpenDaylight project [37] which released its first version in 2013. Despite their late release (with respect to the existence of OpenFlow), these controllers included backward compatibility for previous OpenFlow versions. Others such as Beacon [49] upgraded their functionality to support the new protocols. One of the original controllers that was present at the advent of SDN, NOX [35] recently received a complete revamp.

2.4.2.3 OpenFlow 1.4

One of the most notable additions to the OpenFlow protocol at version 1.4 is the slight increase in switch intelligence [50]. OpenFlow 1.0 touted the switch as a dumb forwarding device, whereas OpenFlow 1.4 imbued it with the ability to decide to actively evict flows without the direct intervention of the controller.

The issue of Flow Table Overloading has been highlighted as an attack on the network in several works and indeed can become an issue even during benign circumstances [51]. In previous versions of OpenFlow, when the table becomes full and the controller attempts to add another flow, the switch would respond with a Table Full error message. The controller would have to directly issue a flow removal command (which may also involve a Flow Table query to determine what flows are in the switch and which to remove) before attempting to insert the flow again. Alternatively, the controller would have to wait until a flow expired before it can add another flow. Both of these options incur significant delays and dropped packets.

OpenFlow 1.4 attempts to mitigate this issue by giving the switch the ability to remove the oldest flow in its flow table before adding the new flow. This reduces the number of messages that need to be sent between the controller and the switch and the minimises computational resources on the controller in the exchange. We focus on this particular aspect of the OpenFlow protocol in this thesis as we highlight the effects of DoS attacks and propose potential solutions.

We note that Pica8 [52], a prominent SDN switch vendor has already implemented support for OpenFlow 1.4 in its switches but it is yet to receive widespread adoption. However, we can assume that future versions of the protocol will include the eviction aspect for the foreseeable future and as vendors implement upgraded versions, their switches will also have this capability [1].

2.4.3 Beyond Openflow: Stateful SDN Data Planes

One of the intrinsic characteristics of OpenFlow SDN is that the switch is fully reliant and dependent on the controller. While this is in many ways useful, it has in itself several setbacks that we identify later including high communications between the switch and the controller (as the switch must go to the controller each time it needs instruction).

In the interest of reducing such communication overhead between the switch and controller, the idea of Stateful SDN switches was introduced. Such switches store more information than basic SDN switches and can independently reconfigure forwarding rules based on the state of the traffic. The switches often store various historical information (state) on a flow based on the packets and make forwarding decisions for the flow based on its current state. These stateful switches still retain dependency on the controller by allowing the controller to define the forwarding decision options which the switches choose from when determining the change in forwarding action based on the state.

While we focus on Openflow SDN in this thesis, we briefly describe some of the work done in Stateful SDN switches here for completion, since our ideas of an in-

[1]OpenFlow 1.5 [53] is the latest publicly available version at the time of writing and also includes the flow eviction capabilities.

telligent Openflow SDN switch share similar attributes with Stateful SDN switches. However, while Stateful switches divert from the Openflow protocol, this thesis implements additional functionality in conjunction with the protocol, creating an avenue for easier adoption as Openflow continues to be the most widely used and de facto protocol for SDN.

Several works have been proposed in this area, some of which we briefly describe here. Openstate [54], SDPA [55] and Fast [56] are platforms which utilize tables in the switches to store various state and transitional information for the flows. The tables store information providing instructions on criteria for transitioning between one state and another, each flow's current state and the actions associated with each state. Packets arriving at the switches have their headers cross-referenced with the information in these tables to determine if a state change is required for the flow and the switches determine which forwarding actions are appropriate for each flow based on the state each packet brings it to- all without needing to consult the controller.

Several switch programming languages have also been proposed, two of which we discuss here. P4 [57], a language for switch configuration, allows for stateful packet processing by switches regardless of the physical attributes of the forwarding device it is deployed on (target-independent) and enables the switch to parse new header fields. P4 allows programmers to configure the fields parsed by the switch, the actions taken by the switch, and how the switch should process the packets. By contrast, the Domino programming language [58] provides a programmable instruction-set specifically to accompany the Banzai hardware switch, a line-rate programmable switch put forth by the same researchers. Banzai uses "atoms, a vector of processing units, which stores state variables for each packet and enables stateful processing of packets each clock-cycle. The programmers therefore only need concern themselves about operations on a single packet and need not worry about state access conflicts since the operations are atomic. While P4 is compatible with multiple hardware switches, Domino is built to accompany the Banzai switch which supports a limited number of packet headers and operations. Nevertheless, both of these indicate the promise of stateful programming for SDN switches. In a more general sense, the field of Stateful SDN dataplanes shows that the research community recognizes the need to explore additional functionality for the SDN switches, moving away from the "dumb forwarding device" paradigm.

2.5 Denial of Service Attacks and Network Security

With the rise of computer networks, the prevalence of malicious behaviour within networks has also grown. Such malicious behaviour can now come in several forms including Worms, web based attacks and Man In the Middle attacks. One of the most popular types of malicious behaviour today is Denial of Service attacks. Denial of Service (DoS) attacks are explicit attempts to prevent legitimate users from accessing a system or service by reducing availability [59]. Given the importance of computer networks as highlighted in Chapter 1, DoS attacks can present a serious threat to the various organisations that rely on computer networks.

One of the earliest examples of DoS attacks was recorded in February 2000 when several high profile web-sites including Yahoo and Amazon were shut down by a series of Distributed DoS attacks. This helped bring the problem of DoS to the forefront of cyber-security discussions [60]. DoS attacks began to receive a significant amount of attention from the security community, and in 2001 a study into the prevalence of DoS attacks was conducted [61]. Using a backscatter analysis technique that looks at victim responses to attack packets, the researchers were able to roughly determine when an attack was in progress. Within the three-week duration of the study, they observed almost 13000 attacks with more than 5000 distinct targets in over 2000 organizations. This study began to shine some light on how widespread the problem was and how badly it warranted attention and solutions.

The traditional DoS attack is often thought of as a high volume stream of traffic being sent toward a server [60] though in reality it can take many different forms, each devastating in their own right [5]. A Denial of Service attack can be as simple as disconnecting the power source of an important host on the network to more complex attacks such as the infamous SYN attack which consumes the number of open TCP connections a server can maintain, stopping legitimate users from establishing a connection [62]. The attack victims may be a network host, a specific application, or the network itself. In this thesis we explore the network infrastructure-targeted attack. Within the traditional DoS space, similar examples to the ones we explore in this thesis include the Coremelt [63] and Crossfire [64] attacks in that they target

specific links or switches to deny connectivity to a particular service on the other side.

Attackers who mount DoS attacks against a particular host, network or service may do so for a variety of reasons. They may be motivated by revenge, personal economic gain, ideological beliefs or may simply desire an intellectual challenge [65]. They often take several steps to conceal their attacks, yet amplify them so that they are as effective as possible. Attackers may use multiple hosts (Distributed DoS Attack) or even intermediate networks [66] to amplify the effects of their attack. To hide their identity and potentially avoid repercussions, attackers often spoof their source addresses to ensure the real source of the attack is not revealed. In a further effort to avoid detection, attackers may vary the rates of attack over time to attempt to subvert some detection mechanisms.

Despite the prevalence and attention of DoS attacks, the most common theme among them is that they are difficult to solve. This is in part due to the range of variation the attack can take but more so, it has proven incredibly difficult to distinguish and filter DoS traffic without affecting the traffic of legitimate users. Using various stealth methods, DoS attackers often find innovative ways to hide their traffic so that it closely resembles legitimate users' traffic [67]. Attacks continue to increase in frequency, impact and sophistication while effective defenses lag sadly behind.

Over the past two decades, a myriad of solutions have been proposed for DoS attacks with varying degrees of success, however none have been established as complete solutions which remove the issue entirely. Some defenses proposed attempt to guarantee fair resource distribution among well-behaved users, so that no one malicious user is able to prevent legitimate users from getting service [68][69]. Others focus on identifying the attack as early as possible using anomaly detection systems on the packet and flow statistics, for example [70][71][72]. Others still either scale up the resources or reconfigure the network in an attempt to handle the DoS attack (not prevent or filter it) and still provide service to legitimate users [73][74] as opposed to those who attempt to filter the attack traffic [75].

Many solutions deploy their DoS prevention systems at the host at which they

expect the attack [76], while others augment the abilities of the router to recognize and mitigate the attack [77][70]. Others still attempt to mitigate the attack as close to the sources as possible [78][79], but this is far more difficult as it requires co-operation from entities with little concern for the victim's well-being.

With its centralized control (as described in Section 2.4.1), SDN brings new opportunities for DoS defense. Several works make use of SDNs ability to monitor and finely control the networks flows to detect, filter and/or redirect attack flows. For example, some SDN solutions monitor the traffic flows [80][81] and flag any that look suspicious [82][83]. Others can go a step further and utiilise SDN to reroute flows to mitigate the attacks [84][85].

While in many ways, SDN is ideally suited for detection and mitigation of DoS attacks seen in traditional networks, the new paradigm brings in itself new vulnerabilities for DoS attacks [86]. Most of the solutions proposed for DoS attacks in traditional networks are not applicable to SDN focused DoS attacks which may make a point of targeting the controller, the controller switch channel (neither of which existed in traditional networks) or the switch flow table which is necessarily limited in SDN. In many cases, these SDN focused DoS attacks lack the characteristics which traditional DoS attacks monitor for (e.g unusually high numbers of packets to a specific server). Due to this, traditional DoS solutions may miss SDN focused DoS attacks in progress and provide a false sense of security to network users and administrators.

Because of this we are forced to explore new solutions to tackle the problems SDNs are faced with in order to bring the SDN paradigm to completion. In the following section we discuss some of the vulnerabilities that have been highlighted in SDN, giving a brief general overview of the issues and focus specifically on the DoS vulnerability and some of the solutions put forth to mitigate this in the subsequent sections.

2.6 Software Defined Networking Vulnerabilities

With the introduction of OpenFlow, SDN overcame a great hurdle to break into the industrial field. The vendor acceptance of the protocol and its ease of use meant

that it could make a reasonable argument for industrial deployment. Add to that the extensive benefits the new paradigm had shown and it seemed almost a foregone conclusion that this new type of network was on the brink of usurping its predecessor. However, as researchers began to look more closely at the SDN it became evident that this new design created in itself a host of new vulnerabilities.

The network infrastructure, due to the separation, if not properly secured opens itself to a range of attacks from eavesdropping to DoS attacks. The lack of mandatory authentication between the switch and controller provides the potential for Man in the Middle and eavesdropping attacks [87]. Lack of separation of privileges and authorization of controller applications can give rise to major issues [39]. Such vulnerabilities would allow attackers to insert fraudulent switches or controllers into the network and order to monitor or control traffic at their will.

Malicious entities can cause wormholes and blackholes in the network by attacking the switch, potentially compromise the administrator station housing the controller [88] or use various side channel attacks to learn the network state [89]. Attackers can also use fake traffic flows to DoS the control plane [88] or fill the switch with arbitrary flow rules such that new, legitimate flows cannot be installed [89]. Such attacks, as discussed later, have the ability to significantly degrade the service provided by the network. We focus our efforts in this thesis specifically upon these last issues but it is helpful, for context to briefly discuss the overall failings of the SDN paradigm to show it is not yet in a position to replace traditional networks.

2.7 Denial of Service Attacks in Software Defined Networks

As previously highlighted, DoS attacks have long been a major vulnerability of computer networking. They are high impact attacks which require little knowledge and expertise to carry out but effectively manage to disrupt service to legitimate users. With the transition to a new paradigm, SDN networks carried over this issue and opened up in themselves new opportunities for attackers to restrict legitimate users' use of the network.

Several early works within the SDN and Network Security community highlighted the possibility of DoS attacks in SDN and explored it from various angles. Kloti et al [89] were among the first to bring this issue to the forefront. They analysed the OpenFlow protocol using the STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege) methodology and concluded that an attacker could launch a Denial of Service attack that would overload both the flow tables and the controller. In a reactive SDN network, by creating a large number of packets with slightly different header fields, the attacker would cause the switch to forward a high number of packet to the controller. This could potentially overload the controller and fill the flow table with rules which prevent legitimate users' rules from being installed. Similarly, [88],[87] and [39] identified the threat of malicious flows which could be used to exhaust controller and switch resources. These identified vulnerabilities which, if exploited, would bring a halt to the operation of the network due to the controller being too busy to create new rules and the switch being unable to install new rules meaning no new traffic could be routed through the network.

2.7.1 SDN DoS Attack Definition:

For the purposes of this thesis, we define the SDN DoS attack as an attacker sending a large number of unique packets to the switch, triggering flow requests to the controller.

The are four major elements of the OpenFlow SDN architecture which can be targeted for a DoS attack [43]

1. Controller
2. Controller-Switch Communication Channel
3. Switch OpenFlow Agent (in charge of issuing flow requests and installing flow rules)
4. Switch Flow table

The target element of the architecture may determine how the attacker performs the attack. An attacker whose target is the flow table, aims to fill the flow table to ensure that no new flows can be added to the switch allowing new traffic to be

routed. By contrast, an attacker whose target is the controller aims to overload the controller resources so that it cannot add legitimate flows to the switch. While the concept of the attack remains within the definition specified above, for each target an attacker may carry it out in a different way.

2.7.2 Reconnaissance

To facilitate their DoS attack, an attacker may perform reconnaissance before the actual attack to glean information about the network configuration in order to make their attack as effective as possible. SDNs protocol of sending the header of the first packet (to the entire packet if the switch buffers are full [90]) of a flow to the controller exposes a vulnerability for side channel attacks. The diversion to the controller causes a delay in the first packet and attackers monitoring for this delay can infer several factors about the network.

In its simplest form, the initial delay can help an attacker determine whether the network they are about to attack is in fact an SDN network [91]. Further investigation using active probing can give attackers an idea of the timeout (time until the flow rule expires and is no longer in the switch) attributes and flow rule fields of the rules in the network [92] and in some cases which rules are and are not in the switch [93]. Once they have determined this, they are able to infer how their packets must be crafted to generate flow requests to the controller and new rules put into the switch. Attackers can also determine the size of the flow table (which is a key factor in attacks that seek to overflow the TCAM), whether or not the controller is actively removing flows in the event of a filled table and if so, the algorithm being used to select a flow for removal (e.g First In First Out, Least Recently Used) [94]. However, it should be noted that the accuracy of these side channel attacks is heavily dependent on the background traffic surrounding the probes at the time of the attack.

2.7.3 Controller

Often called Controller Saturation or Control Plane Saturation, this attack occurs when an attacker repeatedly sends packets to the switch which generate flow requests to the controller. The high rate of the flow requests to the controller exhausts the controller's CPU and memory resources [95]. This makes the controller unable to

service flow requests for legitimate traffic [43] causing the switch to drop packets. The network user's ability to influence the work a controller does makes the most essential part of the network an open vector for attack by design. Attacks on the controller can render the network inoperable and the effects of this attack can be exacerbated in large networks with lengthy flow paths. Paths which must encounter multiple switches can generate flow requests at each hop so that the attack's effect is multiplied [96]. While all controllers have an upper limit of requests they can handle before being overloaded, different controllers give varying performances under attack. For example, when compared against each other, under attack ONOS was found to give the worst performance with RYU performing better than ONOS while Floodlight recorded the best performance [97].

2.7.4 Controller Switch Communication Channel

One of the effects of the DoS attack on SDN networks is congestion of the controller-switch communication channel. When a switch receives a large number of packets for which it has no flow rules in a short time, its buffer gets filled. When this occurs, the switch forwards whole packets to the controller instead of just the packet headers [43]. This leads to congestion in the switch controller channel and can increase the delays in installing new flow rules. As the delays to install legitimate flow rules increase, packet loss in their traffic also begins to increase significantly [98]. An attacker can trigger this congestion effect by sending packets triggering flow requests at a high enough rate to overflow the buffers and consume the bandwidth [99].

2.7.5 Switch OpenFlow Agent

The OpenFlow Agent (OFA) is the component of the switch in charge of interacting with the controller. On hardware switches however, they often run on relatively low-end CPUs (compared to modern server CPUs on which their virtual counterparts are often run), limiting the number of flow requests they can generate and flow installations they can perform per second [43]. Some hardware switches have been shown to have limits of less than 1000 requests per second [100]. This therefore makes the OFA a bottleneck in the architecture and an attacker who overloads this can cause the switch to drop packets for flows it is unable to generate requests for, thus affecting the legitimate traffic.

Furthermore, in switches where a central CPU does the work of both the OFA and the routing of the packets (e.g software switches), the overloading of the OFA may affect the routing as the switch CPU exhausts its resources on control plane activity [97]. Under attack, the CPU can spend its resources installing and removing rules and sending flow requests such that its packet routing is no longer done at optimum rates and packets of legitimate flows are dropped. Again, it has been found that the controller in question for the network plays a major role in the effects of the attack as communication with some controllers demand more CPU resources than others (e.g ONOS was found to incur less CPU load on the switch than Floodlight under high packet_in rates [97]).

2.7.6 Switch Flow table

The aim of attacking the switch flow table is often to restrict new flows from being added to the network. Switches' limited amount of TCAM memory often means that the flow tables are restricted to a few thousand rules per switch [101]. In the current widely adopted version of OpenFlow (v1.3), flow rules inserted into the switch are only removed if they expire, or if the controller explicitly removes them. Thus, barring the latter, an attacker can fill the switch with rules to ensure that no new rules can be placed in the switch to allow new traffic to be routed [43].

Any new commands for flow rule insertion from the controller will be met with "TABLE_FULL" errors and the command rejected. Additionally, filled flow tables can result in reductions to the bandwidth and a significant amount of dropped packets [102] if flows are instead routed through the controller. The attack can come from both a malicious host creating a large number of flows to overload the switch or a malicious application sitting on the controller creating arbitrary rules in the switch [103].

As previously discussed, an attacker can perform reconnaissance attacks in preparation which give information about the network configurations. If they are able to determine the configuration for rule expiry, they are able to determine the bare minimum rate they need to send packets in order to keep their rules in the flow table [92]. This makes the attack stealthy and unnoticeable enough to fly under most anomaly detection systems for SDN, which rely on detecting abnormal rates

of flow requests. Use of a botnet further disguises the attack by distributing the source addresses so that it appears to be regular low frequency traffic [90].

Since aside from direct controller intervention, expiration is the only mechanism of removing a rule from the switch (OF1.3), the expiration configuration can play a large role in the effect of the attack. The time a rule is allowed to remain before expiration is called its timeout value. Long timeout values prevent new rules from being installed at the expense of stale flows- this works in the attackers favour as they are able to hold their positions in the flow table for longer with minimum interaction. Short timeout values however forces the switch to repeatedly make flow rule requests for previously known flows- this works against the controller as it increases the number of flow requests the controller receives [98].

2.8 Denial of Service Controller Based Solutions

Many solutions turn to the controller for help in the event of a DoS attack. At first glance this appears to be intuitively the best point of defence since the controller makes decisions for the network, has a global view, and seems to be in the best position to mitigate the attack. Thus, several solutions to protecting the network against DoS from the controller vantage point have been proposed.
We also include some solutions to generic (non-SDN focused)DoS attacks that make use of the SDN paradigm here because even though they are not intended to protect against SDN specific DoS in one of the four identified target areas above, their methods of protection also inherently offer protection for those target areas.

2.8.1 Monitoring

Several solutions propose entropy based detection of DoS attacks. In these, the controller looks for either a sudden rise or fall in the entropy of one of the packet header attributes being sent to it as flow requests. Mousavi and St-Hilaire [104] propose using the controller to monitor the incoming packets and the entropy of the destination addresses within a given window of requests. If the entropy drops below a given threshold, the controller concludes that an attack is in progress. This system assumes that the attack is targeting a host such as a web server within the network

and not one of the aforementioned targets. Despite this, a side effect of the attack they focus on is that the controller receives a large number of flow requests and the flow table is filled, due to many sources (bots) sending flows into the network. Thus they are also attempting to mitigate SDN focused DoS. They achieve a very high detection rate (between 96%-100% in their experiments) which lends significant credit to their solution. Similarly, [105] has a middle box which caches packets and sends packet headers to the controller in the event of a suspected attack. The controller performs entropy based analysis on the packet headers and places rules into the switches to mitigate the attack. While the researchers in this work do not provide extensive results, their singular result indicates promise in that it was able to reduce the processing time of the table-miss packets under attack. Both solutions that look at entropy of packet headers may be limited to an attacker external to the network however (a factor that we address in the following chapters), and may find difficulty in detecting an attacker within the network who can alter their packet headers without affecting the entropy of the overall traffic in the network.

2.8.2 Machine Learning

Several solutions at the controller use Machine Learning algorithms to detect DoS attacks. As part of their system, FloodDefender [99] uses an SVM (Support Vector Machine) classifier to identify malicious flows in the switch. It uses the presence (or lack thereof) of a reverse flow (including the packet and byte counts for both the flow and its corresponding reverse flow) to classify it as malicious or benign. In evaluation, FloodDefender showed improvements in network bandwidth through its detection and mitigation mechanisms which are further discussed in Section 2.8.3. [106] also proposes the use of SVM classifiers for controllers to identify DDoS attacks in the network. It uses Source IP, Destination IP, protocols, and Source and Destination Port as attributes for classification. It proposes its system as a solution to the problem of controller and table saturation but does not test the system with attack traffic that contains this specific attack. Additionally, some of these attributes can be spoofed by an intelligent adversary to subvert the defense system. While the researchers presented limited results, they were able to show that their SVM system produced a high detection rate when compared to other algorithms. [107] goes a step further using SVM classification but adding to it Self Organising

Maps to classify those flows for which the SVM classification produces vague or ambiguous results. The system regularly polls the switch for the flows in the flow table and extracts from them features over which it performs a quick SVM classification. Any flows whose classification result is "vague" is subjected to further SOM (Self-Organizing Map) classification which is slower but more accurate. However, polling the switch may not be the most effective method of gathering statistics under attack as the controller-switch communication channel can be congested, hindering controller-switch communication. The researchers show that their methodology of combining SVM and SOM produces a higher detection rate than either of those methods on their own- SVM and SOM produce a 92.33% and 93.49% detection rates respectively, while the combination of both produces 96.03% detection. The combination also results in less work being done by the CPU than the individual methods by as much as 15%-20%. [108] uses neural networks as its classifier, extracting features such as packet count, byte count, packet rate, byte rate and the survival time of the flow. While this system is one of those originally meant to mitigate generic DDoS attacks, it may also be applicable for SDN specific attacks. Using its "packet_in trigger", its experimental results show the ability to quickly detect high volumes of packet_in messages which is useful for SDN focused attacks.

2.8.3 Thresholds

Systems may also use thresholding to detect and prevent an attack. The initial detection stage of FloodDefender [99] involves a filtering module which uses thresholds on packet-in frequency to detect and filter out attack traffic. Flow requests that repeatedly appear above a user defined frequency are defined as malicious and dropped before the second stage of SVM classification mentioned above. FloodDefender shows significant success in its software environment where it is able to restrain the effects of the attack to a 35% drop in bandwidth while the native SDN environment's bandwidth drops to 0% under attack. It is also able to protect the SDN switch flow table better than other solutions by minimizing the attacks' effect on the available flow table space. Flowkeeper [109] similarly has a middlebox to which unmatched packets are diverted which attempts to filter out malicious packets by monitoring the frequency of the flow use before sending the request to the controller. It sets thresholds for low and high frequencies. Low frequency flows are

filtered out at the middlebox as malicious, high frequency rules are allowed to stay in the network as legitimate rules. Those that fall in the middle are sent to an app on the controller for further analysis. Under attack, Flowkeeper was able to preserve approximately 80% bandwidth, however the attributes it assesses to determine the legitimacy of a flow are easily spoofable by attackers attempting to subvert their solution. [110] indicate that their analysis of DoS attack data show that IP addresses associated with DDoS attacks make fewer connections than the average user and transmit fewer packets than the average user. With this in mind, they set a threshold- k for the number of connections a user must make and a threshold- n for the number of packets a user must transmit in its connections to be considered benign. Any IP addresses noted to be falling below these thresholds within a given time window is considered malicious and blocked.While the authors present limited experimental results which appear to show improvements in the bandwidth and flow table utilization under attack, this methodology hinges on the accuracy of the source IP addresses on the packets, which are again spoofable by an intelligent adversary.

2.8.4 Queues

Zhang et al [43], Wang et al [111] and Wei and Fung [112] use queuing systems to avoid monopolization of the controller by attackers and to defend against controller saturation attacks. In [43], all incoming flow requests are placed into a queue based on the switch or port that they arrive on and the queues are serviced using a weighted round robin algorithm. The queues can be dynamically resized and are split and combined at various sizes. This solution works well for the attack they aim to mitigate, with their experiments showing that even under attack, the Round Trip Time of their packets was not affected when the queue system was implemented while the standard controller provided no service to the packets after 1000 requests per second. However, [112] takes a more sophisticated approach by assigning trust values to each IP address and under attack times, it gives higher priority to senders with higher trust values. Experimentation showed that this solution, FlowRanger, was able to significantly improve the ratio of attack requests serviced vs legitimate requests services when under attacks, servicing far fewer attack requests than the standard First Come First Serve methodology (in one case as much as 85% fewer). The authors use IP addresses to identify users however, an attribute easily spoofed

by attackers to make their malicious packets appear trustworthy. Both methods are intended to stop an attacker from preventing the controller from servicing legitimate requests. Finally, [111] also uses weighted round robin algorithms to service packet-ins in queues in their Floodguard system, however it separates its queues by protocol as opposed to trust values or incoming port or switch. It uses a middle box to cache and separate packet-ins and rate-limit the flow requests to the controllers to try to ensure that legitimate requests are still serviced. In its evaluation, Floodguard showed the ability to preserve approximately 80% of the bandwidth in the hardware environment and more than 95% in the software environment. The authors do not provide the specifics of the attack and legitimate traffic used in the evaluation, however since the solution divides traffic by protocol, its performance if the attacker utilizes the same protocol as the legitimate traffic remains to be seen.

2.8.5 Other Mechanisms

Wolf and Jingrui [113] propose a Proof Of Work (POW) form of protection against DoS attacks on the controller. It highlights the fact that the balance of work is weighted in favour of the attacker in a reactive SDN network. The attacker forces the controller to receive and process the packet-in, map a route and install the necessary flow rules all for the price of a single packet. To counter this, the researchers propose that anyone connecting to the network be forced to include a proof of work in the first packet of their flow, thus shifting the balance of work back in favour of the controller. In evaluation, the authors demonstrate that the time necessary for the proof of work is not prohibitively expensive for the SDN interactions. They show their solution takes no more than 60 microseconds additional time for a valid POW and reduces the time taken for processing in the event of no or invalid POWs by dropping the packets. While novel, this solution may face major difficulties in implementation as it requires major updates to network connection protocols for end users which may not be easily or willingly adopted.

Yuan et al [114] present a unique solution to TCAM overflow attacks by scaling up the target being attacked. As a switch flow table is nearing capacity, the controller makes use of neighbouring switches which have free flow space and installs the rules that would have been installed on the victim switch on its neighbours instead.

Thus, it makes the attack much harder to succeed and it mitigates side channel attacks which may try to tell the attacker the size of the flow table they hope to attack. They effectively show that the greater the number of switches in the SDN network, the longer the network is able to defend against attacks while the standard SDN implementation is unable to defend itself regardless of the number of switches used. Their results indicate that at 2000 flow requests per second, their network of 100 switches is able to defend itself for more than 5 minutes, however the trends of their results show that with enough switches, the network can overcome the Table Overflow attack.

Bahaa-Eldin et al [115] use a second controller and switch as a "sandbox" network in which it monitors new flows to ensure their validity. Any new flow is placed in the secondary "sandbox" switch for a given amount of time to ensure more traffic is routed through it before placing it into the main switch. If it does not have any further packets, it is deleted. No substantial results are provided by the authors and this method requires more evaluation before it is considered a valid solution particularly to ensure that the sandbox network does not become overloaded in large networks, reverting the network back to its original, unprotected state.

Xu et al [116] identify high value and high risk switches within the network and aim to protect them from co-ordinated attacks. They look for several features in the traffic such as sudden flow rule increase at the target switch and common rules in switches in the network. Having detected the attack, they mitigate it using a token based system in which a user has a limited number of requests they can make for a potential target switch within a given time bucket.Their results show that they limit the flow table consumption under attack to as little as 25% and caused 60% fewer attack packets to be transmitted than the SDN network without their solution indicating this is a potential protection mechanism for SDN switches as the scale of their deployment grows on the Internet.

2.9 Denial of Service Switch Based Solutions

The programmable network paradigm specifies that the switch should be a dumb forwarding device while all the intelligence of the network is moved to the controller. However, placement of intelligence does not need to be mutually exclusive.

Increasing the intelligence of the switch to participate in the defense of the network presents itself as a worthwhile avenue for exploration because while they do not have the global view and control of the Controller, they are the first point of contact for an attack and so are in an excellent position to put a stop to the attack before it affects other elements of the network. Here we look at several attempts to augment the switch's intelligence to aid in the protection of the controller and the network.

2.9.1 TCP Proxies

One of the earliest attempts at improving the switch's capabilities, AVANT-GUARD sought to mitigate control plane saturation attacks and control channel saturation attacks by extending the switch to act as a proxy for attackers attempting to set up TCP connections with victims within the network [117]. The idea was to defend against controller saturation and TCAM overflow attacks by forcing the attacker to prove they are genuinely attempting to establish a connection rather than trying to set up fake flows. Any TCP SYN packet arriving at an AVANT-GUARD switch is made to complete the full TCP handshake before a flow requests is sent to the controller. AVANT-GUARD evaluation showed the ability to deliver 100% of benign packets while under TCP SYN flood attacks consisting of up to 800 Packets per second while the standard SDN network delivered none over 50 packets per second attack rate. LineSwitch [118] further improved on the AVANT-GUARD system which was shown to run the risk of an attacker exhausting the proxies resources. LineSwitch instead only proxies one connection for any source, allowing any other connection attempts to go through unhindered but monitoring the frequency of SYN packets being sent from that source. For those sources attempting additional connections, LineSwitch proxies every 1/x subsequent attempts.In its preliminary evaluation, LineSwitch increased the time for a switch buffer to be saturated by an order of magnitude when compared to AVANT-GUARD, providing protection for significantly longer. Both solutions only address TCP connections however, which allows flooding attacks using UDP packets to subvert them.

2.9.2 Duplicate Flow Requests

One issue in the SDN paradigm is that the protocol which dictates "any packet for which a switch has no flow rule must be sent to the controller" fails to account for packets of flows for whom a request has already been sent. Thus, if packets of a flow arrive at the switch after it has sent the first packet to the controller but before it receives a reply, these packets are also unnecessarily sent to the controller, resulting in the controller receiving duplicate packets, multiplying the work it has to do and easily giving way to controller saturation. [119] and [120] both attempt to resolve this issue by increasing the switch's intelligence. The switch in [119] keeps track of which flows are new and which are simply waiting for a controller response and only sends the first packet of the flow, buffering the rest until the controller responds. The authors do not present results to evidence the improvement to the network, however the concept addresses a known redundancy that can affect the controller performance. Switches in [120] similarly check for packets already sent to the controller but goes a step further by allowing the network admin to specify what should happen to the duplicate packet-ins (e.g rate limit or discard). It additionally asks the controller at network boot time, what packet header fields it is interested in and only forwards flow requests for packets that contain those fields. In their evaluation, their mechanism was able to lower significantly the CPU usage when implemented and configured properly compared to when not, in some cases more than 90% reduction. While they did not present Controller CPU utilization results, the significant improvements to the switch CPU utilization give great merit and warrant further investigation into how this solution improves the SDN network.

2.9.3 Entropy Based Detection

Several other pieces of work extend the switch intelligence to monitor for DoS attacks and co-ordinate with the controller for mitigation. The switch in [121] monitors the flow request rates of the network and if it exceeds a threshold (based on the historical average within a given window), sends all table misses (unmatched packets) to a middlebox for an entropy based analysis to determine if an attack is in progress. In their evaluation, the solution is able to quickly (within 1 second) detect large entropy changes in the Source Address, Destination Address and Protocols of the unmatched packets headers coming into the switch. Having done this, their results

also demonstrate that they are able to put a stop to the DDoS attack within a few seconds by creating rules in the switch which address the packets causing the sharp changes in entropy. In evaluation, the onset of the attack is sudden which produces reasonably good results. However, due to the history-based nature of the solution an attacker that gradually increases the intensity of the attack may manage to subvert the system, causing it to think that the "higher than normal" entropy is in fact normal. In this case, the mitigation mechanism may not be activated, causing the solution to fail and the network to remain under attack.

Similarly in [122] the switch monitors the features of incoming traffic and uses an entropy based analysis to determine if an attack is in progress. In this instance the analysis occurs within the switch itself rather than the middlebox. Its evaluation showed that it was able to quickly detect DoS attacks in the network from the switch better than some commonly known sampling tools however it may not be appropriate for protecting the SDN network against DoS attacks specifically targeted at the network. While [121] is aimed at protecting the controller from SDN focused DoS attacks, [122] focuses on protecting end hosts in SDN networks from traditional DoS attacks. In both cases ([121] & [122]), the switches detect the attacks (showing the merit of switch based intelligence) and notify the controller which places rules to mitigate that attack. [123] focuses on protecting the network from external attackers by placing an entropy based DoS detection system directly in the ingress switch of an SDN network. Implemented in Open vSwitch (OVS), the system extends OVS to look at statistics within the OpenFlow switch and determine when an attack is in progress by looking for change of entropy in attributes such as IP address and number of packets. In evaluation, the researchers show the distinct changes in entropy recorded when an attack was in progress to demonstrate the effectiveness of their monitoring system. The researchers recorded entropy value differentials of up to 20% under attack allowing them to detect the attack and potentially alert a mitigation system. This again attempts to protect against host-targeting DoS attacks and may not be suitable in its raw form for attacks on the network elements. Nevertheless it shows the merit in increasing intelligence in the data plane.

Instead of using thresholding and entropy, [124] maintains profiles for the traffic on the switch and the controller. The switch then monitors the bandwidth for surges and determines which pair of attributes are deviating most from their pre-

surge profiles (Suspicious Pair). Any incoming packets whose Suspicious Pair scores exceed a certain threshold are dropped. Experimental results indicate high accuracy in attack detection (>80%) however, a major limitation which is discussed in the evaluation is that if an attacker's packets closely mimic legitimate traffic, the solution begins to label and discard legitimate packets as attack packets. This may prove an easy avenue of subversion for an intelligent adversary if this solution is deployed in production networks, causing the legitimate user's network experience to be adversely affected.

SDN Switches in [125] contain security extensions to the standard OpenFlow action set (DROP, FLOOD, OUTPUT etc). These extensions come in the form of additional security focused actions which allow the switch to detect a DoS attack, Scan attack or intrusion into the network. Proof of concept examples show actions implemented which allow for detection of DoS attacks by monitoring byte rates and scan detection which monitors TCP/UDP connections. In evaluation, the researchers show that their solution does not significantly affect the throughput of the network or the latency of the packets while monitoring for DoS attacks. They show that both the latency and throughput are comparable to the network without UNISAFE actions implemented, indicating that this is a viable solution to DoS detection. The concept could be further extended to allow for protection of the network infrastructure. The Unisafe extension provides a framework allowing security to be integrated directly into the network devices and protocol rather than a feature proposed for the network as an afterthought. The concept of extending the OpenFlow protocol action set to include security focused actions is a strategy that merits further work.

Solutions such as these show a trend in research of looking past the controller as the primary point of defence within the SDN paradigm and the emerging works make a strong argument for the case for switch intelligence.

2.10 Distributed Control Plane and Load Balancing

The nave SDN concept proposed a single, centralized controller controlling a number of switches. A single controller controlling the network, while novel, was highly impractical due to efficiency, network scalability and availability issues [126]. This singular controller represented a single point of failure for the network since all forwarding decisions depended in the controller [127]. With this in mind, several proposals were put forth which maintained the logically centralized nature of the controller, but distributed the control plane for better resilience. While this approach increased the number of controllers, making it harder to exhaust the available resources, it also brought new challenges as controllers would need coordinate actions to avoid issuing conflicting commands to switches, out of sync network views and race conditions within the network.

To avoid the scenario in which the single controller becomes overloaded or fails, crippling the network, Fonseca et al [128] add a second controller which takes control of the network in the event of primary controller failure. The system replicates updates to the primary controller to the secondary ensuring they both maintain the same view of the state of the network and policies to provide smooth transition in the event of primary controller failure. The Kandoo system [129] adds several more controllers but looks at increasing scalability without handling routing. Kandoo employs a root controller which continues to maintain the global view of the network and deploys a set of local controllers near the switches that handle network events close to the switches to take some of the load off the root controller. The local controllers do not have a global view of the network and as such cannot handle routing. They receive all the network updates sent by switches and pass on any relevant ones to the root controller. Instead of a master-slave architecture, Tootoonchian and Ganjali [130] move to a more resilient peer to peer architecture with HyperFlow. The HyperFlow system describes a control plane which has several controllers distributed around the network. Each switch is connected to the controller nearest to it and the network can hold as many controllers as necessary. Each controller can read and write to all switches (including ones they are not directly connected to) by sending messages to the other controllers.

In addition to these solutions, several open sourced controllers sought to address this problem, providing inbuilt mechanisms for distribution and load balancing. OpenDaylight [37], ONOS [131] and ONIX [132] all offer intrinsically distributed control planes. ONIX provides a distributed view of the network to each controller instance by partitioning the Network Information Base (NIB) and assigning responsibility for a portion of the NIB to each controller instance. ONOS and OpenDaylight similarly provide a distributed (but logically centralized) global network view to all its applications and controller instances which is regularly updated by the instances in the cluster. ONOS also provides load balancing mechanisms that ensure the switches are fairly managed among the controllers in the cluster.

While these works focus on distribution of the control plane, none of them explicitly consider security. Some researchers have since taken the distribution a step further by taking into account the actions of a malicious adversary specifically aiming to overload the controller to cripple the network. [133] proposes a system for mitigating controller overload by attackers by employing a pool of controllers and switching to idle ones when one becomes overloaded. In the event of a controller receiving flow requests at a rate which exceeds a certain threshold, the defence system instructs the switch to select a new controller from the pool while attempting to filter out the attack packets. Similarly, [134] proposes a method for distributing controller load by allowing dynamic mapping between switches and controllers. Each switch has several controllers connected to it in a Master-Slave configuration. In the event of the Master controller being overloaded, it tells another controller to take over as the Master controller. Instead of a middlebox defense system as, [135] uses a master controller which is selected from among all present controllers. The role of the master controller is to monitor the network load on each other controller and switch. If a failure or traffic change is detected, the master controller re-organises the switch-controller mappings to better balance the load.

The systems carry out their load balancing very differently, however. In [134], the controller under attack must tell another controller to take over. This takes system takes 6 round trip messages to complete the handover which may be impractical under an attack which causes congestion in the controller-switch communication channel. Additionally, unless CPU resource is reserved for the defences system

in the controller, the attack may set upon the controller with such ferocity that it is unable to notify another controller about the problem before it goes down. By contrast, [133] has a dedicated monitor for the controller resources and sends only 2 messages in the event of an attack (one to start filtering and one to switch controllers) and so the mitigation method is more likely to work here. Though less so, it is still vulnerable to undelivered messages due to congestion in the channel. Instead of involving the switch in the reassignments, remapping in [135] takes place entirely within the control plane by the master controller and by that virtue, may be the best of the methods put forth here.

2.11 Summary

This chapter aims to present as a backdrop, past and current related work on the state of the research field with regards to Software Defined Networks and Denial of Service attacks within them.

We began by looking at early networks and the need for routing protocols as the networks grew. The evolution of routing protocols which arose as a means to ensure packets and traffic arrived at their intended destination sets the stage for programmable networks and SDN. As the routing protocols evolved, researchers began to toy with the idea of removing the routing system from the forwarding device and creating a centralized intelligent device which made decisions on the paths packets travelled. This began to give rise to the concept of Programmable Networks.

We also noted the progression from Programmable Networks to the modern SDN concept with the OpenFlow Protocol. This protocol helped specify the interface between the now separated control and data planes in the network and brought in vendor support as it allowed them to maintain their proprietary internal switch circuits while exposing a well defined interface. Within the OpenFlow Protocol we noted its progression as newer versions added more functionality. Particularly we aim to focus on functionality added in version 1.4 and later which allows the switch to actively evict flow rules from its table to make room for newer ones.

We explore briefly the concept of DoS attacks on networks and why it has become a notable concern for all networks. We see many solutions put forward for

these attacks but note that no solution has been hailed as a complete solution which eradicates the problem, therefore the research field is still open. We look at DoS vulnerabilities which have arisen within the SDN paradigm and note that despite its many benefits, SDN exposes new DoS vulnerabilities intrinsic to this specific type of network. Several pieces of work have been put forth in an attempt to mitigate this issue, however once again, none have been regarded as a complete solution. We take particular note here of solutions which aim to augment the switch's intelligence to aid in defence and in this thesis, focus our efforts on that space.

To conclude, we note two main points:

- DoS attacks continue to be a prevalent issue in all types of networks
- SDN networks expose new vulnerabilities for DoS attacks which cannot be mitigated in the same way attacks on traditional networks are

2.11.1 Moving Forward

The notion of moving away from the "dumb forwarding device" SDN switch concept and towards one in which the switch is a smarter and more active participant in the network defense has been fielded by several works as seen in Section2.9. While not widely implemented, these works undeniably indicate that switch based intelligence merits further study and may provide more effective solutions to SDN vulnerabilities (such as DoS) than the current "mutually exclusive intelligence" paradigm.

With this in mind, we propose that intelligent SDN switches can work alongside the controller to increase the network resilience. We begin by analysing the most widely implemented mechanism which increases switch intelligence: Switch Based Flow Rule Eviction. Section 2.7.6 discusses the issue of the limited SDN switch Flow Table and how this can be exploited by attackers to deny service to legitimate traffic attempting to traverse the network. As we discuss in the following chapter, Switch Based Flow Rule Eviction alleviates some of the risk of this threat by allowing the switch to automatically remove flow rules from its table when it becomes full. We perform in depth analysis of this OpenFlow modification, analysing both the concept and implementation from a malicious user's perspective.

To further explore the proposal that smarter SDN switches can be beneficial to the network, we propose designs for an intelligent switch which adds resilience to DoS and implement two modules which demonstrate the effectiveness of such modifications to the switch. Through extensive evaluations, we determine whether the proposal of increased resilience through intelligence in the forwarding plane holds its weight and is a viable route to explore for widespread implementation.

In the subsequent chapters, we ask the following questions:

- What are the benefits gained from Switch Based Flow Rule Eviction?
- Does the current implementation allow the network to better defend itself against Table Overflow DoS issues?
- Are there alternative implementations which may better defend the network?
- What further modifications to the switch can increase resiliency?
- What are the notable effects of some of these potential further augmentations?

Chapter 3

An Analysis of Flow Eviction Strategies

3.1 Introduction

The traditional SDN implementation which separates the control and data planes concentrates the intelligence of the network into the control plane and reduces the switches to dumb forwarding devices [95][136]. While this paradigm has been shown to provide many benefits which traditional networks do not, it has also been shown to carry many of its own shortcomings. The previous chapter discussed some of these shortcomings, particularly various new methods of Denial of Service attacks which this network model facilitates. Due to this, new defenses are required to protect the network from malicious use.

As we also discussed in the Chapter 2, recent versions of SDN's defacto protocol OpenFlow have deviated slightly from the traditional SDN paradigm, increasing the switch's intelligence to perform autonomous removal of flow rules. We present in this chapter, an in-depth analysis of the initial steps toward more intelligent SDN switches, defining "switch intelligence" as the switch's ability to autonomously perform tasks without depending on the controller. Later in this section, we revisit the Table Overflow issue in SDN switches and discuss the benefits presented by switch based flow rule eviction. We also discuss the concept of "First Packet Delays" which causes additional latency on the first packet of the flow in SDN networks. In Section 3.2, we examine the potential for new DoS attacks which exploit the first packet delay coupled with the user's ability to force rules out of the flow table.

We discuss the opportunity for alternative flow rule eviction policies and potential further attacks on these alternative policies in Sections 3.3 and 3.4 and analyse the resiliency of the alternative policies for flow rule eviction in Section 3.5 to determine which provides the best protection for the SDN network. Finally, we discuss the implications of attacks such as these occurring in the real world in Section 3.6.

3.1.1 Table Overflow

The opportunity for Table Overflow attacks has been explored as one of the major shortcomings of SDN in the previous chapter. Each flow of traffic in an SDN network requires a matching rule in the switches it encounters in order to traverse the network. TCAM memory, which SDN switch Flow Tables are built on, is typically very expensive, leading manufacturers to restrict the typical Flow Table size to several thousand rules. This makes the Flow Table a limited resource and these relatively small Flow Tables can easily become filled to capacity with flow rules. Once this occurs, no new flow rules can be added to the switch which restricts any new traffic from being routed through the switch until the flows expire or are explicitly removed by the controller.

As of OpenFlow 1.3, the latest widely deployed version of the OpenFlow protocol, the only way to immediately rectify a Table Overflow is to have the controller actively remove flow rules from the Flow Table before inserting new rules into the table. In such a situation, the following procedure may take place:

- The controller issues a Flow Mod command to add a new flow rule
- The switch informs the controller the table is at capacity with a “Table Full” error
- The controller requests a list of flow rules currently occupying the Flow Tables
- The switch responds with a list of the flow rules currently occupying the Flow Table
- The controller selects one to remove and issues the flow removal command
- Finally, the controller re-issues the flow to be added.

In this worst case scenario, the process adds an extra 5 steps to add the flow rule placing additional stress on the controller, which is already a potential bottleneck in the SDN architecture. Such a method is wholly inefficient as it increases the work the controller must do as well as the number of messages which must be sent between the controller and the switch.

3.1.2 Switch Based Flow Rule Eviction

Openflow 1.4 introduced Switch Based Flow Rule Eviction. This configuration enables the switch itself to remove a flow rule in the event of an incoming Flow Add command from the controller in the presence a full Flow Table. In doing so, while the switch Flow Table is still a limited resource, the network is better able to apportion this resources to cater for demand and provide better service.

Autonomous flow rule eviction in the switch increases switch intelligence and in doing so presents several benefits to the SDN network.

1. By enabling the switch to make a decision and perform an action, it reduces its dependency on the controller for operation.

2. There is also less delay between the moment the action is needed and the moment it is performed. By allowing the switch to perform the 5 extra steps itself, the deletion of a flow rule for space and insertion of the incoming flow rule occur almost immediately. This reduces the number of packets dropped in the interim.

3. Along the same lines, it reduces the number of messages between the switch and the controller. In a system in which the bandwidth between the controller and the switch can be a precious resource, a 6x increase in the number of messages between the switch and the controller is a wholly inefficient situation. By enabling the switch to perform its own evictions, there is no need to send these messages, freeing the switch-controller communication channel to handle more critical messages.

4. Finally, it also reduces controller load. The controller no longer needs to process the responses sent by the switch, determine which flow rule would be best to remove and issue additional commands. It instead can focus on

more controller specific tasks such as determining routes for flows through the network and maintaining flow policies.

In some respects, Switch Based Flow Rule Eviction can be presented as a solution to Table Overflow DoS attacks. Flow rules can now be forcibly removed without excessive overhead and therefore attackers can no longer hold the Flow Table hostage preventing traffic from being routed. In this way, Switch Based Flow Rule Eviction represents a change in direction of the SDN paradigm by increasing intelligence of the traditionally "dumb forwarding device". Given the range of benefits achieved by moving this one action out of the controller and into the switch, it strengthens the argument that smarter switches can provide untold benefits for the SDN network.

3.1.3 "First Packet" Delays

Forwarding in SDN can be realised in two forms, known as Proactive and Reactive. In Proactive SDN flow rules are installed in the switch in anticipation of the traffic which will arrive at the switch. This method reduces the load on the controller and ensures that traffic proceeds through the network unhindered by the controller interaction intrinsic to Reactive SDN. However, Proactive SDN can be restrictive in that any traffic without a pre-installed rule is unable to traverse the network. In Reactive SDN, flow rules are installed in response to the traffic demands. It is this form of the paradigm on which we focus in this thesis.

In Reactive SDN, a switch Flow Table does not contain rules for flows before the flow arrives at the switch. Thus, the first packet of a flow arriving at a switch has no matching flow rule to instruct the switch on how to forward it, and so triggers a "Table-Miss". As a result of this table-miss, the switch forwards the packet to the controller for instructions on how to route similar packets. The controller determines the best route for the packet and others of the same flow through the switch and inserts a flow rule into the switch to handle further packets of the same flow. In being diverted to the controller for instruction instead of passing directly through the switch, the packet causing the table-miss registers a significant delay in arriving to its target. Depending on the controller, the CPU power and the distance between the switch and the controller, this "first packet delay" can cause the packet's latency to grow significantly. Under normal circumstances, it is only the first packet of the flow that registers this delay (of the order of ms) [137] with the remaining packets

of the same flow being processed at line rate.

3.2 Analysis of the Switch Flow Rule Eviction Attack Vector

As discussed previously, switch flow rule eviction represents a step towards increasing the switch intelligence and presents several benefits. Its implementation in OpenFlow switches enables the switch to remove the oldest flow in favour of the incoming flow rule- a First In First Out (FIFO) policy. While eviction is an excellent strategy in the right direction, FIFO is a poor choice of policy both from a security and network performance perspective. In mitigating one issue, it presents a new vector for attack which can have similarly devastating consequences.

By requesting flow rules to be placed into the Flow Table to facilitate his/her traffic, any network user has the power to influence the state of the switch Flow Table at any time. It is this ability of the user to influence the state of the Flow Table coupled with the first packet delay concept which provides a new vector for attack within the SDN/Openflow network. Leveraging the newly introduced flow rule eviction mechanism, a malicious user can forcibly remove legitimate flow rules from the switch. Under the FIFO policy, the oldest rule in the switch is removed to make room for new flow rules to be inserted. A malicious user who requests enough flow rules to fill the Flow Table after a legitimate flow rule has been inserted causes the legitimate rule to become the oldest flow rule and thus causes it to be removed from the switch. This causes the next packet of the legitimate rule that arrives at the switch to experience another "first packet delay" since there is no longer a matching rule for it. The more evictions of the legitimate flow rule(s) the malicious user can cause, the more packets of the legitimate flows register first packet delays giving the aggregate flow a higher latency and reducing the overall throughput of the flow (Note, we use the terms "throughput" and "bandwidth" interchangeably throughout this chapter).

3.2.1 Analysis Of The Effects Of Delayed Packets

To illustrate the effects of first packet delays being spread over a large portion of the packets in the flow, we compare the throughput achieved when, as in normal operation, only the first packet[1] is sent to the controller and all other packets forwarded at line rate through the switch vs the throughput when all packets are sent to the controller.

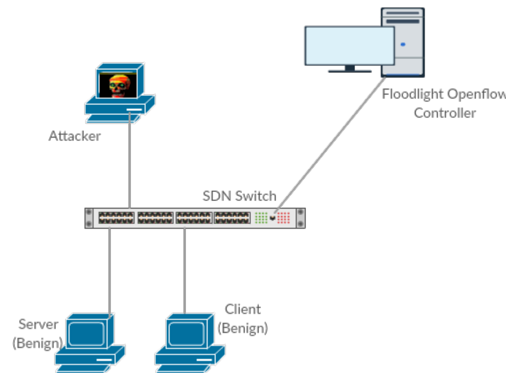


Figure 3.1: Network Setup

We perform these experiments using the setup displayed in Fig 3.1. For the switch, we use both a Pica8 3297 switch and an Open vSwitch instance running on a desktop machine with quad-core i5-3570 processors and 32 GB of RAM running 64 bit Centos 7 Linux Operating System. For the controller, we use a Floodlight controller running on a desktop similar to that of the OVS machine. Connected to the switch are 3 hosts- 2 benign (Client and Server) and one malicious. We exclude the malicious host from this experiment as it does nothing here, however this setup will be consistent throughout our remaining experiments, so it is included here.

Using the Pica8 Switch, we first configure the Floodlight controller as normal, installing a flow rule to the switch upon request. This means that only the first

[1]While the SDN paradigm indicates that, in the case of Reactive Forwarding, it is only the first packet of the flow which is sent to the controller, depending on the packet arrival rate, it may actually be the first several packets which are sent to the controller. After the first packet is sent, several other packets may arrive at the switch while the switch awaits the rule insertion triggered by the first packet's table-miss. Since these subsequent packets also arrive to no rule, they are also sent to the controller until the rule is inserted.

packet registers the delay and the rest are forwarded using the flow rule at line rate (normal network behaviour). We use the *iperf* [138] tool to measure the throughput between the client and server under these conditions. We then configure the Floodlight controller to forward all packets to their destinations without installing corresponding rules in the switch. Since there is never a matching rule installed in the switch, all packets of a flow will be forwarded to the controller registering a delay. This is the opposite of normal network behaviour and illustrates how an attacker can influence the network to reduce the Quality of Service. We then use the *iperf* tool again to measure the throughput between the client and server. Finally, we substitute the Pica8 switch for the OVS instance and perform the same experiment. The results are displayed in Fig 3.2. From the graph, we see that the difference between sending all packets but the first through the switch (a flow rule is in the switch) vs sending all packets to the controller (no flow rules in the switch) achieves a throughput reduction of almost 99% with the Pica8 Switch and 91% with the OVS instance (reduced QoS). Wang et al [100] note the deficiencies of hardware SDN switches generating flow requests which is the likely cause for the differences in the OVS and Pica8 results. Given the extreme results of both, however, the aim of the attacker is to cause as many packets as possible to be sent to the controller.

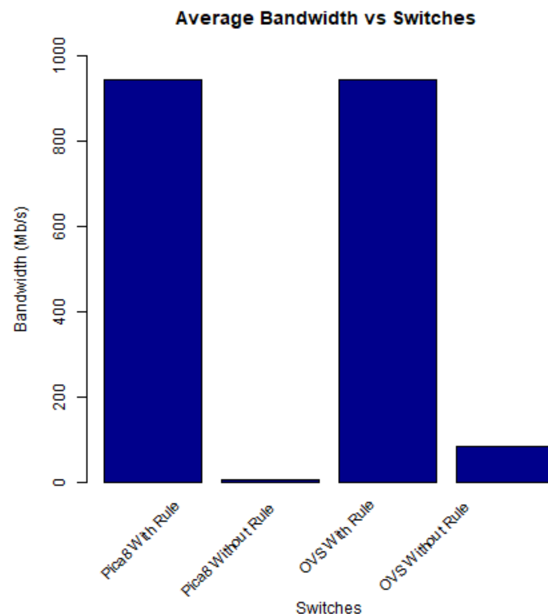


Figure 3.2: Bandwidth With and Without Matching Flow Rules

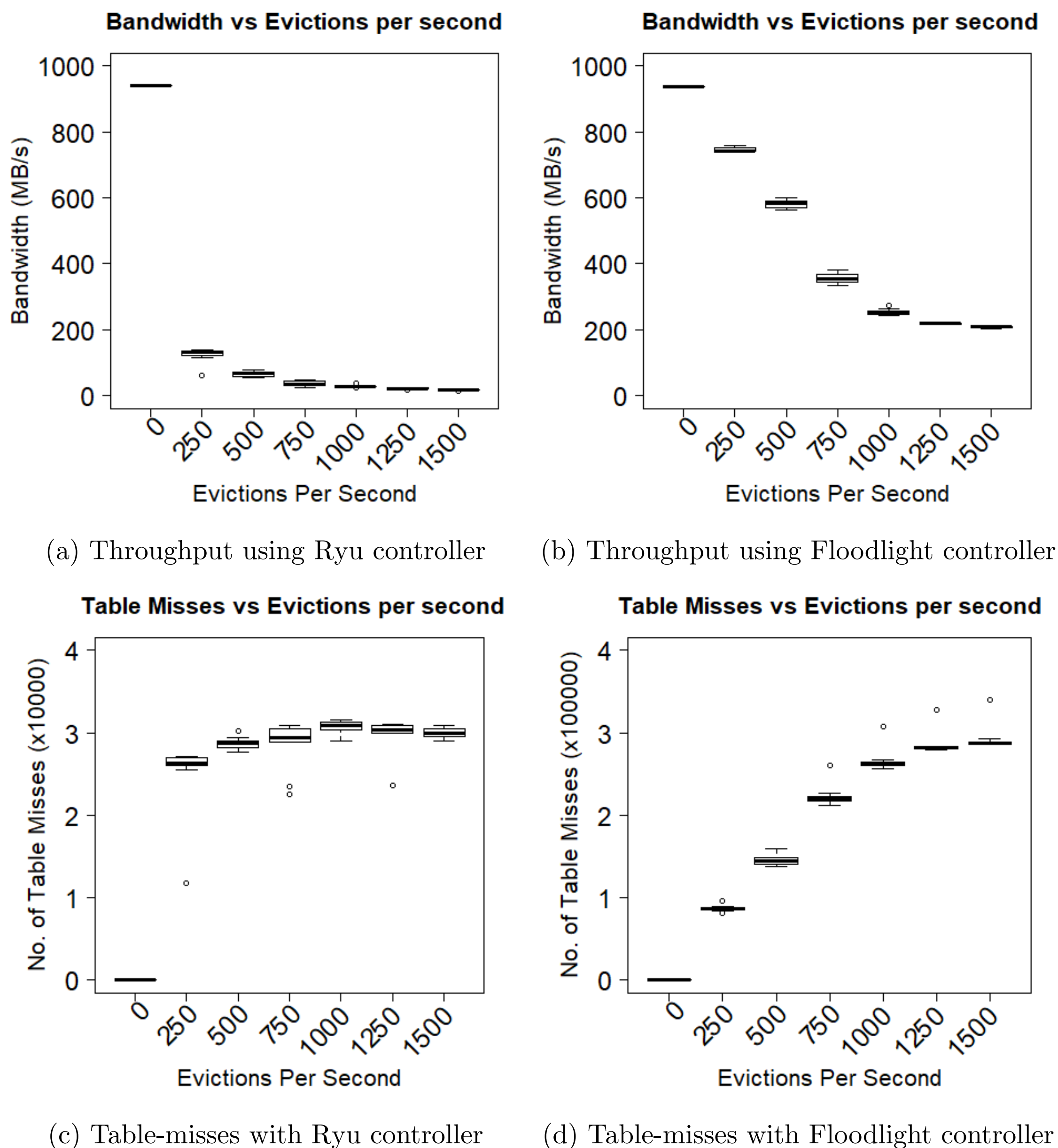
3.2.1.1 Frequency of Rule Removal and the Controller Effect

We perform further analysis on the effects of packet delays on the throughput experienced by the flow by examining two factors here: The controller in use and the frequency at which table-misses occur.

The previous section demonstrated that if all the packets are forced to go to the controller, the throughput is drastically lower in comparison to all the packets going through the switch without diversion. To examine the effects of the regular absence of a rule more closely, we extend the Open VSwitch instance to regularly remove the rules for our experimental flow at a prescribed frequency. Varying frequencies alter the subset of packets diverted to the controller due to absence of a rule. The more often a rule is missing, the more packets are sent to the controller.

Additionally, we investigate the effect of the controller responding to the diverted packets. The speed at which the controller can receive and process a request, and install the corresponding flow affects the amount of time the diversion to the controller adds to the latency of the packet. This factor is influenced by variables such as the distance between the controller and the switch, the programming of the controller (e.g single threaded vs multi threaded) and the underlying hardware the controller process uses. Controllers which are able to quickly process and forward packet_ins record minimal delays in packet diversions to the control plane while controllers which take longer can increase the effect of the attack. We therefore compare the effects of different controllers on the throughput experienced by a flow receiving frequent evictions.

Using the network setup in Figure 3.1, we again use the *iperf* tool to measure the throughput between the client and server, with the Open VSwitch instance regularly removing the corresponding flow rule (causing a subset of the packets to be diverted to the controller). In each instance, we also measure the number of packets diverted to the controller. To compare the controller effects, we first measure the throughput and table-misses of each frequency of rule removal with the switch connected to a Ryu controller and then the Floodlight controller used for the previous experiments. Both controllers use the same controller station, keeping the underlying hardware consistent between them. The results are displayed in Figures 3.3a-3.3d.



(a) Throughput using Ryu controller

(b) Throughput using Floodlight controller

(c) Table-misses with Ryu controller

(d) Table-misses with Floodlight controller

Figure 3.3: Frequent Evictions with varying controllers

In both experiments, the steady increase of evictions produces an increase in table misses and a steady decrease in the registered throughputs. The results illustrate a notable relationship between evictions, tablemisses and throughputs but indicate a significant difference between the two controllers. The Floodlight controller provides far better service than Ryu (consistent with other studies such as [139][140]) which results in the same rate of evictions having less of an effect in the network using the Floodlight controller than the one using Ryu. The slower Ryu controller not only

causes an increase in the diversion time, but also likely causes more dropped and out of order packets which need to be resent, which causes TCP bandwidth throttling, all of which contribute to a reduced throughput. The number of Table misses using the Ryu controller is notably less than the Floodlight controller and fewer Table misses appear to produce a higher effect on the throughput. This is likely due to the TCP bandwidth throttling effect in that as the throughput drops so steeply, the source begins sending fewer packets to adjust.

In summary, we analyse here the effects on network throughput of packets being diverted to the controller to show that if an attacker can divert a significant amount of packets of a flow to the controller, he can affect the throughput the user experiences. We show that the numbers of packets being diverted to the controller is inversely related to throughput of the flow. Additionally, the capacity and performance of the controller connected to the switch plays a large role in the effects of the packet diversions. Better performing controllers reduce the negative effects, providing better throughput by minimising the latency caused by the diversion to the controller. Lesser performing controllers increase the diversion's effect by not only increasing the latency of table misses but causing a ripple effect of increased packet dropping and resending, all of which drives down the bandwidth.

3.2.2 Attacker Model

Before looking at the potential attacks, we discuss the attacker model and describe what the attacker is and is not capable of. We make the following assumptions about the attacker.

- We assume the attacker(s) is a regular user of the network who does not have admin access to the network equipment or any hosts on the network except the ones they have compromised.
- The attacker may have compromised or otherwise has root access to a small subset of the hosts in the SDN network excluding the hosts whose throughput they are attempting to control.
- The attacker is not able to directly access or control the physical switches or controllers but is able to influence the state of the switch Flow Table using traffic from the hosts they control.

These assumptions show that our attacker is a relatively standard user of the network with no special abilities and a secure SDN network should have the robustness and intelligence to handle a user (or small subset) who suddenly turn malicious.

3.2.3 Attacking the FIFO Policy: The Spray Attack

We therefore propose an attack on the FIFO flow eviction policy. Assuming a Flow Table which holds N rules, the Spray Attack, as we call it from here on, involves sending a single packet to N or more destinations. This causes the Flow Table to be filled with the attacker's rules, removing any rule in place prior to the attacker sending his/her packets. We call this "flushing" the Flow Table. We assume here that the attacker is able to accurately determine the size of the Flow Table in order to effectively flush it. Several works including [94] and [92] discuss how the attacker can determine the Flow Table size using side channel attacks based on packet response times. In addition to this, the attacker must be aware of what header fields need to be varied in order to trigger a table-miss and a new rule installation. Techniques to learn such are discussed in [93] and [92].

Having seen the effect of all packets being sent to the controller, the attacker aims to cause a subset of the legitimate flow's packets to be sent to the controller by frequently filling the Flow Table with malicious rules and causing all legitimate flows to be evicted (flushing). Each time the attacker flushes the Flow Table, the legitimate rules need to be requested again by the next packet of their flow, inducing a delay for that packet. The more frequently the attacker flushes the Flow Table, the higher number of table-misses and lower the overall throughput of the flow. We note that this attack is specifically targeted at long lived flows which carry a high number of packets (elephant flows) and often contribute to a large portion of network traffic [141]. It is less likely to affect short lived flows which have far fewer packets and run for shorter times (mice flows).

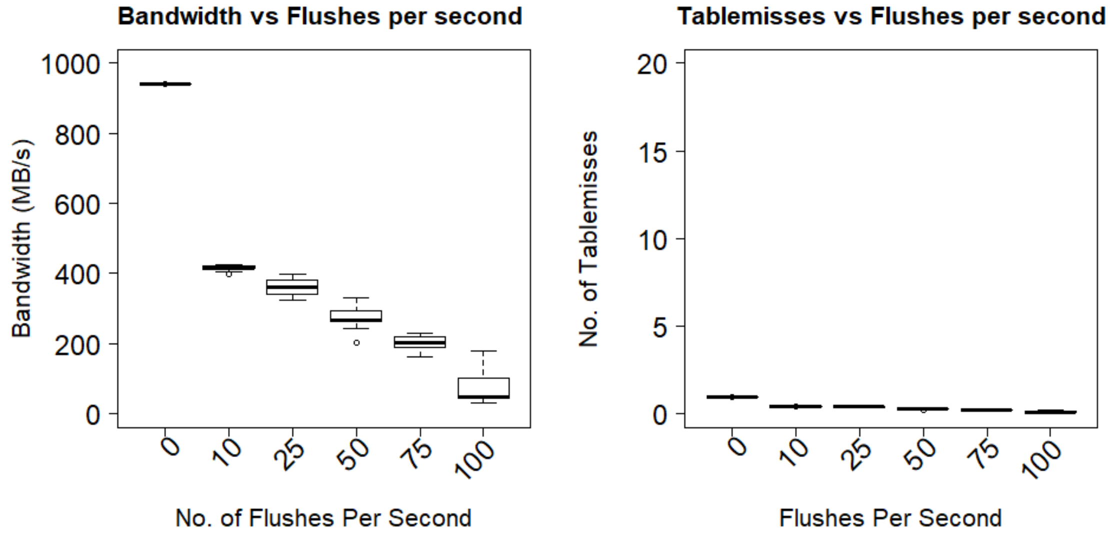
Attack Implementation: The Floodlight controller used in our experimental network is configured to insert flow rules with an in_port, source IP Address and Destination IP address tuple. We therefore create a *python* script using the *scapy* libraries [142] which creates and sends packets with varying destination IP addresses. The script takes as parameters two values: the number of different IP address des-

tinations for which packets should be generated and a "sleep" value which allows us to control the frequency at which the Flow Table is flushed. The script generates the packets according to the specified number of destinations, sends the packets to the switch and then goes to sleep, waking several milliseconds later to send the list of packets again.

3.2.4 Spray Attack Proof of Concept

As an initial proof of concept (we perform more in depth evaluation later in this chapter), we perform the attack on the network setup in Fig 3.1. We introduce into this experiment the attacker, which was not used in the previous experiment. In this, and all further experiments, the Floodlight controller is configured as normal to install flow rules into the Flow Table in response to flow requests. We again use the *iperf* tool to measure the throughput between the client and the server, representative of any two hosts on the network. We measure the throughput between them without the attack and then with the attack at various flushing frequencies Figure 3.4 displays the findings.

We see that at varying rates, the attacker is able to steadily reduce the throughput experienced between the two hosts. By increasing the number of times the table is flushed, the attacker increases the number of table-misses. We note the correlation between the number of table-misses and the throughput, that as the former rises, the latter decreases. As the attack intensifies, the attacker is able to reduce the throughput nearly 90%. More importantly, using the attack, the attacker is able to finely tune the throughput experienced between the client and server as they desire by adjusting the rate of their attack. However we also note that as the number of flushes increase, the precision to which the client throughput is tuned decreases. This is shown in the range of bandwidths experienced at 100 Flushes Per Second (FPS) vs that of 10 Flushes Per Second (FPS). As the frequency of flushing increases, so too does the chance of an entire flush occurring between two consecutive packets of a user's flow which would have no effect on the flow.



(a) Legitimate user's Bandwidth during Spray Attack

(b) Number of Table Misses during Spray Attack

Figure 3.4: Spray attack effects

3.2.5 Limitations

The success of the attack is dependent on 2 factors outside of the attacker's control: The Control Plane capacity and the Switch OFA capacity.

Control Plane Capacity: The ability to install flows quickly is essential to the attack, therefore the control plane must be able to keep up with the rate of flow requests coming from the attacker. Several works have discussed the issue of control plane bottle necking and the constraints it places on the network (e.g [143][144]). Several works have also proposed various multi controller architectures to alleviate the bottleneck issue (e.g [100]) by distributing the work of the control plane. This attack assumes that the control plane, whether by load distribution or sheer CPU power, is able to handle the flow requests it receives and perform flow rule installations at the same rate it receives them. If this is not the case, the attack is then restricted to whatever rate the control plane manages to output rather than the intended rate of the attacker. This dependency also affects the benign user, however, since their flow re-installation request must now go to a back-logged control plane. This will increase the time it takes to have the benign flow re-installed, increasing the number of packets which are forwarded to the controller as a result of a missing

flow rule. We use a single floodlight controller for this experiment which performs flow installations at a rate of roughly 4K flows per second. As such, we restrict the table size to 10 flows to ensure the controller is able to match the flow request easily. We examine the effects of larger table sizes later and assume that the control plane implemented in more realistically sized networks are capable of handling their flow installation rates for the attacker.

Switch OFA capacity: Similarly, the ability of the switch OFA to install flows as fast as the controller issues them is a key part of the attack. Once again, if the attacker's flows are not being installed at the intended rate, the benign flow's throughput may not be reduced as much as intended. While Open vSwitch has been shown to install rules at rates in the order of milliseconds, several physical switches have been shown to be limited to a few hundred rules per second [100]. This also works against the benign users as the OFA is not only responsible for installing the attack flows, but for re-installing the benign flow rules as well. As the OFA's buffer becomes filled to capacity with flow rule installation instructions, it begins to drop the commands, causing lost packets which the benign hosts will need to resend, further reducing the quality of service experienced by the benign users in the network.

3.3 Alternative Policies for Eviction

Given these potential ramifications of implementing a FIFO eviction policy, we ask what alternative options for rule evictions can SDN switch designers turn to to provide better performance and attack-resilience? Referencing the work done in the field of CPU cache-design (e.g [145][146][147]), we consider several other policies for evicting rules held in the switch Flow Table and evaluate our previously discussed Spray attack on each as well as a second attack, the Clog Attack.

3.3.1 Least Recently Used

Rather than removing rules according to the time they were inserted, the LRU (Least Recently Used) policy aims to retain heavy-hitting rules in the Flow Table by removing from the table the flow rule with the longest time since its last hit. In

doing so, it attempts to predict which rules will be the most useful to the network and remove the ones which are least useful.

3.3.2 Least Frequently Used

Similar to LRU, LFU (Least Frequently Used) is aimed at keeping the most valuable rules in the switch Flow Table. This policy rates "usefulness" by how often the rule has matched in its time in the table rather than how recently it has been matched. Since regular network traffic follows a Zipf distribution [148], the LFU or LRU policies may be better suited to a switch attempting to retain its most used rules.

3.3.3 Random

Alternatively, a non-deterministic rule eviction policy such as Random rule eviction may be the key to attack resilience as it minimizes the attackers control of which rule is evicted. Adhering to this policy means the switch selects, at random, a rule to be removed from the Flow Table in the event of a full table, regardless of the time the rule was placed in the switch or the number of packets it has matched.

At the core of the Spray attack principle is the ability of a user to use malicious rules/rule requests to influence the switch to remove legitimate rules from its Flow Table. Since policies such as LFU and LRU are designed counter-intuitively to FIFO in that they retain flow rules which have the greatest value irrespective of "time of insertion", the Spray attack is unlikely to have the same effect on them. Instead, we propose a second, more effective "Clog Attack" for volume-based eviction policies, in which a malicious user aims to make his/her rules appear to be the most valuable rules in the network, causing legitimate rules to be evicted instead.

3.4 Attacking Volume Based Eviction Policies: The Clog Attack

Least Frequently Used and Least Recently Used both look at the packets hitting a flow rule. The Clog attack specifically targets these two by sending a constant stream of packets across several flows in order to “clog” the Flow Table. With this stream of packets, in LFU, the aim is to ensure his rules have the highest packet counts for their time within the table, i.e his rules are the most frequently used; in LRU the aim is to ensure that all his rules have a smaller delay between hits than the legitimate traffic, ensuring his rules are always the most recently used. By ensuring that his rules are the most frequently used or most recently used rules, the attack hopes to cause legitimate rules to be evicted when a new rule is inserted. If done correctly, the attacker is able to successfully reduce the amount of Flow Table space the legitimate traffic has available to it (hence the name “Clog”) and so any new legitimate rule which needs to be inserted into the Flow Table will cause another legitimate rule to be removed. By constantly streaming traffic through a subset of the rule space, the attacker creates an illusion for the switch that his rules are most valuable and should not be removed. In this way, any other traffic attempting to traverse the network is forced to make use of whatever rule space is left (the unclogged subset of the Flow Table) causing high rates of removal and churn as the switch evicts and re-installs rules.

Clog attack implementation We implement the clog attack using the *trafgen* tool -a network tool (part of the *netsniff-ng* toolkit) useful for generating large quantities of packets [149]. We configure the *trafgen* tool to generate streams of packets with each stream having a unique destination address in order to trigger a new rule. We build a script around the *trafgen* tool which takes as a parameter the number of streams (rules) we would like to generate. It then accordingly creates packets for each stream and sends these packets to the switch as fast as the CPU and bandwidth allow. With the attacking host on a machine similar to that of the Open VSwitch, a single attacker is able to generate between 530000 and 540000 packets per second.

We additionally create a variable stream of packets with a unique destination IP address using *python* and *scapy* libraries. This allows us to control the number

of packets sent per second through this flow rule. In doing so, we hold a portion of the Flow Table using our malicious streams and using our variable rule, increase the "competition" for the remaining "unclogged" portion of the rule table. The variable rule ensures that flow requests are made which cause the legitimate rules to be evicted from their spaces.

3.5 Resilience Analysis of various eviction policies

Previous discussions highlighted the benefits of switch based flow rule eviction but demonstrated that the chosen policy (FIFO) presented a new vector for DoS attacks. Through extensive analysis, we aim to determine here which eviction policies provide better resiliency in the face of attacks and may prove a better choice than FIFO for the implementation of switch based flow rule eviction.

Neither Open vSwitch nor Physical switches are optimized for these alternative flow rule eviction policies therefore we perform an in-depth evaluation by way of simulation of the effects of the attack on each rule eviction strategy. While we use a table size of 10 spaces previously, many would argue that this is an unrealistic size as most SDN network tables are significantly larger. However, an attacker performing these attacks on a larger Flow Table must send significantly more traffic to the controller. That is to say, while 10 flow requests may successfully flush or clog the Flow Table here, at a Flow Table size of 500 spaces, 500 flow requests are required to flush the table. By increasing the number of packet-ins in this way, our experiments begin to stray into the "Control Plane Saturation" attack which has already been discussed at length (e.g [39][96][97]).

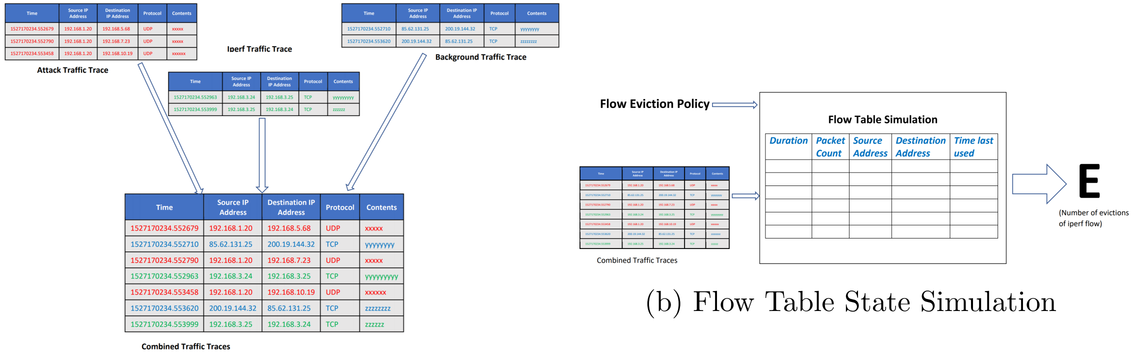
Because we aim to look specifically at the effects of eviction, we perform a simulation of the attack rather than a live performance of it. This allows us to increase the table size while still focusing on the eviction problem. With our simulation, as described here, we remove the strain of servicing the attack flows from the controller. We acknowledge in Section 3.2.5 that the ability of the controller to install the attack flows at the rate the attacker requires is essential to the success of the attack and we make the assumption that an attacker attacking a larger network has such a control plane at his disposal.

The attack simulation process is carried out in three (3) parts (illustrated in Figure 3.5a- Figure 3.5c).

1. **Traffic Capture**: We begin by combining attack traffic, an *iperf* flow and some background traffic into a single pcap trace file. We capture the attack traffic by having the attacker perform the required attack for 60 seconds and capturing the traffic sent (by the attacker) using tcpdump at the switch. We also capture the *iperf* flow in a similar fashion by using tcpdump at the switch while the client and server perform a 60 second *iperf* flow. Finally, we also take a subset of a caida dataset [150] of varying sizes (emulating networks of various sizes) to add surrounding background traffic. We combine all three traffic traces into a single trace file, aligning their start times appropriately to produce one 60 second traffic trace which represents all the traffic at the switch while an attack is in progress.

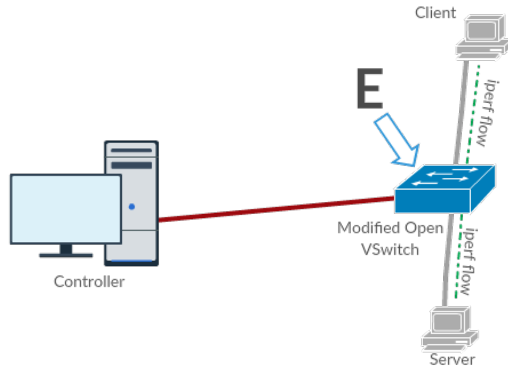
2. **Flow Table State Simulation**: Using a simple Java script, we read in each packet of the network trace and simulate the state of the Flow Table (represented by a HashTable structure) after each packet. As each packet is read in, the script checks the Flow Table for a matching flow rule as the switch would do. If the rule exists, it updates the rule's statistics accordingly (e.g number of packets rule has seen). If no matching flow rule exists, the script selects a rule for eviction from the Flow Table according to the current eviction policy and removes the rule, adding a new rule to the Flow Table to match the current packet as the controller would do. The output of this simulation gives the number of times, E, the experimental *iperf* flow (representative of any benign elephant flow) was removed from the Flow Table during the attack.

3. **Eviction**: Having determined the number of times the *iperf* flow rule would be removed under attack, we record the effect of these removals on the flow throughput. We modify Open vSwitch by adding an extra thread which evicts the rules from the Flow Table at a given frequency. For each evaluation here, we set the frequency using the output, E, of its related file. That is to say, we perform the simulation and set the eviction frequency of the Open vSwitch thread to evict the rules E times. We then re-run the 60 second *iperf* flow 10 times between the two benign hosts (client and server) with the modified Open vSwitch evicting its rule at the specified frequency and record the effect on throughput/bandwidth and table-misses.



(a) Traffic Capture

(b) Flow Table State Simulation



(c) Eviction

Figure 3.5: Simulation of the Effects of Frequent Flow Rule Evictions

In this way, we are able to isolate the eviction attack from the "Control Plane Saturation" attack as much as possible by having the control plane only service the benign flow. We are able to study the effects of frequent eviction of a flow by an attacker without actually having to service the attack flows.

Using the simulation procedure we manipulate several variables of the network to evaluate the attack's effect. We manipulate a single variable in each turn, giving it a range of values and hold the others constant for that turn. The constant variables (unless manipulation is indicated) are as follows:

- Attack Power: We hold the Spray attack constant at 300 Flushes per second and the Clog attack at N-1 rules (where N is the size of the Flow Table).
- Number of attackers: For the Spray attack we use 1 attacker and the Clog attack, 7 attackers unless otherwise indicated. We discuss the need for 7 attackers in the Clog attack later.

- Flow Table size: We hold the size of the Flow Table at 500 flow spaces. Our background traffic emulates a network of size 50 hosts which generate 743 flows within a 60 second window. In their study of SDN network performances, Curtis et al [151] recommend 10 flow table spaces per connected host on a switch. This table size is therefore large enough to accommodate such a network without an attack.
- Background Traffic: As indicated previously, we reduce a Caida 2016 traffic trace to a trace containing 50 unique hosts which creates 743 flows without eviction. The caida dataset is a capture from an internet backbone router which holds roughly 700000 flows in the 60 second window. This is far too many flows for an SDN switch TCAM, which typically hold a few tens of thousand flow rules at most, to accommodate [152][151]. Therefore we reduce the dataset to a more manageable size for our experiments.

3.5.1 Attack Power

We begin by measuring the effect of varying the attack power. The aim of the attacks is to frequently remove the victim's rule from the Flow Table causing the switch to request instructions from the controller adding latency to the victim's packets. The more powerful the attack, the more frequently the victim's rule is evicted and the lower their achieved throughput. As previously shown, the Spray attack and the Clog attack accomplish this goal in different ways.

3.5.1.1 Spray attack

The strength of the spray attack lies in how often it flushes the Flow Table, removing the victim's rule. The more frequently the Flow Table is flushed during the victim's flow, the more table-misses the victim registers, the more packets register an above normal latency and the lower the total throughput of the flow. We measure here the effects of varying the frequency at which the attacker attempts to flush the Flow Table. We capture attack traffic from the attacker performing the spray attack at varying rates and combine each instance with the background traffic and *iperf* flow as described previously. We then perform the simulation for each eviction policy and the eviction process. We record the results in Tables 1a-1d in the Appendix and display the bandwidth changes in Figure 3.6.

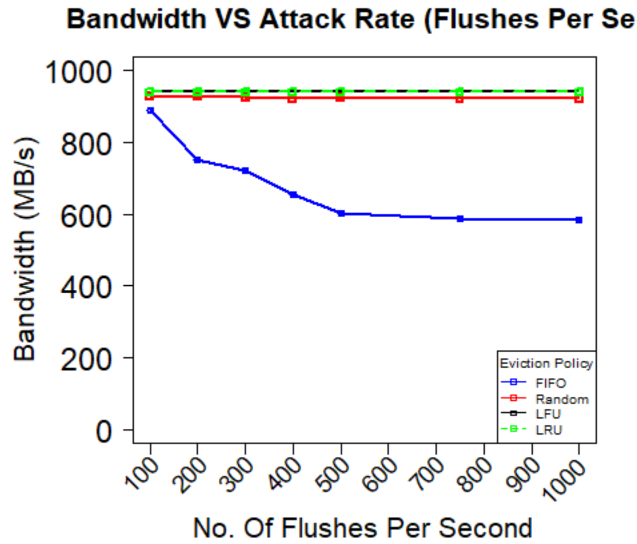


Figure 3.6: Bandwidth Changes under Increasing Spray Attack Power

The results in Table 1a and Figure 3.6 show that, as expected, the Spray attack is particularly effective against the FIFO eviction policy. Under this policy, the throughput is steadily decreased as the rate of flushing increases, reduced by approximately 38% at 1000FPS. However, we see that as the attacker approaches his computational limits at 500 FPS, he is unable to realise his desired output in terms of flushes per second. At this point, the throughput is reduced by 35% and between 500FPS and 1000FPS a throughput decrease of only 3% is recorded. This is due to bandwidth and CPU restrictions on the attacker node so that while the attacker aims for a particular flushing rate, the machine is unable to output the necessary number of packets at the rate he would like.

The experiments also show that on tables of this size, the Spray Attack has little to no effect on the volume based rule eviction policies. At the peak attack rate, 1000FPS, the attack registers no throughput reduction under the LFU and LRU policy. It is almost impossible for a single attacker to make all his rules appear more valuable in the switch flow table using the Spray method. Finally, the use of the non-deterministic Random eviction also sees a very low impact, though not as low as LFU and LRU. The Random Replacement policy registers a mere 2.5% reduction in the legitimate flow's throughput at the highest attack rate. This is likely because

the higher the number of malicious rules in the switch, the higher the chance of a malicious rule being removed rather than the targeted legitimate flow rules. While the attack does not register anywhere near the number of rule removals intended by the rate of flushing, it does see increase in removals as the rate of flushing increases. Despite its randomness, the increase in flow removals gives greater chance that the experimental flow will be the one removed.

3.5.1.2 Clog attack

The Clog attack attempts to cause removal of the victim's rule by holding on to as many flow spaces as it can. This causes the victim's rule to be evicted often to make space for incoming flows which need rules. We measure the effects of varying the number of rules the attacker attempts to hold. In doing so, the attacker's aim is to force the benign traffic to share a limited rule space, inadequate for the traffic. In this experiment, the benign traffic creates 743 flows within its 60 seconds run. The aim of the attacker is for these flows, plus the experimental flow to share the unclogged rule space among them causing frequent evictions.

The *iperf* flow generates packets at a rate of roughly 7050 packets per second. With this in mind, we set a target for the attacker to generate packets for each of his rules at a rate of >7050 packets per second. Using the *trafgen* tool, a single attacker is able to generate between 530000 and 540000 packets per second. To generate more traffic per rule than our benign victim, we limit our attacker to 75 rules which gives approximately 7060 packets per rule per second. For a Flow Table space of 500 flow rules, we therefore use traffic from 7 attackers to clog the table. This fits within our threat model since the background traffic we use comprises approximately 50 hosts, therefore 7 attackers constitutes a "small subset" of the hosts on the network. We show later that fewer than 7 hosts are unable to generate enough traffic across each flow to effectively hold the flow spaces they intend to clog and cause eviction frequently enough to significantly affect throughput.

Traffic is captured from these 7 attackers working in unison to stream through the given number of rules and the attack traffic is merged with the *iperf* flow and background traffic. We then perform the simulation for each eviction policy and the eviction process and record the results in Tables 2a-2d in the Appendix and the

throughput changes in Figure 3.7.

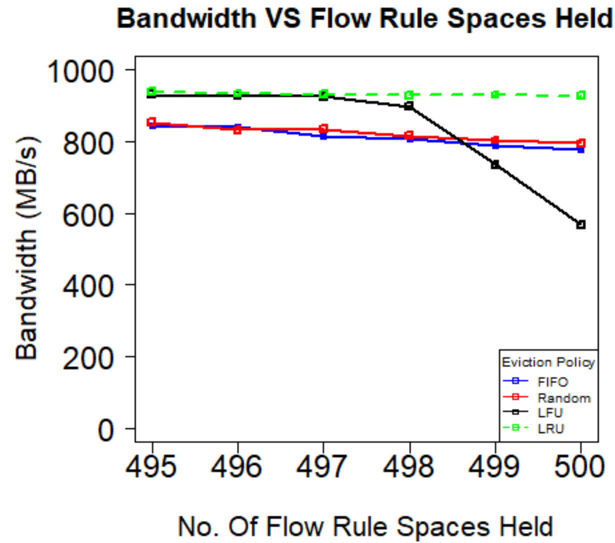


Figure 3.7: Bandwidth Changes under Increasing Clog Attack Power

The experiments show that while the Spray attack has little effect on volume based policies, the Clog Attack sees at least some effect on all policies. In particular, with the LFU policy, the throughput is gradually lowered as the attacker increases the number of rules clogged. As predicted, this forces the experimental flow to fight for rule space with the rest of the traffic at the switch. As the available rule space shrinks, the intensity of the competition for that limited resource grows, increasing the number of evictions. At the height of the attack, the attackers attempt to fill the entire flow table, at which point the LFU policy registers a 40% reduction in the legitimate flow's throughput. However, the same attack sees very little effect on the LRU even at it's maximum, causing a 1.5% throughput reduction when the attackers attempt to fill the entire flow table. To cause the experimental rule to be removed under LRU policy, each attacker would need to have a packet hit each of their 75 rules before the next packet of the experimental rule arrives at the switch. That is to say, between any two packets of the experimental flow arriving at the switch, all 495-500 of the attackers' rules must receive a hit to evict the experimental rule. Due to the packet sending rate of the attacker, this is nigh impossible here and thus, the attack produces little effect. The FIFO and the Random eviction policies both

register small reductions in their throughput under peak attacks, registering 17% and 15.5% reductions respectively.

An attacker can more finely tune the throughput a benign host experiences with the clog attack by clogging a portion of the flow table and varying the number of packets per second sent through the rules in the rest of the table. We demonstrate this using the LFU flow eviction policy, the volume based policy (at which clog attacks are targeted) upon which the clog attack has had the greatest effect. We hold the clog attack at 499 rules and vary the number of packets per second sent through the 500th rule. Using this method we show that we are able to vary the number of evictions caused and achieve throughputs between the 499 and 500 rule clog experiments shown previously. The results, shown in Table 2e in the Appendix, indicate that the number of evictions when the variable rule is added range between the 16412 and 27930 evictions caused by 499 and 500 rules clogged, respectively, in the previous LFU experiments. By doing so, we are able to vary the percentage throughput reduction between 22% and 40% with 1000PPS (Packet_ins Per Second) in the 500th rule giving a 26% reduction in the throughput and 5000PPS giving 35% reduction as shown in Table 3.1.

Packets/Sec	Throughput Mb/s
1000pps	694
2000pps	675
3000pps	654
4000pps	621
5000pps	606

Table 3.1: Bandwidth changes with variable rule

3.5.2 Number of Attackers

We measure the effects the number of attackers have on the power of the attack. For each attack, we vary the number of attackers generating attack flows in the network while holding their attack power constant. In each case, increasing the number of attackers increases the potency of the attack. This begins to look at the possibility of a distributed attack source (e.g botnets) in these attacks which would overcome bandwidth or CPU restrictions of using a single attacker. We capture each attacker's

traffic in turn and combine them as previously discussed with the background traffic and the experimental *iperf* flow.

3.5.2.1 Spray Attack

Multiple attackers performing the spray attack requires minimal co-operation, aside from agreeing on the initiation time of the attack. As the number of malicious users increases, each generates and sends packets at a predetermined rate causing the combined rate of malicious flow rule generation to increase as well as the frequency of legitimate rule evictions. This ultimately causes lower throughputs as the number of malicious users rise.

To demonstrate this, we capture attack traffic from a number of attackers each attempting to flush the 500 space Flow Table 300 times per second. We combine the attack traffic with the *iperf* flow and background traffic and perform the experiments as described previously in this section. The results are displayed Tables 3a-3d in the Appendix and the throughput changes are shown in Figure 3.8. The experiments show that under both FIFO and Random flow eviction policies, as the number of attackers increases, the frequency of the experimental flow's eviction increases almost proportionately, decreasing the throughput. Under the FIFO eviction policy, while a single attacker flushing the flow table at a rate of 300FPS achieved a 23% reduction in throughput, at the same rate 5 attackers, were able to reduce the throughput by 77%. Similarly, under the Random eviction policy, 5 attackers reduced the throughput by 76%. Attackers working in unison therefore are able to reduce the throughput experienced more efficiently than a single attacker and overcome the bandwidth restrictions a single attacker may have (which we saw previously over 500 flushes per second). We note also that despite the increase in attack force, the Spray attack is still unable to affect the volume based flow eviction policies. Neither eviction policy registered any reduction in its throughput.

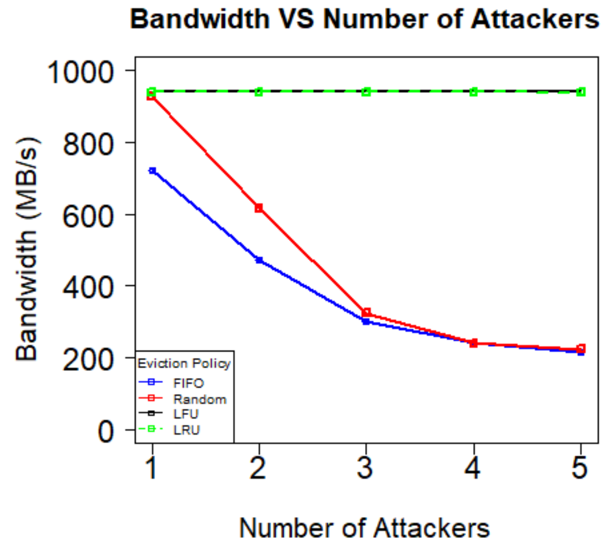


Figure 3.8: Bandwidth Changes with increasing number of Spray Attackers

3.5.2.2 Clog Attack

We capture attack traffic from several sets of attackers performing the clog attack. In each case, the attackers co-ordinate to attempt to clog N-1 rules in the Flow Table (where N is the number of rules the Flow Table can hold). Multiple attackers performing the clog attack requires more co-ordination than the Spray attack. On its own, a single attacker, due to bandwidth and CPU restrictions, is unable to generate enough packets per second through 500 rules to make their rules more valuable than the legitimate rule. Instead, the attackers co-ordinate so that each attacker individually generates traffic to attempt to hold a subset of the rules of the overall attack. This reduces the number of rules a single attacker must attempt to hold and increases the frequency of the packet hits the rules receive. For example, with 5 attackers attempting to clog 499 rules, 4 of the attackers generate flows for 100 flow rules each and one attacker for 99 flow rules.

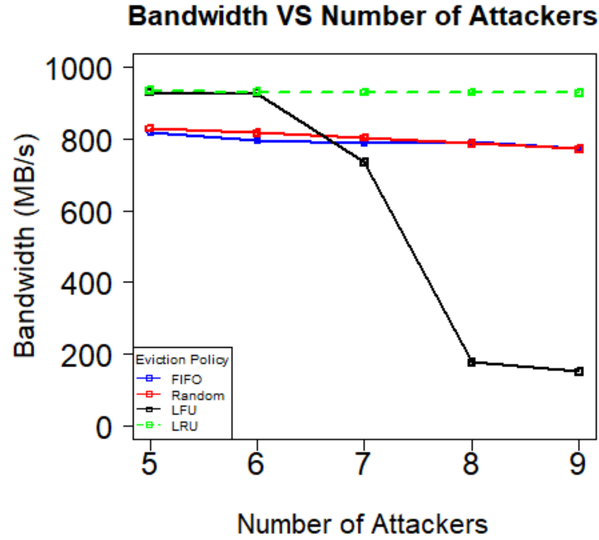


Figure 3.9: Bandwidth Changes with increasing number of Clog Attackers

The results of the experiments are shown in Table 4a-4d in the Appendix and in Figure 3.9. The addition of attackers appears to produce only marginal effects on the FIFO eviction policy. We see a steady decrease in throughput as the number of attackers rise, however the difference between 5 and 9 attackers is only 45Mb/s a 5% difference. Similarly, with the Random eviction policy, we see a steady decrease again however it is a relatively small decrease of 57Mb/s between 5 and 9 attackers- 6%. We see greater variations in the LFU eviction policy. The experiments show that with fewer 7 attackers there is very little effect on the victim throughput as the attackers are unable to generate packets fast enough to hold the Flow Table over so many rules. However, at 7 attackers there is a 22% decrease which increases to an 84% decrease in throughput at 9 attackers. We see a steady increase in the number of evictions of the experimental flow under the LRU eviction policy as the attacker rules receive more hits in a shorter space of time, however 9 attackers is still not enough to cause the number of evictions required significantly reduce the throughput under the LRU policy (reductions of approximately 1%).

3.5.3 Flow Table Size

We measure the effects of the attack on various Flow Table sizes. The size of the Flow Table is an important parameter of the attack. The attacker must have sufficient

computational and bandwidth resources to influence the state of the Flow Table enough to cause frequent evictions of other flows. The attacker must understand the limitations of a single host and determine whether more machines need to be compromised to aid in the attack. We have previously determined that for a table size of 500, at least 7 attackers are needed to make a significant impact on the LFU policy using the Clog attack. Similarly, an attacker needs to scale up or down his attack resources to match the size of the Flow Table. We now show here that it is possible to attack both larger and smaller Flow Table sizes by scaling up or down the resources as necessary to get similar outputs as the values we have previously shown.

3.5.3.1 Spray Attack

The experiments in the Attack Power section showed that for a flow table of 500 flow spaces, a single attacker's potency is limited past 300FPS. At this rate, we therefore recommend one attacker per 500 flow rule spaces. For these experiments,we use one attacker for Flow Tables of size 100, 250 and 500 spaces and we add another attacker for the Flow Tables of size 750 and 1000. The bandwidth changes are documented in Figure 3.10 and the full results in Tables 5a-5d in the Appendix. In each case we hold the attack rate constant at 300FPS for each attacker.

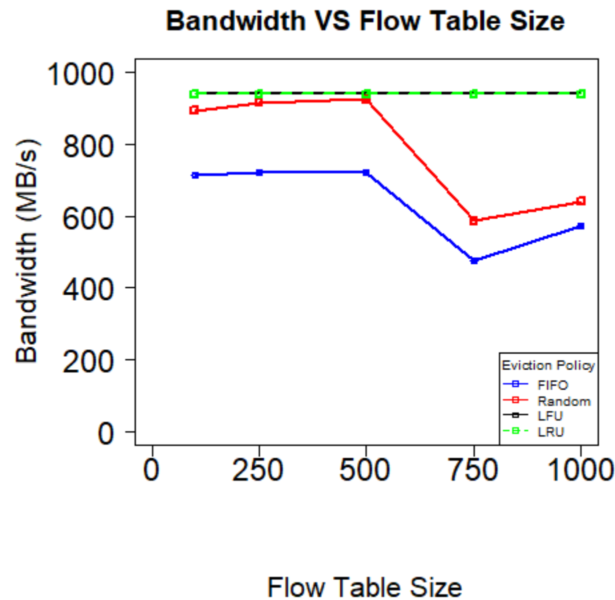


Figure 3.10: Bandwidth Changes with varying Flow Table size: Spray Attack

The results of the experiments show that under both the FIFO and Random eviction policies, the attack is able to reduce the throughput of the experimental flow having adjusted the attack force. For a single attacker flushing at 300FPS, the throughput is reduced by approximately 24% under FIFO in table sizes up to 500 flow spaces. The addition of a second attacker increases the throughput reduction to 49% with the 750 flow space table, however the effect of the attack is reduced as the size of the table grows. The same two attackers reduce the throughput by 39% with the 1000 flow space table. Still a significant reduction, however each attacker is now attempting to generate 1000 packets, 300 times per second rather than 750 packets. Thus the effect is reduced as the attackers struggle to maintain this output. Using the Random flow eviction, a single attacker has a very small and diminishing effect on the throughput as the pool of rules which can randomly be selected for eviction increases, decreasing the chances of the experimental rule's eviction. At 100 flow spaces, there is a 5% reduction. At 500 flow spaces, this decreases to a 1.5% reduction in the throughput. Increasing the number of attackers as we move to larger flow table sizes again sees an increased effect on the number of evictions and throughput, however similar to the FIFO, the effect of increasing the number of attackers begins to decline as the flow table size increases. We continue to see, however that the Spray attack has no effect on the volume based flow eviction poli-

cies (LRU and LFU) regardless of the table sizes used in the experiments.

3.5.3.2 Clog Attack

For the clog attack, at each flow table size, we aim to send streams of traffic through N-1 flow spaces where N is the size of the Flow Table. For each table we attack, we keep the attacker-table size ratio constant: 1 attacker per 75 flow spaces. Thus, for the Flow Table of size 100 spaces, we use 2 attacking machines, for 250 spaces, we use 4 attackers, etc. In this way, we scale up the attack resources to match the Flow Table size, ensuring that regardless of the size of the table, the experimental flow and other traffic are still competing for the same limited resource.

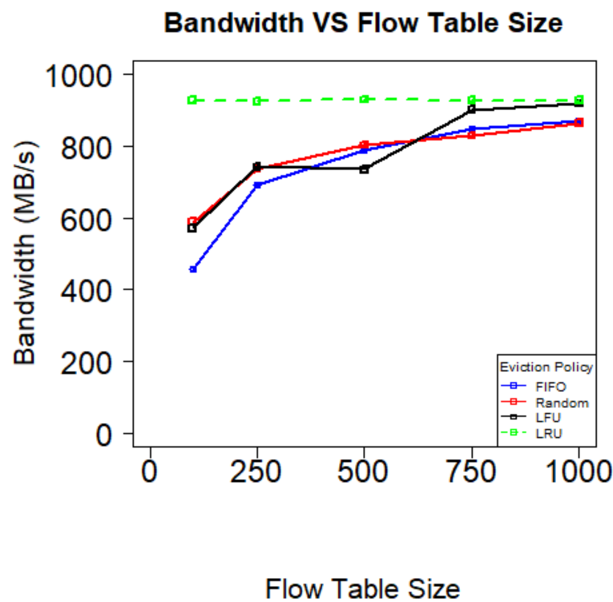


Figure 3.11: Bandwidth Changes with varying Flow Table size: Clog Attack

In spite of this, we see that when the ratio of attackers to size is held fixed, the effects of the attack decline across all the policies as the size of the flow table increases. The bandwidth changes are displayed in Figure 3.11 and the full results of these experiments are documented in 6a-6d in the Appendix. Under the LFU policy, the smallest table (100 flow spaces) achieves the lowest throughput at 572 Mb/s- a 39% throughput reduction. As the table size grows, the reduction in

throughput becomes smaller until the largest flow table (1000 spaces), at which the attack causes just a 2% reduction. More rules increase the likelihood of having malicious rules whose frequency slip below the threshold of the experimental flow packet rate causing those to get evicted rather than the experimental flow rule. For LRU, the likelihood of the attacker hitting all malicious rules between experimental flow packets decreases as the number of rules increases. The experiments show that the number of evictions decrease as the table size increases, however the number of evictions are still too small to cause notable decreases in the throughput (between 1% and 1.6% reduction across all table sizes) which continues the trend of LRU being the most resilient to both attacks. Using the Random eviction, the increase in rule space decreases the probability that the experimental rule will be the one removed, as evidenced by the results which show that the number of evictions decrease relative to the flow table size. Consequently, the reduction in throughput caused by the attacker becomes smaller, moving from 37% reduction at 100 flow spaces to just 8% at 1000 flow spaces. Under the FIFO policy, the time taken for the experimental rule to become the oldest rule increases, similarly reducing the number times it is evicted and reducing the effects of the attack (51% reduction at 100 flow spaces to down to 7% reduction at 1000 flow spaces).

In seeing the trend of reduced effects of the attack as the flow table size increases, the attacker may then decide to increase the attacker-table size ratio. This would reduce the likelihood of falling below the experimental flow's frequency (LFU), increase the likelihood of hitting all the attack flows between experimental flow packets (LRU) and increase the rate of evictions making the legitimate rule the oldest faster (FIFO)and increasing its chance of removal (Random).

3.5.4 Background Traffic

Finally, we evaluate the influence of the surrounding traffic on the effectiveness of the attack. The traffic of a network is one of its most variable attributes and can differ greatly between different periods of a day, week or month depending on the services provided by the network. We look at the effects of the potential variation in traffic (particularly the number of flows being created) in the network on the attack to determine how it affects the number of evictions flows experience.

As explained before, the Flow Table is a limited resource and even without the presence of a malicious user performing the attack, the benign users compete for the resource. It stands to reason therefore that the more benign users competing for this resource (i.e the more benign users attempting to get flow rules into the Flow Table to move traffic through the network), the more evictions will occur. While a resource at a fixed size may be equipped to provide good service to a certain number of users, service may degrade as the number of users increases. Thus, the background traffic during the attack can actually increase the effectiveness of the attack as more users cause more traffic and more evictions. We show this here by varying the background traffic seen during the attack, holding the table size fixed and increasing the number of users/IP addresses in the traffic trace.

Our constant background traffic through the previous experiments has been a subset of a Caida dataset which includes 50 unique source-destination pairings which generate 743 flows with 199997 individual packets within a 60 second period. We vary this fixed dataset, substituting it for several other subsets of the Caida dataset which vary the number of unique source-destination pairings and so vary the number of flows in them. In this way, we are able to model a network of varying sizes, from a small network of 40 hosts to a larger network of 750 hosts. Table 3.2 lists the relevant properties of each dataset.

Name	Number of Unique IP addrs	Flows	Packets
Caida_40	40	331	154690
Caida_50	*50*	*743*	*199997*
Caida_100	100	1730	493050
Caida_200	200	3948	1234544
Caida_500	500	12472	2597887
Caida_750	750	17061	3272669

Table 3.2: Background Traffic Datasets

3.5.4.1 Spray attack

We keep the spray attack at our constant rate of 300FPS and use a fixed Flow Table size of 500 spaces while varying the background traffic in the datasets. We document the results of these experiments in Tables 7a-7d in the Appendix and the

throughput changes in Figure 3.12

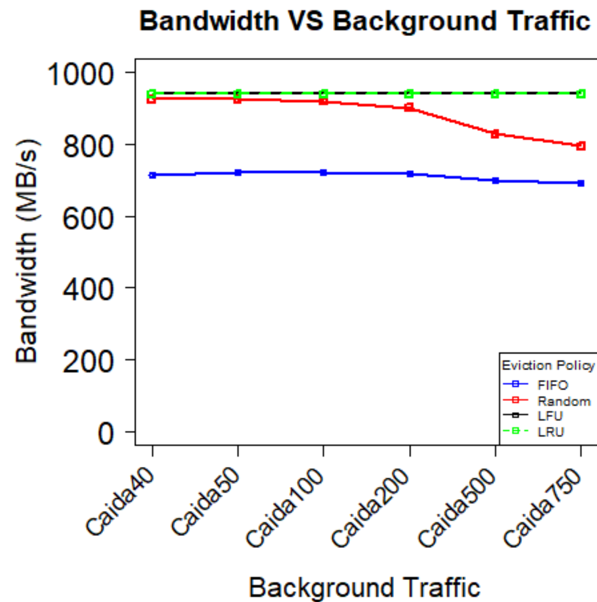


Figure 3.12: Bandwidth Changes under varying Background Traffic: Spray Attack

The experiments show that as the amount of traffic and flows in the network increases, the number of of evictions of the experimental flow increase for both the FIFO and Random eviction policies. Increase in the number of flows in the network intensifies the competition for the flow table resource, aiding the attack in causing a higher number of rule evictions. While the base dataset (50 hosts) caused 17670 flow removals, the largest network (750 hosts) increased the number of flow removals by 2630 reducing the throughput by a further 3% in the FIFO eviction policy. The Random flow eviction policy records a difference of 8384 flow evictions between the smallest and largest network, with the largest network registering a 14% throughput reduction from the smallest. We also note that it continues to have no effect on the volume based policies. This is likely because most of the flows in the trace are short lived and do not carry large amounts of traffic which create enough competition to remove the experimental flow under these policies.

3.5.4.2 Clog Attack

We perform the clog attack with 7 attackers sending traffic through 499 rules of a table of size of 500. With the clog attack, additional traffic in the network can drastically increase the competition for flow rule space within the switch. It significantly increases the rate at which new rules are added to the Flow Table and in doing so, increases the rate at which they are evicted. The more flow rule evictions that need to occur while the attack is attempting to hold the majority of the flow spaces, the more likely the experimental flow is to be selected as the flow for eviction and removed.

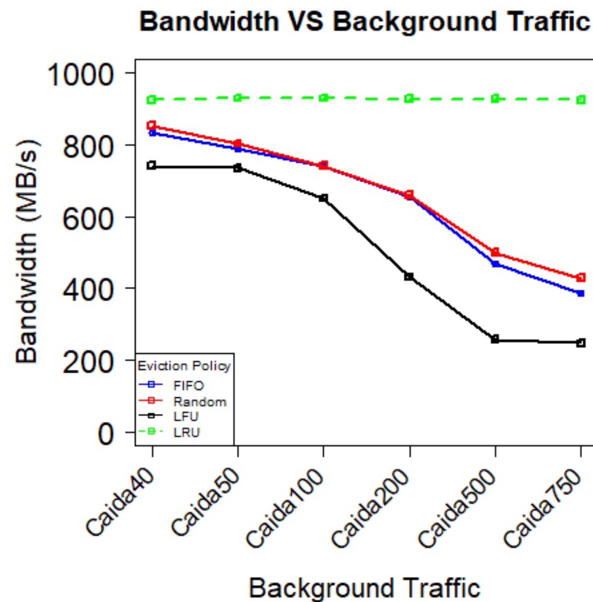


Figure 3.13: Bandwidth Changes under varying Background Traffic: Clog Attack

The results of the experiments (Tables 8a-8d in the Appendix and Figure 3.13) show that the number of evictions of the experimental flow increases as the amount of traffic in the network increases. Under the FIFO policy, the number of evictions ranges from 7931 evictions in the smallest network to 41755 in the largest for the duration of the 60 second flow. Consequently, the throughput reductions increase with the evictions from an 11% reduction to a 59% reduction in the largest network. Similarly with the Random eviction policy- as the number of evictions increase due to the competition for rule space, the throughput reductions increase from 9% to a 54% reduction . The LFU policy, at which the attack is aimed, registers the largest

throughput reduction (73%) in its largest network. This is consistent with the trend that the clog attack is most effective against the LFU policy and is made even more effective by the increase in traffic. LRU policy registers marginal increases in its number of evictions from 190 evictions in the smallest network to 325 evictions in the largest. It produces marginally different throughputs among the varying network sizes which is also consistent with this policy being the most resilient against attacks on long lasting flows.

3.6 Implications

The implications of such an attack are wide reaching. An attacker with the ability to induce arbitrary delays into another user's flows can pose a very powerful threat. Here, we look at a number of scenarios in which an attacker may be motivated to attempt to reduce the bandwidth of another user and its potential consequences.

3.6.1 QoS of Streaming Applications

In 2009, Dell began a focus on flexibility in their workplace which promoted telecommuting instead of on-site attendance. This movement, heavily reliant on the benefits of VoIP (Voice over Internet Protocol) technology, paid exceptional dividends both financially and in employee satisfaction and productivity, saving an estimated $39.5M through telecommuting [153]. In 2012, the corporate consumer market for VoIP was valued at roughly US$43 Billion and forecasted to continue growing [154]. Such technology is enabled and supported by its underlying network infrastructure and is only as good as the service provided to it by the network. VoIP technology requires end to end delays to be capped in the region of 100 milliseconds [2] and provide <1% packet loss rates to retain reasonable quality of service.

Similarly, Video Streaming carries its own requirements. End to end delays should not exceed 250ms and the packet loss should be <0.5% [2][155]. Beyond these, the quality of service quickly degrade to becoming unusable. According to Cisco, 73 percent of internet traffic in 2016 was comprised of video streaming [156] and a recent Deloitte survey [157] indicated that US households collectively spend on average 2.1 billion per month on video streaming services. As one of the largest

video streaming companies, Netflix's revenue in 2017 exceeded US$11 Billion [158]. These statistics show that video streaming is a huge industry and to maintain its profitability and competitiveness, the companies must provide good service to its customers.

Attacks demonstrated in this chapter present a real and credible threat to both types of streaming activities. As the academic community continues to promote SDN as the future of networks due to its ease of control, issues like this promise to have devastating effects if not properly thought through before deployment. In both of the examples described in this section, the profits derived were supported heavily by the quality of service provided to the services in use by their underlying network infrastructure (servers, switches and other network devices). Compromise of such network infrastructure via the network hosts is a well studied topic within the field of Computer Science. Viruses and Trojans, weak passwords and un-updated Operating Systems and applications all present avenues for a malicious entity to infiltrate the network. Once the attacker has taken control of one or more hosts on the network, various side channel attacks such as those mentioned in Section 2.7.2 give the attacker the required information to carry out these attacks.

An attacker who can therefore carry out such attacks an SDN network hosting Netflix's video content can influence the latency and ordering of the packets within the stream's flow and degrade the user experience or render the service unusable for any local users connecting to that server. This can result in out of order packets being dropped which translates to frozen pictures and out of sync audio for the user. Similarly, Dell's reliance on VoIP technology can be undermined if an infected host begins to launch these attacks on the underlying infrastructure, disrupting important business interactions. All of this culminates in massive financial and reputational damages as we see entire industries building their businesses on the foundation of faultily implemented network protocols.

3.6.2 Critical Infrastructure

Cyber Physical Systems (CPS) have demonstrated an incredible ability to bring savings in manpower and energy which translate to financial savings by outfitting

systems with automated electronic sensors which monitor and react to situations without the need for human intervention. CPS have been shown to help save energy in Data Centers by monitoring temperature and adjusting the supplied energy accordingly [159].Similarly, in heating systems, monitoring weather conditions brought an 85% energy savings equivalent to 197 tonnes of Carbon emissions in a 2015 experiment into CPS deployed in a football stadium [160]. In more critical systems, CPS have been integrated into healthcare along with cloud computing to automate a wide range of processes. Monitoring and alerts for changes in patient's vital signs, analysis of symptoms, predictive diagnostics, sharing patient information between hospitals and assistance for remote surgical procedures are all procedures into which CPS have become heavily involved in the healthcare industry [161]. Such systems can reduce hospital expenses and relax the necessity for routine checking of patients by the hospital staff [162] which may already be stretched thin. All of these contribute to a more efficient healthcare establishment and translate into lives saved and quality of care increased.

Cyber Physical systems implemented in smart cities also carry critical consequences if mismanaged. As the concept of smart cities in which automated systems are extensively implemented to convenience and efficiency, CPS can play an essential part in the day to day functionality of the city. [163] particularly focuses on CPS implemented in sewage systems to control the issue of flooding, pointing to climate change and urbanisation as factors causing flooding increase. It proposes the use of interconnected devices to control and regulate sewage levels in real time. Such systems become part of the critical infrastructure of the city upon which the population depends to maintain their livelihood.

Wireless Sensor/Actuator Networks (WSANs), of which many CPS systems such as the ones that would be employed in smart cities require routing protocols more advanced than current widely deployed MAC and transport protocols. Self management technologies which can control the service level given to each type of traffic is required for such systems [164]. The centralised and fine-grained management of SDN makes it an excellent candidate for the underlying network infrastructure of Cyber Physical Systems. The ability to control the flow of information across a wide geographical area from a single location promises exactly the type of convenience prescribed for WSANs [165].

CPS systems built on SDN networks due to its perceived management benefits inherit a huge liability if the vulnerabilities discussed here remain exposed. Commands and actions in many of these systems are extremely time sensitive and in the most critical cases, reduced service and packet loss may have a devastating effect on the institutions relying on them [164]. An attacker with the ability to arbitrarily delay commands, updates and responses by way of the attacks demonstrated in this chapter can cause great amounts of destruction. The computerised sewage system monitoring levels of flooding which suddenly has its throughput throttled can cause a city to flood when its reports and commands are not handled in a timely manner. In another case, alerts at a hospital to notify staff about critical changes in their patient's conditions are suddenly delayed drastically may become the cause of many casualties. It is vastly important that in addressing one issue within the SDN architecture, we carefully evaluate the implemented solution to ensure it does not bring comparable harm in another way.

3.7 Summary

Switch Based Flow Rule Eviction, introduced in newer versions of the OpenFlow protocol, empowers the switch to remove flows without the intervention of the controller. In doing so, the developers aim to improve network performance and increase the security of the network, particularly against issues such as the Table Overflow Attack. This represents an increase in the switch intelligence in the SDN paradigm as it enables the switch to autonomously perform tasks without dependence on the controller. While this concept presents a range of benefits (Section 3.1.2), our closer analysis in Section 3.2 indicated that its First In First Out implementation created a new vector of attack on the SDN network which allowed a malicious host in the network to control the bandwidth their neighbours experienced on the network. We demonstrated the potential for such attacks on the eviction mechanism in this chapter and proposed alternative eviction policies which may be better suited to a network. Through simulation of attacks on the various flow rule eviction policies, we determined that evicting the Least Recently Used flow rule is the most attack resilient of the evaluated policies, best protecting benign elephant flows in the network. We conclude in this chapter that in increasing the switch intelligence, an LRU policy for switch based flow rule eviction should be implemented rather than the

current FIFO. The LRU policy both prevented Table Overflow issues and closed the attack vector on switch based flow rule eviction opened by FIFO.

Having noted the benefits to the network performance of these initial steps towards smarter switches, we extend this further to the concept of an intelligent SDN switch. We propose a design for such a switch in the next chapter which includes the Switch Based Flow Rule Eviction concept as well as several other functionalities which allow the switch to aid the performance and protection of the network autonomously. We aim to demonstrate, through implementation and rigourous evaluation of some of these concepts, the merit of switch intelligence in the SDN network.

Chapter 4

Increasing SDN Switch Intelligence

4.1 Introduction

The typical SDN paradigm concentrates the intelligence of the network at the centralized controller and designates the switch as a dumb forwarding device. Switch based flow rule eviction marks a step in the opposite direction by restoring some modicum of intelligence to the SDN switch (we define "switch intelligence" as the ability of the switch to perform tasks autonomously without dependence on the controller). We saw in the previous chapter that this intelligence can present many benefits, one of which is an effective defense against Table Overflow attacks (provided the right eviction policy is used). Having seen the resilience intelligence in the form of switch-based flow rule eviction can add to the SDN networks, **we propose that a more intelligent switch can work in conjunction with the control plane to aid the security and performance of the network.** We present in this chapter high level designs detailing the functionalities of an intelligent switch as well as the implementations of one of these functionalities. In Section 4.2, we outline the functionalities of a DoS-resilient intelligent SDN switch. We expand on one of these functionalities (Intelligent Flow Request Handling) in Section 4.3 and 4.4, providing implementation specifics for modules which realise this design attribute.

4.1.1 Benefits of Switch Intelligence

Increasing the intelligence of the SDN Switch can be a strategically beneficial move for the Security and Networking community for several reasons:

- The Switch is the first point of contact with the network for a flow. Therefore, in attempting to secure the network against many attacks, it makes sense to attempt to stop the attacks in their tracks at the point of entry.
- Positioning defenses at the switch can help reduce the load on the controller. The centralized controller is already seen as a potential bottleneck in the network. Configuring the switch as a first line of defense can ease the security responsibility of the controller, taking some of the load off the control plane.
- As the "brain" of the network, the controller is the most sensitive component of the network. Control plane failure can cripple the network. Therefore, the first line of defense should come well before the controller. If the first and only line of defense is the controller itself, then the network is far less secure than it should be.

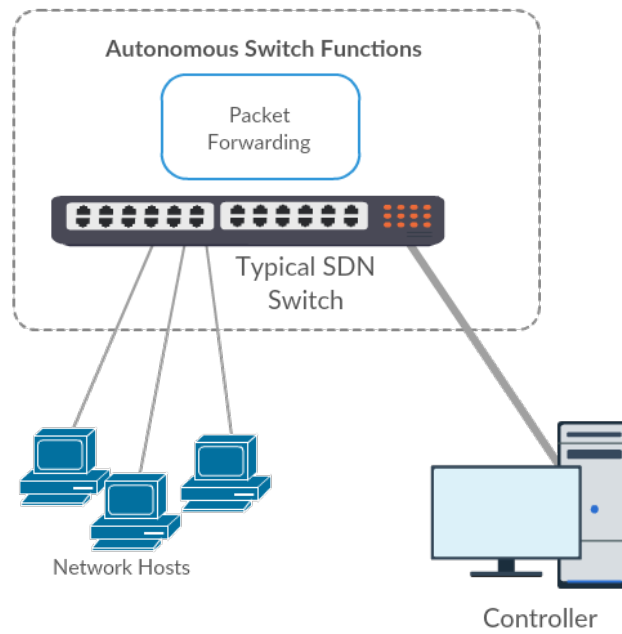
We have previously also seen evidence for the merit of switch intelligence in systems such as AVANT-GUARD [166] and LineSwitch [118]. By proxying TCP connections at the switch, they were able to filter out malicious TCP connections without the need for controller intervention, thereby reducing the load on the controller. While innovative, an expressed goal of AVANT-GUARD was to "add limited intelligence" to the switch. We propose to lift these limitations. Our goal is to augment the intelligence of the switch such that while the Control plane is still the central intelligence of the network, the switch is able to add resilience to the network by autonomously handling some tasks.

4.2 Intelligent Switch Design

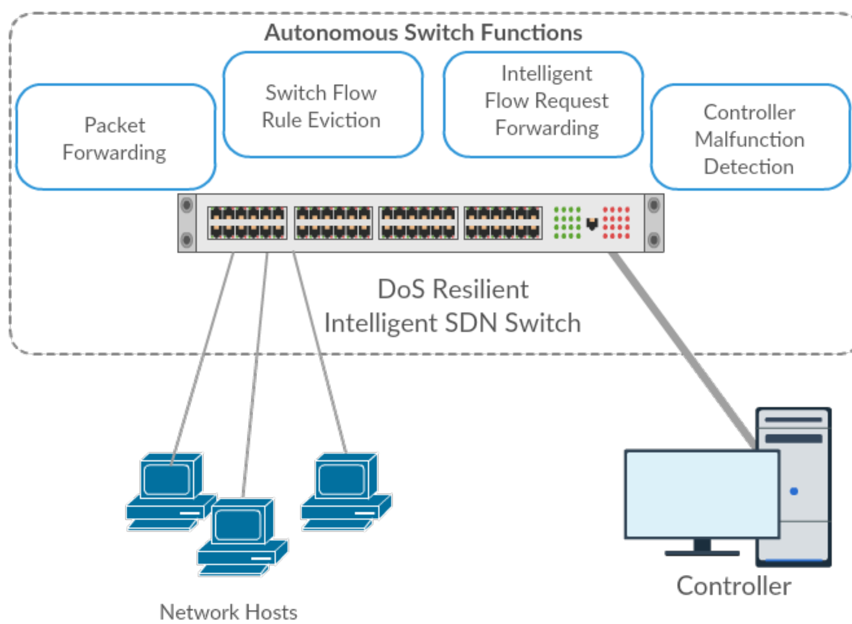
The primary goal of increasing switch intelligence is to enable it to autonomously contribute to the performance and security of the network. Augmentations proposed should therefore comprise of functions which enable and maintain such a focus without compromising the main function of the switch: forwarding data plane packets.

With that in mind, we outline a high level design for switch intelligence.

While there are a myriad of network issues an intelligent switch could potentially address, we restrict our focus to the issue of DoS (both incidental and intentional) affecting the SDN network infrastructure. DoS is one of the most prevalent issues affecting networks and SDN networks, as outlined in Chapter 2, require new innovative solutions to mitigate the attacks they face. As we show in Chapter 5, implementation of these designs in the switch can help alleviate the threat of DoS in SDN networks and improve the performance of the network under DoS conditions. The concept diagram in Fig 4.1 outlines the intelligence amplifications proposed for the SDN switch. The functions are further expanded upon in the following subsections.



(a) Functions of a Typical SDN Switch



(b) Functions of an Intelligent SDN Switch

Figure 4.1: Intelligent SDN Switch Design

4.2.1 Switch Flow Rule Eviction

We begin by including the capabilities discussed in the previous chapter. SDN flow table restrictions is a well noted issue of SDN which presents the opportunity for Denial of Service. A malicious user may explicitly attempt to fill the flow table or it may be the result of a misconfiguration which forces a large amount of traffic through a single switch creating more rules than the switch can manage. In both cases, the result is that new traffic is unable to traverse the network and finds itself denied the expected service of the network.

An intelligent switch should therefore have the ability to remove flow rules without needing control plane intervention. Section 3.1.2 discusses the benefits of such capabilities in relieving the controller of some of its load and reducing the delay between the command and the action, reducing the number of packets which may be dropped. While Least Recently Used is the best of the 4 algorithms evaluated in Chapter 3, there may be alternative algorithms better suited to the network. Research done in the field of CPU cache-replacement has produced a wide range of algorithms to replace CPU cache content similar to replacing flow table rules. Thus, the implementation specifics of this feature requires further research to determine which algorithm is best suited however, its merits are clear.

4.2.2 Intelligent Flow Request Forwarding

Flow requests are one of the main differentiating factors between SDN and traditional networks. Due to its separated control planes, a switch that receives a packet for which it has no instruction (flow rule) forwards this packet to the controller in the form of a "packet_in" or flow request. While this mechanism enables great flexibility in the SDN network, it can also create issues. Large volumes of packet_ins can be generated under both benign/legitimate circumstances (e.g Flash Floods) and by malicious activities (e.g attackers performing Control Plane saturation attacks). In both cases, a flood of packet_ins can exhaust both the controller resources and the bandwidth between the switch and controller. This has been shown to negatively affect the performance of the network, providing reduced bandwidth and higher packet loss rates [99][89], both of which reduce the quality of service provided by the network.

SDN switches in their native form, blindly send all flow requests to the controller. An intelligent SDN switch should perform more intelligent flow request handling. Instead of simply forwarding all requests to the controller, it should be more aware of how its large volumes affect such a critical point of the network (the controller) and act accordingly. In explicit attempts to attack the control plane by flooding it with requests, the switch should differentiate between legitimate and malicious requests. In non-malicious circumstances, in which the high rate of requests is caused by legitimate requests (possibly through a misconfiguration or flash flood), the switch should be conscious of its effect on the control plane and take steps to relieve the burden. One form of this "awareness" which has been widely implemented is rate-limiting which allows switches to drop requests if the controller is unable to handle the volume however, this in itself still causes service to be denied to some portion of the traffic.

4.2.3 Controller Malfunction Detection

The controller is the most critical piece of the SDN infrastructure. A malfunction on the part of the controller can cause a black hole in the network, denying service on a greater level than an attack or misconfiguration on a single switch (because the controller oversees multiple switches). Kreutz et al [88] also discusses the issue of compromised controller stations and the magnitude of the threat it presents to the entire network. A rogue controller can reprogram several switches at once to the detriment of the network.

Just as controllers monitor the switches connected to them, switches should perform more active monitoring of the controllers connected to it. Switches should have the capacity to detect a malfunctioning or rogue controller which is issuing no or incorrect routes. A controller that does not respond to flow requests may be easily detected in a multi-controller architecture in which controllers propagate reports of their activity among the controller pool. Nevertheless, depending on the frequency of report propagation among the controllers, switches within the offending controller's domain may be better suited to detect non-responsiveness on the part of the controller than other controllers.

Compromised or deliberately malicious controllers present a more challenging

threat. In a multi-controller system, the rogue controller may be able to operate undetected by propagating false reports to appear benign to the other controllers while placing incorrect flow rules into the switch. Implementing controller monitoring capabilities in the switch has the ability to prevent the effects of a compromised controller which intentionally aims to damage the network in this way and acts as another layer of "checks and balances" within the SDN network.

4.2.4 Implementation

We aim to illustrate the effectiveness of a smarter switch in this thesis by focusing on the intelligent flow request mechanisms proposed for the SDN switch described above. We aim to resolve the issues of high control plane load enabling the network to provide good service to legitimate traffic even under such circumstances (high control plane load) by implementing remediation techniques in the switch itself. Rather than blindly forwarding all flow requests into the control plane, we enable the switch to intelligently handle the requests it receives before sending them on. Under attack, we reduce the amount of attack traffic that arrives at the control plane by enabling the switch to identify and filter (drop) flow requests deemed as malicious. In the event of legitimate high control plane load, we enable the switch to perform fine-grained distribution of the flow requests among the controllers available to it. Both these techniques reduce the response time for flow requests from the control plane for legitimate traffic resulting in better service from the network. We design and implement in this chapter two mechanisms to achieve these goals. Our designs here do not represent a complete solution for an intelligent switch but aim to demonstrate the value of an increase in switch intelligence by focusing on the flow request mechanisms of a DoS resilient intelligent switch.

4.3 Load Distribution

It has been widely acknowledged that a single controller SDN architecture imposes scalability and reliability issues on the network [126][127]. Given that it can only support a limited number of flow setups, a single controller providing instructions to many switches can quickly cause the controller to be overwhelmed by the amount of work required of it. Once a controller becomes overloaded in this way, it is no longer

able to process all the messages coming from the switch and for those it is able to process, the response time increases drastically. This results in large amounts of packets being dropped at the switch as the buffers overflow and degrades the network performance.

As such, many works have looked at distributing the control plane [132][130][167]. Once the control plane has been distributed however, the question is raised of how to make the best use of the additional processing power available to the network. If a multi-controller architecture is implemented yet a minority of controllers are overloaded while the rest are idle the problem has not been solved. Thus, load distribution within a distributed control plane is a necessity and has the potential to increase the network performance and relieve the pressures of irregular traffic distribution.

4.3.1 Taxonomy

Previous work in the area of controller load distribution are all implemented within the control plane and can be divided into two sections: centralized and distributed decision making.

Centralized Decision Making: Centralized controller load balancing systems use a hierarchical approach which involves a decision-making controller selected out of the pool of controllers. In each case, this lead controller is tasked with balancing the flow request load among the other controllers as best as possible to ensure the imbalances are kept as small as possible. Each switch in [168] is connected to one controller. The head controller here, monitors each controller's load and the traffic in the network. It then dynamically maps and remaps as necessary the switch-controller pairings to ensure one controller is not receiving an unreasonably high number of flow requests while another is idle. Rather than remapping switches and controllers, Balanceflow [169] handles its load balancing at the flow level providing a far more fine-grained approach. Its head controller here monitors the number of flow requests received by each controller and rectifies imbalances by placing rules in each switch with instructions for distributing its flow requests to various controllers. The flow requests here are differentiated by IP address with the rules indicating to which controller flow requests involving a certain IP address should be sent. By load

balancing via distribution of flow requests rather than distribution of switches, the methodology here allows for more precise load balancing.

While a centralized intelligence within a distributed control plane allows for efficient co-ordination among the controllers, it may be less practical in the face of a sudden onset high volume attack. An inherent problem is that such systems are only as good as the head controller's ability to react. If the head controller polls the other controllers repeatedly, there is a tradeoff between reaction time and the number of messages in the system. If it polls often, it may be able to react quickly but the number of messages between the controllers increases drastically as the rate of polling increases. If the polling is infrequent, the head controller may not react quickly enough and network performance is reduced.

Distributed Decision Making: Several other systems perform load balancing without the use of a centralized authority [170][171][172][134]. The preferred method of load balancing in each is to dynamically find the best controller switch pairing in the network to ensure all the controllers manage similar loads. [172] uses a one-to-many matching game to calculate optimal pairings, considering a minimum utilization of processing capacity that each controller must achieve and the maximum possible processing capacity of each controller as factors in the game. Switches can dynamically vary the controller they attempt to connect to but controllers have the final say on whether they will allow a switch to connect based on the switch's load and their current capacity. This system does not include inter-controller communication as each controller operates independently of the others. This raises questions about the controllers each keeping a consistent view of the network which the others solve by including a communication system. Both [170] and [171] use JGroups to send messages between controllers. [170] collects load information about other controllers in the network when under high loads and hands off switches to under-loaded controllers to relieve itself and rebalance the network load. By contrast, the controllers in [171] proactively and periodically inform each other about their loads to allow for a faster handover when overloaded rather than polling when they exceed a given threshold. Instead of a one-to-many mapping, [134] uses a many-to-many mapping in which each switch has several controllers connected to it in a MASTER-SLAVE configuration. In the event of a switch's master controller becoming overloaded, the MASTER controller initiates a process to swap with one

of its SLAVE controllers.

With the exception of BalanceFlow, the solutions aim to balance the load on the controller by remapping the switches and controllers. While this may work under benign traffic surges, the problem with this under an intentional high-volume attack aimed at overloading the controller is that it may simply transfer the attack from one controller to another without actually relieving the problem. This methodology either causes the new controller in charge of the switch in question to be overloaded or in a worse scenario, causes a high amount of churn in the network as the switch under attack is passed from controller to controller, causing each new controller to be attacked in turn. Another problem with controller based solutions is that high volume attacks cause congestion in the controller-switch channel making passing instructions to switches very difficult. For example, [134] takes 12 messages to complete its controller exchange. None of these are sure to be delivered if the attack is clogging the channel. Depending on the controller to issue instructions to the switch while under attack itself may as well prove to be problematic. In either case, the problem with the solution is that it may never arrive where it is needed.

With these issues in mind, we propose to place the load balancing solution in the switch. Deploying defenses at the switch level provides an opportunity to tackle saturation attacks earlier and provide better defense. Our load balancer is inserted directly into the switch and distributes the flow requests among the available controllers. Similar to BalanceFlow, our load balancing is performed at the level of the flow request, rather than at the switch level. It does not add further control messages to the network to poll the statuses of controllers, nor does it require a centralized decision maker as each switch is capable of distributing its loads independently.

4.3.2 OpenFlow Support

Native OpenFlow (v1.2 onward [46]) provides support for multiple controllers. Several controllers can be connected to the switch in one of three configurations.

1. MASTER- This often works in conjunction with the SLAVE configuration. A controller configured to be the MASTER controller of a switch has both read and write access to the switch and is responsible for inserting flow rules and

other commands. The switch allows at most, one MASTER controller at a time.

2. SLAVE- This works in conjunction with the MASTER configuration. This controller configuration only has read access to the switch, cannot issue commands and does not receive most switch notifications.A SLAVE controller can request a role change to MASTER, at which time in which case the previous MASTER controller reverts to a SLAVE role and can no longer issue commands.

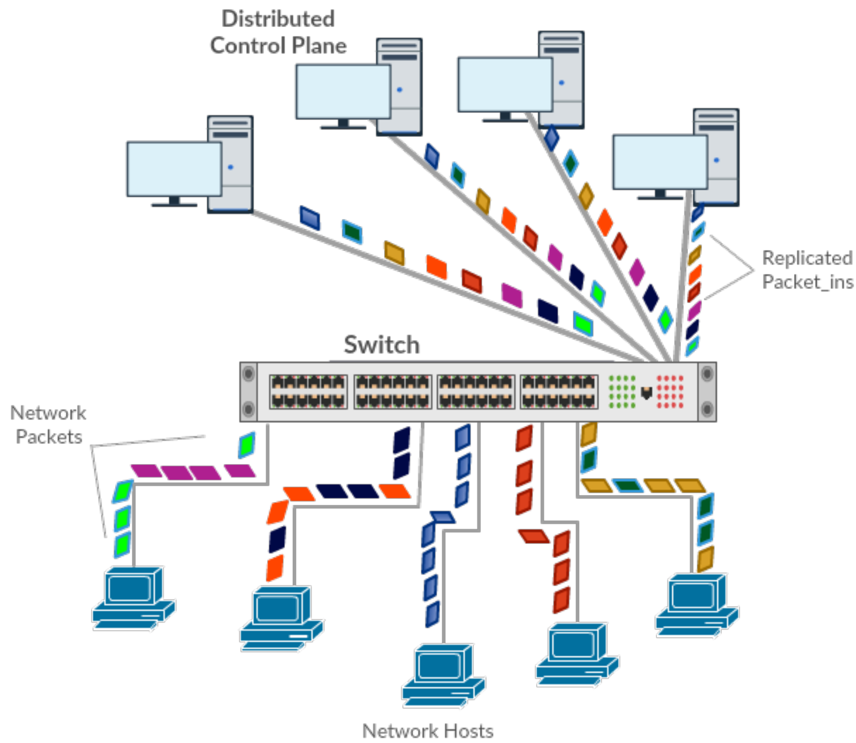
3. EQUAL- Controllers configured to EQUAL roles can all write to the switch (issuing commands) and all receive notifications from the switch (such as flow requests or port updates). If not carefully managed, this controller configuration can lead to massive amounts of redundancy and inefficiencies as each connected controller attempts to perform the same jobs (e.g responding to flow requests) on the switch.

Our Load balancer design takes multiple controllers connected to the switch in EQUAL roles, enabling them all to read and write to the switch. Our system deals solely with the distribution of flow requests and leaves the inter-controller communication up to the controller implementation. We assume that all the controllers have a consistent view of the network, in particular the states of the flow tables and relevant network policies which must be enforced. Each controller can read from the switch, and so can request updates at any time. We trust the controllers to make each other aware of the updates they make to the switch via whatever implemented communication system they have.

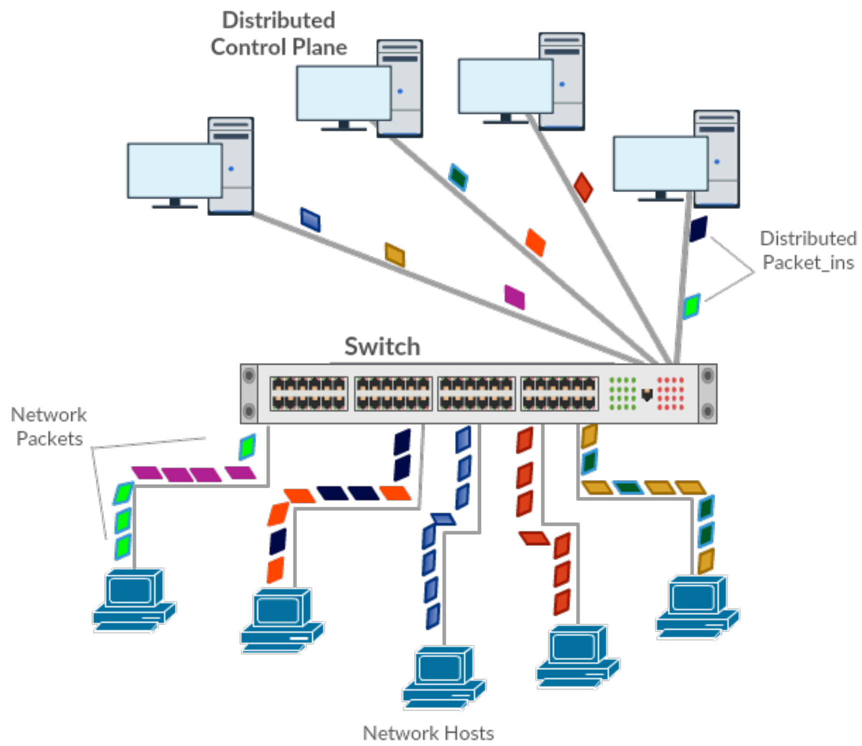
Under normal circumstances, with the native OpenFlow implementation, the switch, when connected to multiple EQUAL controllers sends all flow requests to all controllers. Without an efficient inter-controller communication system (i.e the controllers operate independently and unaware of each other), each controller responds with a flow rule addition command for the request it was sent. This results in multiple instances of the same flow rule being added for a single flow request. Our Load balancer systems instead allow the switch to select which controller it will send a flow request to out of the pool of controllers it is connected to.

4.3.3 Switch Based Load Distribution Design

We propose two designs for the load distribution in our switch: Round Robin and Load Aware Load Balancing. Both designs enable the switch to determine which controller each packet_in is sent to in a distributed control plane. Figure 4.2 illustrates the concept. As traffic flows through the network, the flow requests generated are distributed to specific controllers determined by the switch rather than forwarding all flow requests to all controllers as the native OpenFlow mandates (replicating each flow request across all connected controllers).



(a) Native OpenFlow Flow Request Replication



(b) Flow Request Distribution

Figure 4.2: Switch Based Flow Request Distribution

Round Robin: In this system, each switch connects to multiple controllers in EQUAL mode and each controller is assigned a unique ID. When the first packet of a new flow arrives at the switch, instead of sending the flow requests to all controllers, the switch selects one controller to send the request to. Each of the connected controllers is selected in turn in a round robin fashion, ensuring that each controller receives 1/n flow requests, where n is the number of controllers the switch is connected to. Each controller in this system maintains its own view of the network and as stated before, we rely on the controller communication system implemented in the distributed control plane to ensure they have a consistent view of the network and relevant policies via updates.

We implement the algorithm outlined in Algorithm 1 in Open vSwitch. It assigns each new flow request to a different controller, incrementing the controller ID (to select a new controller at the next request) after each flow request.

Algorithm 1: Round Robin Load Balancer Algorithm for Packet_in Distribution

```
currentControllerID = 1;
foreach flowrequest (f) do
    foreach Controller (c) do
        if c.ID == currentControllerID then
            c.send(f);
        else
            continue;
    currentControllerID++;
    if currentControllerID > TotalControllers then
        currentControllerID = 1;
    else
        continue;
```

A Round Robin algorithm system is ideal for ensuring the load on the controllers are perfectly balanced since it ensures that no controller has more than one additional flow request to deal with than any other controller in the pool at any point in time. A drawback of this approach is that it is simplistic and assumes a perfect network. The network design here assumes homogeneity among the controller pool.

The assumption is that all the controllers are the same, their underlying hardware have the same CPU and memory resources and they are all equidistant from the switch in question. Therefore, each controller takes exactly x time to respond to any flow request. These assumptions would mean that the controllers are truly equal and rotating the flow requests in a round robin manner keeps them all in sync with each other with regards to their load. In reality, if any of those factors are different, it influences the rate at which the controller responds to flow requests. If the controller responses fall out of sync with each other, eventually some controllers become overloaded while others remain well within their capacity. Thus, this system of blindly rotating the controllers the requests are sent to may not work as well as intended in for all architectures. Nevertheless, as we show later, in multicontroller systems which have equal resources, this solution works well to ensure controllers are not overwhelmed.

Load Aware Load Balancer: We therefore propose a second design that may be better suited to the practical drawbacks of an imperfect network. Assuming the controller, the underlying CPU resources and the latency may not all be perfectly homogeneous, we propose a Load Balancer which selects a controller based on its perceived current load.

In similar fashion to the Round Robin system, each switch is connected to multiple controllers in EQUAL mode. In this design, the Load Balancer module of the switch keeps track of both the number of unanswered flow requests for each controller and the length of the queue of requests to be sent to the controller.

When the first packet of a flow arrives at the switch, the Load Balancing module reviews the number of outstanding flow requests each controller has and selects the one with the least number of unanswered requests. It then checks the status of that controller's queue. If a particular controller's queue is full, the switch drops the next request to the controller. Therefore, if the selected controller's queue is full, the module selects the next lightest controller (controller with fewest outstanding requests) which has space in its queue. To reduce the likelihood of over-dependence on a single controller, if several controllers have the same (least) number of outstanding requests, it selects randomly one of the controllers from this set. This aims to avoid a single controller being selected more often simply because it is first in

the queue. The module then updates the number of outstanding requests associated with that controller, adding one to the value. When the switch receives a FLOW ADD command from a controller (which adds a flow rule to the switch table in response to a flow request), it decrements the number of outstanding flow requests associated with the controller before installing the rule. In this way, the switch attempts to infer and monitor the load of the controller without adding extra communication with the controller. If the controller has a high number of unanswered requests, the switch assumes that its load is higher than one with few unanswered requests. This may be for any number of reasons including higher latency or slower processing capacity. By considering the queue status as well, the algorithm gives the flow request the best chance of being delivered and responded to in a timely manner.

Algorithm 2 outlines the Load Aware load distribution process. Part (a) selects the controller based on the least number of outstanding requests and the queue size and increments the number of outstanding requests. Part (b) decrements the number of outstanding requests upon receiving a FLOW ADD command from a controller.

Algorithm 2: Load Aware Load Balancer Algorithm for Packet_in Distribution

```
(a)
foreach FlowRequest(f) do
    least_requests = Integer.MAX;
    Primary_controller_choice_set = null;
    Secondary_controller_choice_set = null;
    foreach Controller (c) do
        if (c.out_reqs <least_requests) && (c.queue != full) then
            Primary_controller_choice_set.clear();
            least_requests = c.out_reqs;
            Primary_controller_choice_set.add(c);
        else if (c.out_reqs == least_requests) && (c.queue != full) then
            Primary_controller_choice_set.add(c);
        else if  (c.out_reqs <least_requests) then
            Secondary_controller_choice_set.clear();
            Secondary_controller_choice_set.add(c);
        else if  (c.out_reqs == least_requests) then
            Secondary_controller_choice_set.add(c);
    if (Primary_controller_choice_set != null) then
        controller = Primary_controller_choice_set.random();
        controller.send(f);
        if (f.sent == true) then
            controller.out_reqs++;
        else
            continue;
    else
        controller = Secondary_controller_choice_set.random();
        controller.send(f);
        if f.sent == true then
            controller.out_reqs++;
        else
            continue;
```

```
(b)
if c.receive(flow_mod) then
    c.out_reqs–;
else
    continue;
```

4.3.4 Summary

We propose here two designs for load distribution by the SDN switch which enables the switch to partition the control plane load it generates among the available controllers. By introducing distribution mechanisms in the switch, we increase its intelligence with the aim of avoiding the issues identified in previous control plane load balancing systems while making the best use of the control plane resources available. The evaluation of these modules in the next chapter aims to show that enabling the switch to autonomously distribute its load among multiple controllers allows for better performance in the network.

4.4 Malicious Packet_in Classification and Filtering

The Load Balancer module focuses solely on distribution of traffic. It does not aim to stop attacks on the network controller but rather treats all flow requests as equal and forwards each flow request received to the controller indiscriminately, assuming it to be a part of legitimate traffic in the network. We address here the issue of intentional controller saturation by considering the presence of a malware infected/malicious user attempting to attack the network. To reduce the potency of control plane saturation attacks and the switch based flow rule eviction attacks discussed previously, the network should be able to distinguish between legitimate and malicious flow requests- classing malicious flow requests as ones which are simply issued for the purpose of disrupting regular use of the network. Ideally it should be able to distinguish between malicious and legitimate requests from a single host such that in the event of the host becoming infected with malware, the network is still able to provide appropriate service to the other users of the network while

defending itself against the effects of the malware.

In its current state, the very nature of SDN enables these DoS attacks on the network. The passive capabilities of SDN switches inherently create the opportunity for these attacks as they are mandated to forward unmatched packets to the controller. To reduce the vector for attack, we propose to amplify the switch's intelligence such that it no longer passively forwards every unmatched packet to the controller, but rather only legitimate requests. We propose a policy that mandates that the switch should automatically detect and discard packets that generate malicious flow requests. Many solutions are implemented at the controller level, with the SDN paradigm having moved much of the intelligence away from the switch. We propose placing this solution at the switch level, thus moving some of the intelligence back to that layer. We argue that this stops attacks at the source (before it can enter the network) and reduces the load on the controller, thereby protecting the switch, the control channel and the controller from this attack. Figure 4.3 illustrates this concept. The packet_ins generated by the malicious network packets specifically crafted to DoS the control plane are dropped before they can be sent to the controller while legitimate packet_ins are forwarded as normal to the controller.

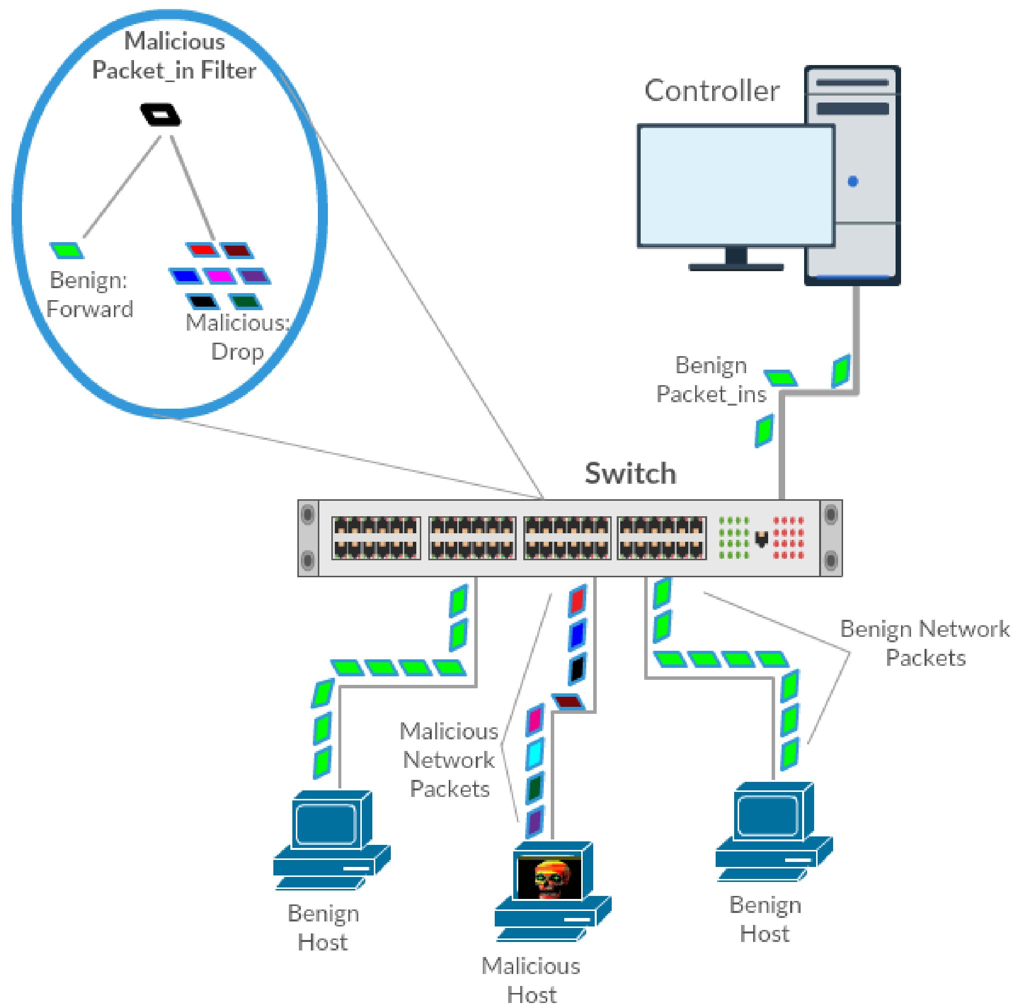


Figure 4.3: Malicious Packet_in Filter

4.4.1 Adversarial Model

We propose a similar attacker model as in the attacks in Chapter 3. We assume the attacking host is a regular user of the network which does not possess any special administrative privileges in the network. The attacker may have compromised a small minority of the hosts on the network (for example via malware infection) and will use these to perform attacks on the network. The attacker does not have direct access to the switches or controller(s) he/she is attempting to attack nor do they have root access to any other hosts apart from the minority under their control. The

attacker's aim is to reduce the quality of service experienced by the other network users by DoS-ing the control plane, clogging the switch-controller communication channel and attempting to fill the switch Flow Table by creating as many flow requests/flow rules as possible.

Unlike other works, we assume the attacker has infiltrated the network and is now internal to the network. This is a slightly more complex attacker model to defend against. Other works which sought to defend against attackers external to the network have noted that an external attacker attempting to attack a network is restricted in the destination IP addresses he can use in order to ensure his traffic enters the targeted network. By this restriction, researchers are able monitor for and identify attacks based on the IP addresses being used (e.g [104]). An internal attacker attempting to DoS the controller or switch is not subject to this restriction since his traffic is already, by nature, part of the network. With full control over his own packet creation process, the attacker can spoof and randomize the IP addresses (and any other packet header fields) as necessary to subvert detection mechanisms.

4.4.2 Taxonomy

4.4.2.1 Machine Learning in SDN Security

Machine Learning has been implemented in various forms across a number of solutions proposed to mitigate network attacks on SDN networks. We briefly discuss some of these solutions to demonstrate the value of machine learning to SDN in preventing network compromise.

Support Vector Machine (SVM) algorithms have been implemented in several solutions to protect SDN networks. [106] proposes use of the SVM machine learning algorithm to detect various forms of malicious behavior including DDoS within an SDN network. The system here uses traffic attributes such as Source and Destination IP address, Source and Destination Port and Protocol, fed into the SVM classifier to detect the attacks. Limited evaluation is performed on this solution making it difficult to assess whether it is viable as part of a real time DoS prevention system, however, as discussed in the following section, their use of easily spoofable attributes for detection may prove a significant shortcoming in deployment. [173]

also uses SVM classification to categorize network threats in an Intrusion Detection System for SDN. The solution uses the ID3 decision tree theory to select the best attributes for the SVM classification which iterates over the dataset, selecting which attribute gives the greatest information gain at each iteration. The solution also uses the sFlow sampling tool set to sample data from suspicious connections and the sampled data is fed into the classifier to detect anomalous traffic flows. Among the attacks the classifier is used to detect is Host-based DDoS attacks. While this aims to detect a host targeted and not the infrastructure targeting attacks of this thesis, it continues to evidence the usefulness of Machine Learning for security in SDN environments. Upon receiving packet_ins, FloodDefender [99] uses SVM machine learning to classify a flow as benign or malicious. It considers attributes such as packet count, byte count and reverse flow statistics to determine whether a flow is benign or not. Their proposed solution comes in the form of a low level controller app that sits between the controller and all other apps on the controller and monitors both the packet_ins to the controller and the rules in the flow table to reduce memory consumption in the Switch Flow Table. [107] uses both SVM and SOM (Self Organising Maps) classification to determine malicious flows within the network, performing a quick SVM classification on the flows and a further SOM classification on those flows which SVM produces no clear result. In doing so, it aims to prevent against DDoS attacks and to reduce the workload of SDN controllers. We critique both [99] and [107] in the following section and focus on their merits in merging Machine Learning into SDN here.

Moving past SVM classification, [174] implements 4 different Machine Learning algorithms and analyses their ability to detect DDoS attacks in SDN networks. The algorithms examined are Naive Bayes, K-Nearest neighbour, K-means and K-medoids. The data fed into these algorithms consists of the time at which the connection is requested, the source and destination hosts and the packet flag bit. These data values are used to train the algorithms and the algorithms are implemented in RYU. The evaluations are performed in mininet and indicate that while the Naive Bayes algorithm significantly outperformed the others in Detection Rate, it also required the greatest amount of processing time to achieve its results. K-means had the smallest processing time but also performed the worst in detecting the attack. [175] focuses on the feature selection portion of the ML defences by analyzing the traffic collected in the NSL dataset which contains traces of several

network attacks and allowing WEKA, an open source machine learning software, to select the key features from the traffic dataset. It then inputs these selected features into the Sequential Minimal Optimization (SMO) algorithm for training and stores these attack patterns in a local database to be used to detect the DDoS attacks. The researchers then tested the system using the SMO, J48 and Naive Bayes machine learning classification algorithms to determine which feature sets and algorithms provide the best detection accuracy, finding that the SMO and J48 gave the highest detection with the featureset selected by WEKA in comparison with any other tested subset of features. [108] proposes a solution for DDoS attacks using SDN which includes a neural network that classifies each flow as benign or malicious based on the flow stats such as packet and byte rate. Once the neural network detects the DDoS attack, the system takes steps to trace the attack to its source and then mitigate by blocking the source and clearing the flow table of the detected attack traffic. In evaluation, the solution showed the ability to accurately detect the attack and free the flow table space occupied by the attack traffic, allowing legitimate traffic to use the resources.

4.4.2.2 DoS Prevention in SDN

In Chapter II (Background), we looked at several DoS prevention systems proposed for SDN. We revisit some of them here with a more critical perspective. In particular, we look at the systems that explicitly attempt to prevent control plane saturation attacks and the ways in which our system may be able to improve results specifically against the attacker model described above. We highlight several issues with current proposals for mitigation of DoS in SDN networks, citing examples as we discuss them.

Evaluations using SDN Specific Attacks: One criticism that some SDN DoS mitigation systems face arises from the lack of SDN specific DoS attacks in evaluation. They specify the aim of mitigating controller saturation attacks but use datasets containing traditional DoS attacks. These traditional DoS attacks target a specific host, server or link in the network. By comparison, SDN specific attacks are geared towards the network infrastructure, saturating the control plane or filling the switch flow table using a specially constructed stream of packets with diverse headers. A traditional DoS attack may or may not include such packets as they are not intrinsically characteristic of traditional DoS attacks and evaluating SDN DoS

defenses using traditional DoS attack data may produce misleading results. However, there are few publicly available SDN attack datasets upon which work may be evaluated. As such, if researchers use instead traditional DoS attack datasets, their solutions may evaluate reasonably well on the traditional DoS attack however, it may only protect against SDN specific DoS attacks which also look like traditional DoS attacks.

[106] uses a DARPA intrusion detection dataset in its DoS Classification proposal. It uses features such as IP addresses, protocols and TCP/UDP ports to detect the attack, training and testing on the Darpa dataset. The attacks in this dataset are not targeted to SDN networks and so may not accurately represent the key characteristics of a DoS attack on the core of an SDN network. This means their solution may not perform as well as hoped in deployment. They also do not evaluate their solution's performance on a live attack to ensure it can actually detect the attack in the field.

The system proposed in [119] purports to consider the issue of DoS attacks against the controller, however their system is unlikely to work under SDN specific DoS attacks whose aim is to overload the controller since attack flows characteristically do not have further requests coming after them to be buffered. They do not evaluate their system in the paper, however conceptually, the premise their system is based on (that buffering subsequent table misses of the same flow will protect the controller) may only work in the event of benign traffic flows or traditional DoS attacks whose flows carry large amounts of packets.

SDN-Score's system [124] monitors the packets in the network and is only triggered if the bandwidth usage of a host exceeds a particular threshold. This is a fundamental component of a traditional DoS attack which consumes bandwidth resources, but may not hold true for an SDN focused controller saturation attack. Even a high volume control plane attack involving a large number of controller-triggering packets may not necessarily mean a high bandwidth. Empty packets (since the unique packet headers are the main ingredient of a controller saturation attack) can be created as small as 60 Bytes. Assuming an attack rate of 5000 flow requests per second, the bandwidth consumed by the attacker registers a paltry 300 KB/s. Since the average web page size is around 2MB [176], this bandwidth usage is

equivalent to 1 web page request every 4 seconds and therefore is unlikely to trigger their system.

Spoofable Features: Another problem across a number of solutions is the use of spoofable features for detection and filtering. The attacker in our threat model has full control over the packet creation process in the attack, therefore the solutions presented to mitigate the attacks must carefully choose the features they look for to ensure the system is not easily subverted.

Within the packet protocol space, AVANT-GUARD [166] and LineSwitch [118] explicitly offer protection solely against TCP flows. Both acknowledge the obvious shortcoming that if an attacker can create UDP flows, he/she will subvert the filtering system and indeed our own attacks demonstrated in Chapter 2 make use of the UDP protocol. Similarly, Floodguard [111] partitions requests into queues based on four protocols when under attack (TCP, UDP, ICMP and Default). The assumption here is that the attacker will only use one protocol in his attack and so flow requests of other protocols will retain reasonable service from the controller. An intelligent adversary, aware of the implemented security measure, can assign any protocol to the attack packets created and so fill all queues, causing legitimate requests to be dropped or receive slow service. While this does not overload the controller, it accomplishes the same result of severely reduced Quality of Service in the network because the attacker has full control of the packet-creation process.

The FlowRanger system [112] uses IP addresses to identify users and assigns each user a trust rating. Since the IP addresses which only appear during attack times gain a low trust rating, the attacker can spoof the IP addresses on his packets, using other uninfected hosts' IP addresses to subvert the attack detection. Also within the IP address space, two entropy-based systems provide solutions based on threat-models which assume external attackers (such as those using botnets) attempting to DoS the controller of a particular SDN network. [121] looks for packets which cause dramatic reductions in the entropy of the requests' Source IP address, Destination IP address and Protocol. It finds the minimum set of packets causing the reduced entropy and filters those requests out as malicious. Similarly, but more specifically, [104] monitors the entropy of the incoming flow requests' Destination IP addresses, rationalizing that an external attacker is limited in the range of destination IP ad-

dresses they can use to target a specific network. Use of a few destination addresses for a large number of flow requests causes the entropy of destination IP addresses to decrease, allowing the attack to be detected. While these systems may work well to detect external attackers, it cannot be applied to internal hosts whose machines have become infected and attempt to launch an attack. The internal hosts, being already in the network, are not bound by the restrictions of IP addresses or protocols to ensure their traffic enters the network. Their packet headers can therefore be randomized, confusing the detection system into allowing attack traffic to get through to the controller. However, these systems can be deployed in conjunction with our own as our work specifically target internal attackers and will not function as well if set to monitor ingress ports which by nature receive a large amount of diverse packet headers.

Controller based solutions to protect the controller: Finally, placing solutions which aim to protect the controller in the controller is inherently problematic. Simply having to process a high number of OpenFlow packets (without additional route mapping functions) can prove too much for the controller host and even low-level controller apps that attempt to filter malicious packet_ins run the risk of being overwhelmed by the number of packet_ins they receive under attack. Another effect of the controller saturation attack is the congestion of the bandwidth between the controller and the switch. Controller solutions in which the controller attempts to issue commands through a congested channel may not be as successful in a high-volume attack.

FloodDefender [99] is a low-level app placed in the controller which filters malicious flow requests using a threshold-based frequency filter as well as SVM classification on the rules in the switch flow table. The system is tested on packet rates of up to 500 packet_ins per second (PPS). This performs reasonably well in their evaluation, however their results indicate that the performance of their solution degrades as the intensity of attack increases. The maximum attack rate used for evaluation of the system is 500 flow requests per second, however an attacker is capable of creating attacks significantly more intense than this. In our experiments we generate attack rates of up to 10000pps (from a single host) in the face of which it is unclear how well performs.

Both FlowRanger [112] and the proposal in [43] employ a system in which requests are divided into queues in the controller. The queues are then serviced in a round robin fashion, ensuring that an attack coming from a particular host does not impede the others from receiving service. Unfortunately, the switch, ignorant of the queues in the controller, may register the backlog in the number of requests sent to the controller and begin to drop flow requests (both malicious and legitimate) in its controller queue. Additionally, by virtue of being in the controller, these solutions still absorb precious computational resources of the controller and offer no protection for the controller-switch channel bandwidth which in many cases is a precious resource.

While controller-based solutions provide some benefit, the security of SDN networks can be improved by implementing solutions which stop malicious flow requests *before* they reach this critical element of the network. We show in our work that by increasing the intelligence of the switch, our switch-based system significantly enhances the performance of the network under attack when compared with other solutions.

In an attempt to improve the DoS resilience of the SDN network around these issues, we extract from SDN focused DoS attacks what we believe are intrinsic features of the attack and use a classifier in the switch to filter out packets aimed at controller saturation before they arrive at the controller. We create a malicious packet_in filter using Random Forest Classification [177], a high accuracy, widely used classification method (e.g [178][179][180]) to distinguish legitimate from malicious flow requests before forwarding them to the controller.

4.4.3 Random Forest Classification

Decision trees: Random Forests make use of Decision trees for their Classification Process. Decision Trees are structures often used in Machine Learning to represent decisions based on conditions. Each internal node of a tree holds a condition against which the tree evaluates the features of a dataset. The leaf nodes of the tree represent the conclusions the tree can arrive at based on the decisions made by the internal nodes. Thus, decision trees are used to classify the points of a dataset by evaluating the features of each point against the conditions in the tree.

Random Forests: Random Forests is a simple, fast and flexible supervised machine learning algorithm which uses a collection of decision trees to classify data. The classification process takes place in two stages: the Learning stage and the Classification stage.

Learning stage: The Random Forest first trains through "bagging". With Bagging, the system takes random samples from the dataset to train on. Ideally around 66% of the total dataset is sampled for training. The system then chooses a different subset of features from the dataset for each sample to create the conditions for the internal nodes of each tree. Out of its feature set, the feature which provides the "purest" division of the data (purest meaning the least mixing of data on each side of the split) is selected at each internal node of the tree. The final output of the learning stage should be a diverse set of decision trees which use varying feature sets to determine the classification of data.

Classification Stage: Having amassed a "forest" of trees, the Random Forest classifier is then able to determine a classification for each datapoint of a dataset by allowing each decision tree in the forest to classify the datapoint according to its training and output a decision. The forest then selects the most popular classification over all the trees to give a final prediction about the datapoint.

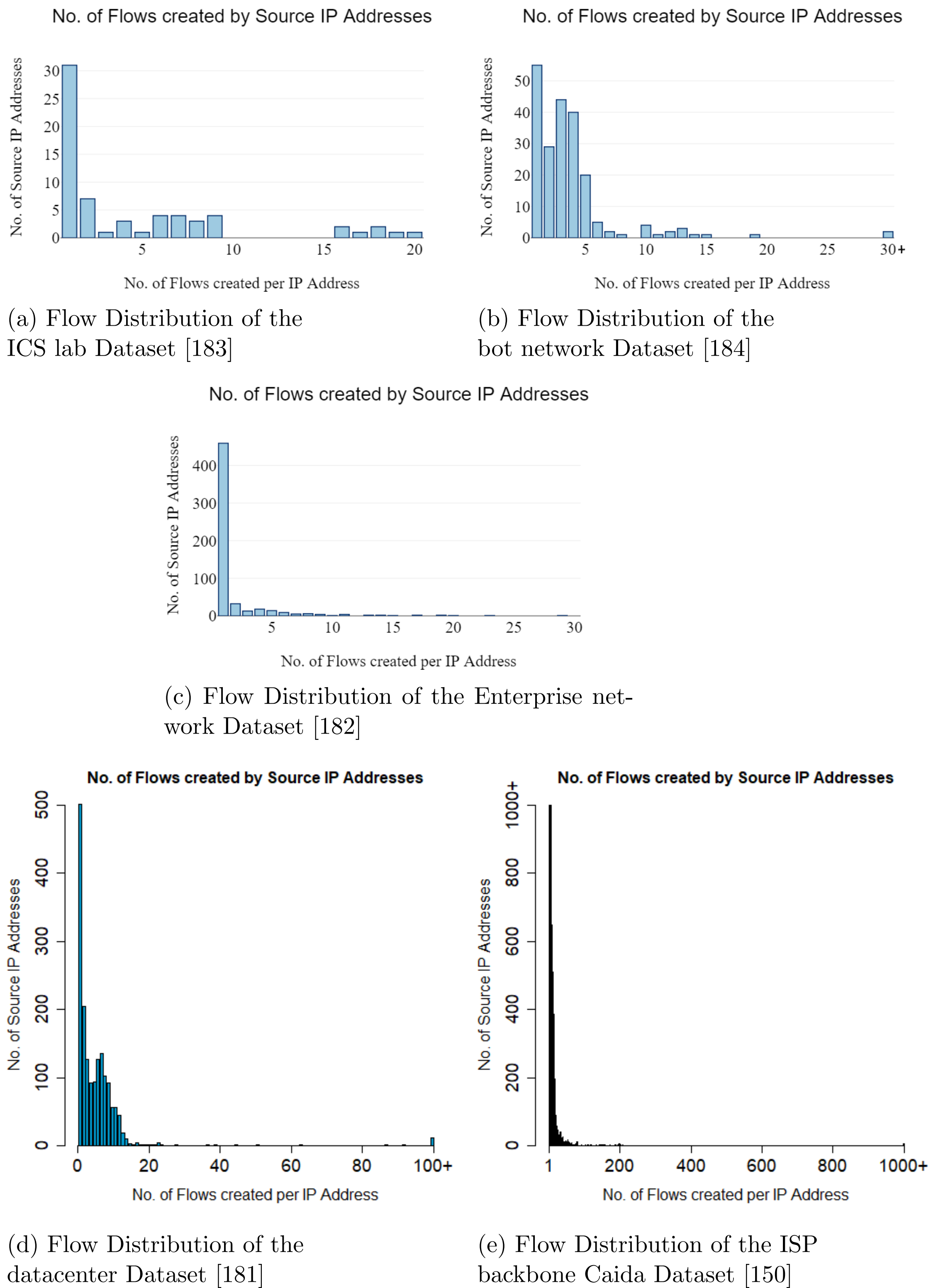
4.4.4 Attribute Gathering

A key factor in any Machine Learning application is the features or attributes used to classify data. As highlighted previously in the Taxonomy section, it is important to use features intrinsic to the attack which cannot be altered by the attacker but are not present in benign traffic. We begin therefore by analyzing the behavior of legitimate traffic using 5 publicly available traffic traces. These traces (shown in Table 4.1) represent networks of varying sizes, include a data center [181], an enterprise network [182], an ICS (Internet Connection Sharing) lab network [183], a Caida trace [150] captured from an ISP backbone router and a small experimental network of bots which randomly surfed websites and read emails and files [184]. Analysis of the flows existing in each trace indicate that in any one dataset, 98.8% of IP addresses initiate fewer than 25 flows per minute. We illustrate this in figures 4.4a-4.4e.

Dataset	Time Frame (seconds)	No. of Flows	No. of Packets
ICS Lab Network [183]	907	301	2253164
Bot Network [184]	1400	879	4813568
Enterprise Network [182]	600	1144	5697364
Data Center [181]	475	15495	4613099
ISP backbone [150]	50	643057	20089824

Table 4.1: Dataset Attributes

Through analysis of these traces, we show in Figure 4.4a that of the 65 unique IP addresses 58 IPs create fewer than 10 flows. Similarly, Figure 4.4b indicates just 2 of the 211 Source IP addresses in the trace create more than 20 flows. This trend continues among all the datasets analysed, where in each case a very small number of IP addresses are found to create many connections. By contrast, in order to overwhelm the controller and deteriorate the quality of service received by users of the network, attackers must generate a large number of flows using diverse packet headers in a short space of time [43][97][95]- the exact opposite of legitimate user behavior.



(a) Flow Distribution of the ICS lab Dataset [183]

(b) Flow Distribution of the bot network Dataset [184]

(c) Flow Distribution of the Enterprise network Dataset [182]

(d) Flow Distribution of the datacenter Dataset [181]

(e) Flow Distribution of the ISP backbone Caida Dataset [150]

Figure 4.4: Flow Distribution of various datasets

We therefore extract from the attack attributes differentiating malicious and benign behavior using three key points identified from the traffic.

1. **Large Number of Requests:** The controller can easily handle a small number of requests therefore the attacker must send enough requests to overwhelm the controller

2. **Entropy of flow request packet headers:** To maximise the number of attack packets which go to the controller, the attacker randomizes the packet headers to ensure that they do not match the rules in the flow table. The diverse set of packet headers results in high entropy. Since we investigate every flow request, this also helps give benign weighting to packets from the same flow which trigger further flow requests while awaiting the response from the first packet.

3. **Time Period of 25 requests:** The behavior of the traffic analysed shows that >98% of traffic generates more than 25 flows in 60 seconds. Since an attacker is likely to send hundreds of packets per second to the controller, this value for the attacker is likely to be very small (milliseconds), while for a benign user, it is much larger (seconds).

Finally, we consider the ingress switch port of a flow request to be a spoof resistant attribute and use this as the main identifier to group flow requests. Attackers can spoof many attributes of a packet in their attack (e.g IP addresses). However, one attribute the attacker is unable to control is the physical switch port upon which the attack packets arrive. Our legitimate traffic traces do not record the incoming port of the flows therefore, we assume that each unique source IP address originated from a unique host/port (since a legitimate host would have no reason to spoof its source IP addresses) and so we use the source IP addresses to group the flows by port.

4.4.5 Switch Module Design

Having identified the intrinsic characteristics of an attack, we build a "profile" for each port on the switch which monitors and records the behavior of the port using the three (3) attributes described above. For each potential flow request, we extract the necessary attributes as described below.

Number of flow requests: From the analysis of both malicious and legitimate traffic, we found that legitimate sources generate a few tens of flows while malicious sources, in order to overwhelm the controller or Flow Table, must generate a large number of flows (at least several hundred flows in most cases). Having analysed the legitimate traffic on a 60 second scale, we keep track of the number of flows generated by a source per 60 seconds. We classify a "source" as a host connected to a switch port.

Flow Request Window: The best option for tracking the number of flow requests occurring from a given port is using a sliding window. This is challenging for active classification being performed on live traffic in the network as it would involve storing each flow request from each port with a timestamp to track the number of flow requests which occurred 60 seconds or less prior to the current request. In the event of an attack in which thousands of flow requests are generated, this can quickly exhaust both memory and processing capacity. By contrast, using a static window reduces the amount of information that needs to be stored but also reduces accuracy of the recorded number of requests for each port. In a static window collection method, the number of requests seems small at the start of each window, as if they were legitimate.

We instead attempt to merge the static and sliding windows into a hybrid. We use a static window of 60 seconds for the number of flow requests. To avoid the case where the first requests in a window are taken as values independent of the previous requests, we consider the number of requests in the latter half of the previous window for the first half of the current window. With this idea, we only need to record the number of flow requests in the latter half of a window instead of each flow request and its associated port and timestamp. We then add this value to the number of flow requests in the first half of the next window when calculating the number.

Algorithm 3 explains the procedure:

Algorithm 3: Calculation of the Number of Requests in 60 seconds

```
Time_window_start = 0 Time_window_end = 60 Time_window_half =
 (((time_window_end  time_window_start)/2) + time_window_start)
 Prev_half_window_val = 0 Current_window_value = 0
foreach FlowRequest(f) do
    if (f.time >time_window_start && f.time <time_window_end) then
        if ( f.time <time_window_half) then
            Num_of_requests = (Current_window_value++) +
             Prev_half_window_val
        else
            Num_of_requests = Current_window_value++
             Prev_half_window_val ++
    else
        while (windowEnd ≤ time) do
            time_window_end = time_window_start time_window_start =
             time_window_start +60 Current_window_value =
             Prev_half_window_val Prev_half_window_val =0
        Num_of_requests = Current_window_value++
```

Entropy of Flow Requests: The rules placed by our controller match the attributes source and destination IP address therefore we consider these particular attributes of the packet headers. Therefore, the attacker must alter the source and destination IP addresses in his packets to create new flow requests. Both the training and classification data can be configured to consider the entropy of others such as protocol type and MAC address, however.

To monitor the entropy of all the flow requests within a given window (which we choose to be 1 second in our implementation), we would be forced to record all the requests within the window and calculate the entropy. Given the processing time constraints and storage resources constraints within the switch, storing several thousand requests for a port (as is received under attack) is impractical. We therefore draw on the behavioral characteristics of the legitimate traffic. We noted that >98% of legitimate traffic generated fewer than 25 flows per minute. With this in mind, we record the last 25 flow requests of each port, storing only the time it was generated and integer representations of the source and destination IP Addresses.

These values are stored in a linked list with each node representing a single flow request and each switch port being monitored is given its own linked list. For legitimate traffic, we can easily calculate the entropy for the last second using these 25 flow requests (since only one or two of them would fall within that window). If the host in question creates more than 25 flow requests within the time window (as malicious traffic does), we only consider the entropy of these last 25 flow requests (which will show itself to have a noticeably different entropy).

Time taken to generate Flow Requests: As our final attribute, we consider the time taken to generate 25 flow requests. We use the same 25 flow requests stored in the previous attribute and simply calculate the time between the first and last flow request in the 25. From this, malicious flow requests will have a very small-time gap between their first and last requests, whereas legitimate flow requests should show larger gaps. Sources which have not generated 25 flow requests have this value extrapolated based on the number of requests they have generated, and the time taken for these requests.

All of the above attributes are calculated in the switch for each incoming flow request. We shy away from using other factors such as distribution of IP addresses (which is a common attribute selected for classification techniques) since many other potential attributes are easily spoofed by the attacker and can be used to subvert the classification process as previously discussed. We also keep the list of attributes small in an attempt to minimize the number of trees needed. By minimizing the number of trees used to classify a flow request, we reduce the processing time necessary for classification which is essential in the system.

4.4.6 Switch Implementation

Training Data: We generate attack data for training using a single host sending packets with randomized destination headers at a rate of 2000 packets per second. We capture the attack packets at the switch and use this as the malicious dataset for training. For legitimate datasets, we take 3 of the 5 datasets analysed as "legitimate traffic", excluding the two larger datasets [181][150] (as these are too large for the network). The dataset from [182] is used for training and classification testing for accuracy is carried out on [183][184] as explained in the subsequent chapter. These

legitimate datasets contain between 300 and 1200 flows. To extract the packets which cause flow requests in our datasets, we simulate a flow table of 500 spaces in *Java*. Using a hashtable data structure to represent the flow table, the state of the flow table is checked at each packet for a matching rule (matching by source and destination IP addresses) in a similar manner to an SDN switch. If a rule is not found, this packet is recorded as a flow request and a rule is inserted into the hashtable. Having collected the flow requests of the datasets, we extract the attributes discussed above from our malicious and legitimate datasets. This data can then be fed into a Random Forest implementation for training.

For the Random Forest implementation, in the training phase, we use *ranger* [185], an open source C++ implementation. We feed the training dataset to the ranger Random Forest program which outputs a "tree" file which can be used to classify similar data. We extract from *ranger* its code for classifying datapoints and implement the algorithm as a function within Open vSwitch. We create a decision tree structure in Open vSwitch and populate (at network boot time) our Random Forest within the switch using as tree values, the decision trees generated by ranger's training phase, extracted from the output file. Thus, using our in-switch Random Forest and Classification function we classify each incoming flow request generating packet by placing a function call in the "execute_controller_action" function (which prepares a flow request to be sent to the controller). This function call extracts the necessary attributes from the packet_in and executes the classification code to determine whether this packet_in is malicious or benign. If the packet_in is malicious, it is dropped without further processing. If the packet_in is classified as benign, the flow request is created and forwarded to the Load Balancing module for controller selection and sending.

4.4.7 Summary

We present here a flow request filtration system which uses Random Forest classification to distinguish between legitimate and malicious flow requests on a switch port. By placing this solution in the switch, we create a smarter switch and deploy defences for the control plane before the packets can arrive at the controllers. We place emphasis on the feature set used to identify malicious flow requests on a switch port to ensure the attacker is unable to subvert the attack detection mecha-

nism. We implement the filtration system in the switch, enabling it to identify and drop malicious flow requests intended to overwhelm the control plane.

4.5 Conclusion

We describe in this chapter the concept of an intelligent SDN switch and its merits in boosting the performance and security of the network. We present several high-level concepts discussing various mechanisms that should be included in the design of such a switch and how they may improve the network functioning. We subsequently outline the implementation of two modules which demonstrate one of the concepts: intelligent flow request forwarding.

We present both these modules (packet filtration and load distribution) as switch based solutions to the controller overloading problem, both by surges of benign traffic and attack. By implementing intelligence in the switch (enabling it to autonomously perform various tasks), we can stop attacks entering the network core without additionally stressing the controller. This also frees the controller to perform other critical tasks such as route planning instead of focusing on defending the network.

In the following chapter, we evaluate both modules and demonstrate that they solve the issues of controller saturation while avoiding the problems highlighted in the related work. We put both modules through rigorous testing to identify its effects on network characteristics such as switch and controller CPU usage, Latency and Bandwitdth. The strength of our evaluation lends credit to the opposition of mutually exclusive intelligence in the network.

Chapter 5

Evaluation of Additional Switch Intelligence

5.1 Introduction

Building on the premise of increased switch intelligence analysed in Chapter 3, we outlined in Chapter 4 designs for an intelligent SDN switch which contributes to the performance and security of the network in addition to its packet forwarding. Several augmentations to the SDN switch capabilities were proposed and specific designs which realised the implementation of the Intelligent Flow Request Forwarding module were detailed and implemented into Open vSwitch.

In this chapter we aim to evaluate the effects of the increased intelligence in the switch outlined in the previous chapter. We examine the tangible benefits of smarter switches that autonomously perform tasks which help alleviate the problem of control plane overload. With each module implemented, we perform a range of tests on the switch, comparing the results to both the generic, unmodified version of Open vSwitch (which we refer to as the "Stock OvS") switch and other works aiming to perform similar functions where applicable. We use Open vSwitch version 2.7.0 for all modifications and experiments.

In section 5.2 we evaluate the Load Distribution aspect of the Intelligent Flow Request Forwarding, determining each algorithm's effect on the number of flow requests serviced by the control plane (Section 5.2.1), effect on the CPU loads of each

controller (Section 5.2.2) and effect on the latency of packets sent to the controller. In section 5.3, we examine the effect of the malicious packet_in filter on the network. We examine first the detection rates of the Random Forest algorithm (Section 5.3.1) and later evaluate the effects of the filter within the switch on factors such as network connectivity, Bandwidth and resource consumption within the switch and controller (Section 5.3.2).

5.2 Load Distribution

We evaluate here both implementations of the Load Balancer. We aim to investigate the benefits of enabling the switch to distribute its load among the controllers connected to it for better service. We also aim to determine which of the designs proposed extracts the best service from the network.

Using the network setup in Figure 5.1, we vary the number of controllers connected to the switch in each experiment to determine how the size of the controller pool affects the experimental results. Additionally, to illustrate the merit of the Load Aware Load Balancer, we evaluate each algorithm using Homogeneous and Non-Homogeneous controller pools. We use the host labeled “Load Generator” as a traffic generator which allows us to vary the load on the controllers in each round of experiments. In the multi controller architecture depicted, each controller is connected to the switch in “Equal” status.

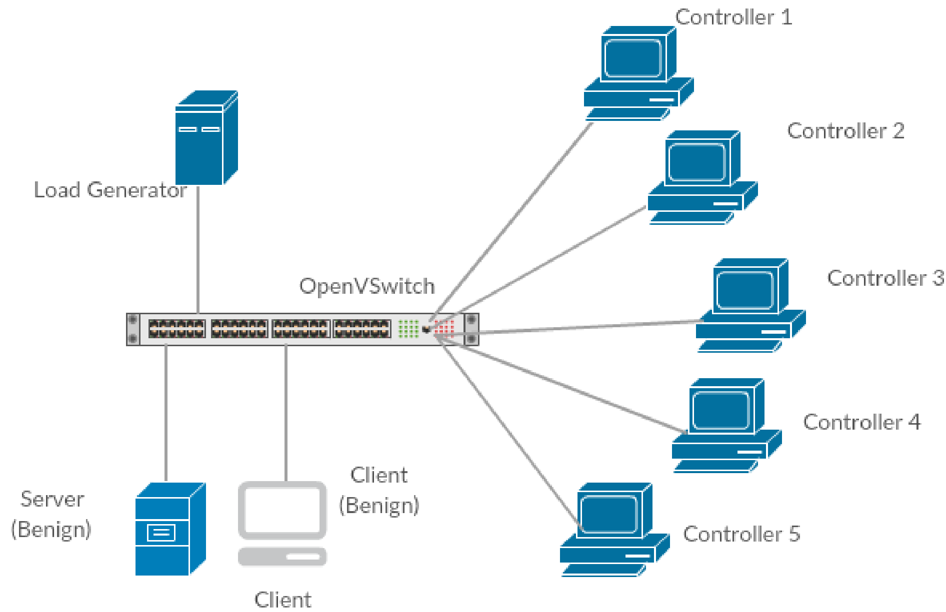


Figure 5.1: Network Setup

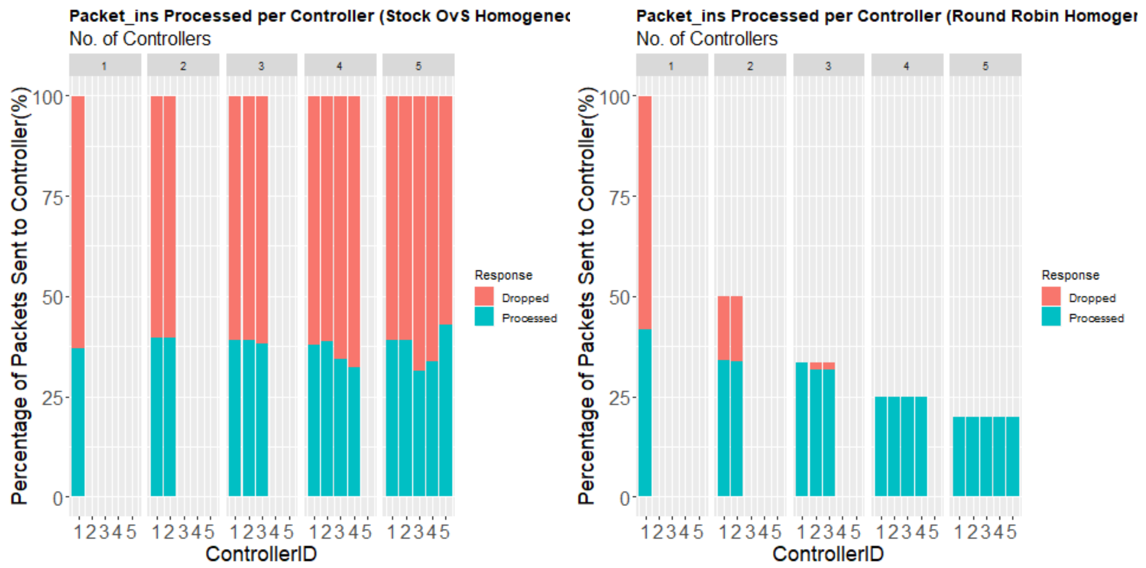
The Homogenous Controller pool maintains similarity between controllers. We use 5 Ryu controllers each hosted on a virtual machine having 1 CPU core and 1GB RAM. Each controller is located on a host machine 1 hop from the switch ensuring they are equidistant from the switch and equally provisioned.

The Non-Homogenous Controller pool varies the controllers both in type and resources. Three of the controllers (Controllers 1,3, & 5) are Ryu controllers hosted on a virtual machine having 1 CPU core and 1GB RAM. Controller 2 is a Floodlight controller also hosted on a virtual machine with 1 CPU core and 1 GB RAM. Controller 4 is a Floodlight controller hosted on a server with 4 CPU cores and 32GB RAM. While they are all equidistant from the switch, there is notable disparity in the pool.

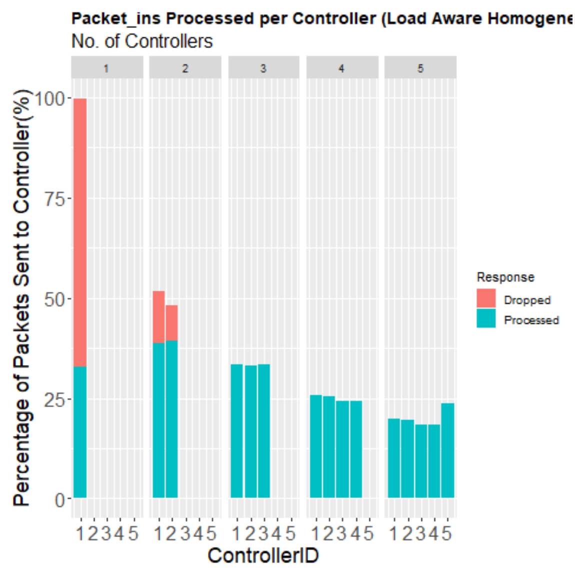
5.2.1 Controller Performance

We examine the first the distribution of flow requests within the controller pool using both algorithms. We specifically look at the individual controller's ability to respond to the flow requests assigned to it with and without load balancers' request distribution. Using the load generator, we generate approximately 250000 packet_ins

at a rate of 5000 packet_ins per second (pps) sending packets with randomised header fields to the switch. We first use the Stock OvS switch and then switches with both load balancer algorithms implemented. We vary the number of controllers in the controller pool for both the Homogeneous and Non-Homogeneous pool recording the number of requests assigned to each controller by the switch, and the percentage of which the controller is able to process. Ten rounds of this experiment are performed for each variation of the controller pool and switch and the average percentage of both processed and dropped requests across these ten rounds is recorded. The results are displayed in Figure 5.2 and Figure5.3.

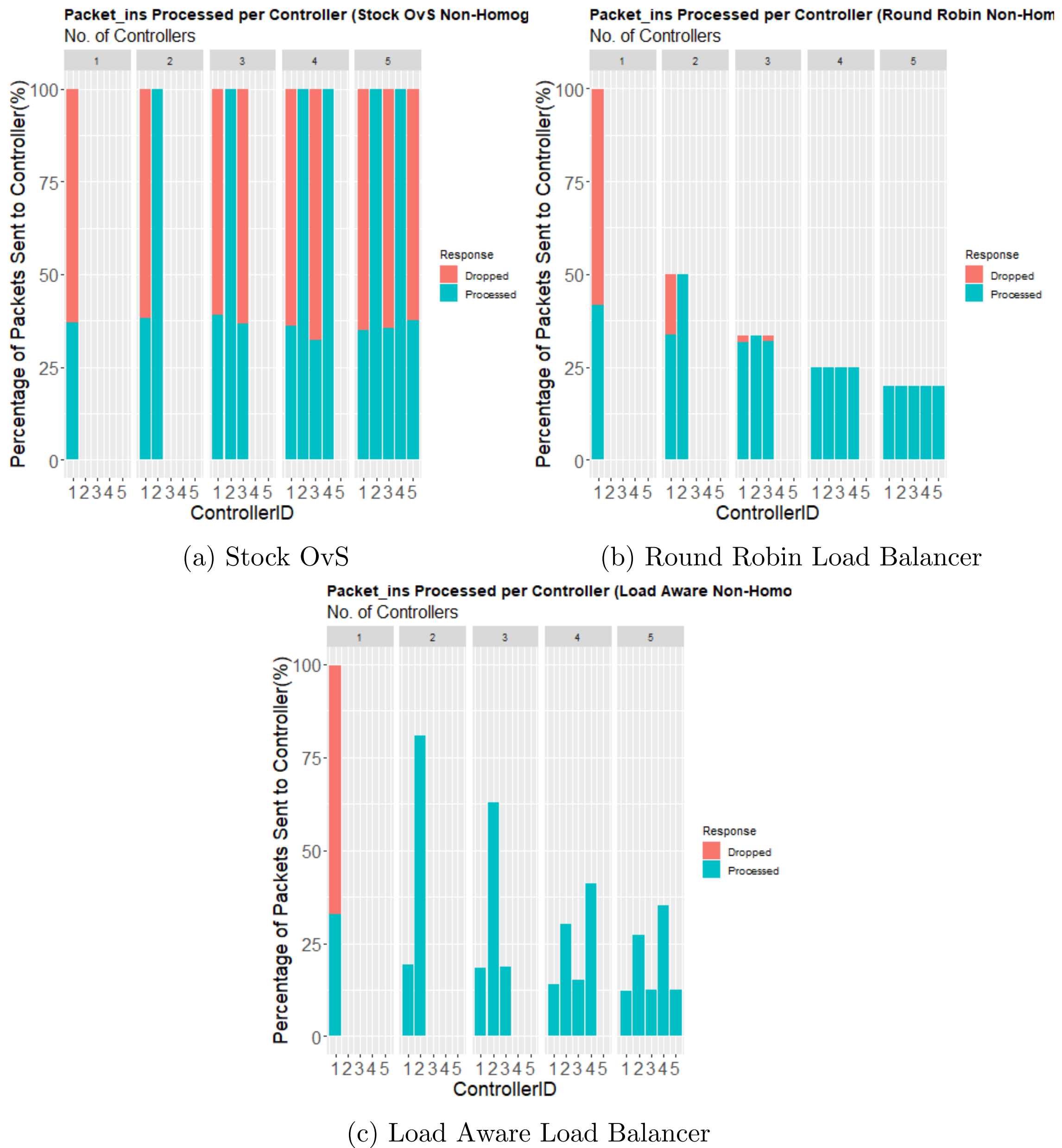


(a) Stock OvS (b) Round Robin Load Balancer



(c) Load Aware Load Balancer

Figure 5.2: Percentage of Packet_ins processed per controller (Homogeneous Controller Pool)



(a) Stock OvS

(b) Round Robin Load Balancer

(c) Load Aware Load Balancer

Figure 5.3: Percentage of Packet_ins processed per controller (Non Homogeneous Controller Pool)

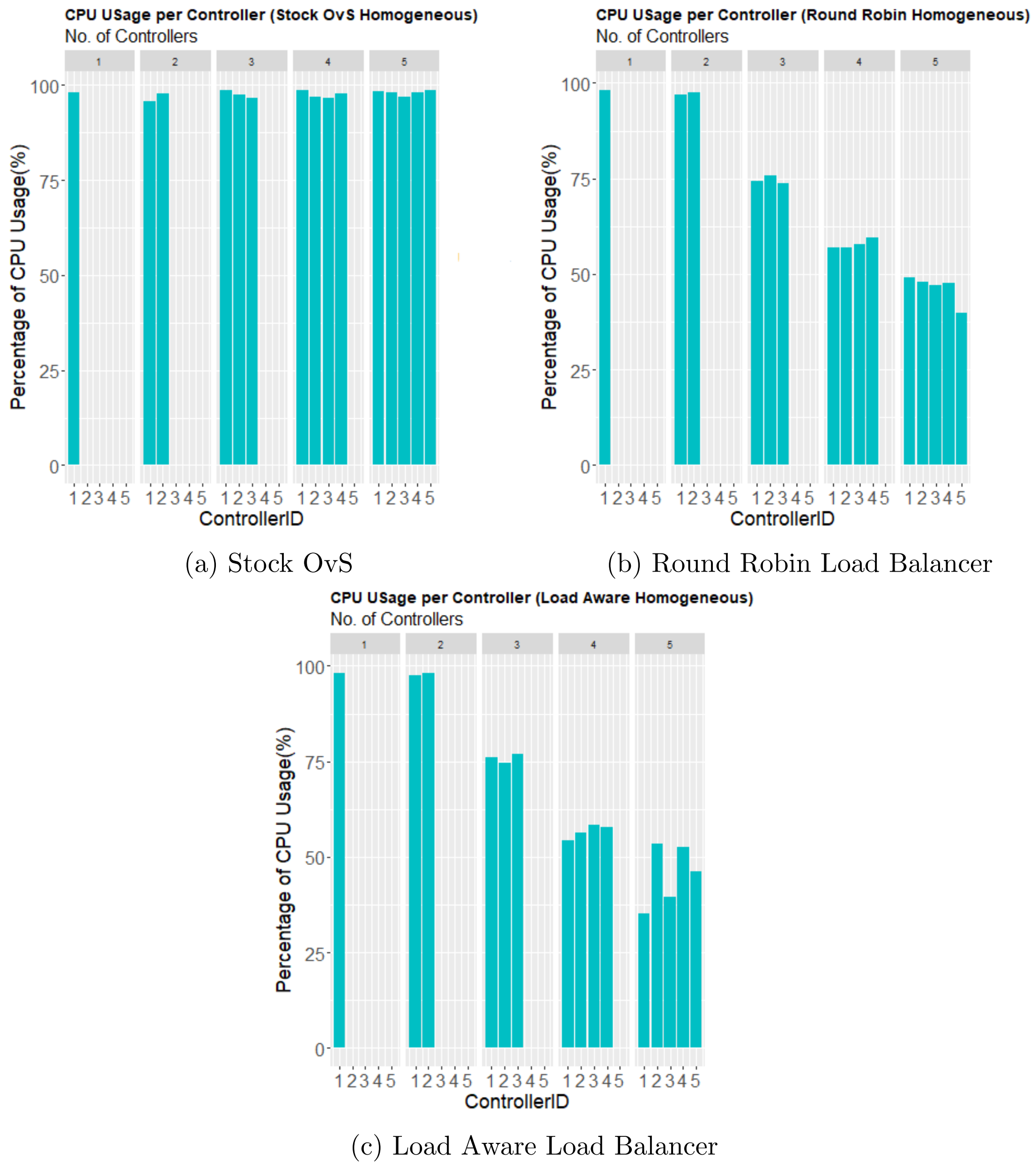
In each case, the figures show that the singular controller dropped around 67% of the flow requests assigned to it, showing immediately the drawbacks of the single controller architecture. Figure 5.2a shows that the Stock OvS forwards all requests

to all controllers. This poor usage of the controller pool results in between 60% and 75% of requests being dropped regardless of the number of controllers in use. Figure 5.2b shows that using the Round Robin approach with 2 controllers begins to improve the network performance significantly, dropping around 35% of the flows. Further increasing the number of controllers allows the network to handle more than 95% of the flows at 3 controllers and 100% with 4 and 5 controllers. The value of Load Aware algorithm is shown in Figure 5.2b where it is able to limit the amount of requests dropped with 2 controllers to 20%. Both controllers still process equal amounts of flow requests (since they have equal capabilities), however the algorithm extracts a better performance between the two by monitoring the load on each at any given time. Controller 2 records slightly more dropped flow requests simply because more requests were randomly assigned to that controller when both queues are full. We note that past 3 controllers, there is no additional benefit seen in the Load Aware Load Balancer as 4 controllers provide enough resources for the Round Robin algorithm to handle all flow requests when distributing them equally.

Several studies point to the Floodlight controller providing better flow rule installation throughput than Ryu [139][140]. Thus, in Figures 5.3a,5.3b and 5.3c, we see that controller 2 of the Non-Homogeneous pool (a Floodlight controller), due to its superior performance, processed 100% of the flow requests assigned to it in the 2 controller experiment. The Stock OvS achieved 100% packet_in response because at least one of the controllers was able to respond to all requests but this is again a poor use of resources. Both Controller 1 and Controller 2 were equally assigned 50% of the flow requests by the Round Robin algorithm, however Controller 2 significantly outperforms its counterpart. The Load Aware Load Balancer exploits this performance disparity as shown in Figure 5.3c. With just 2 controllers it is able to give a 100% response to flow requests by providing more flow requests to the better performing controller. As a rule with the Load Aware Load Balancer, controllers that respond faster are apportioned more requests according to their rate of response. This explains as well why in the 4 and 5 controller experiment in Figure 5.3c, we see the 2 Floodlight Controllers handling the bulk of the requests. Controller 4 in particular is apportioned the highest amount as it is a better performing controller with superior hardware resources.

5.2.2 CPU Load Tests

We examine here the effect of the switch's load distribution on the CPU usage of the controller processes under high load. Using the load generator, we send packets at a rate of 2000 packets per second to the controllers for 30 seconds and record the average CPU usage of the controller process on each controller during the 30 seconds. This is done 5 times for each controller pool and switch combination and the average CPU use over 5 rounds is recorded. We use first the Stock OvS and then both algorithms implemented in the switch. We vary the number of controllers in the controller pool for both the Homogeneous and Non-Homogeneous pool and display the results in Figure 5.4 and Figure 5.5.

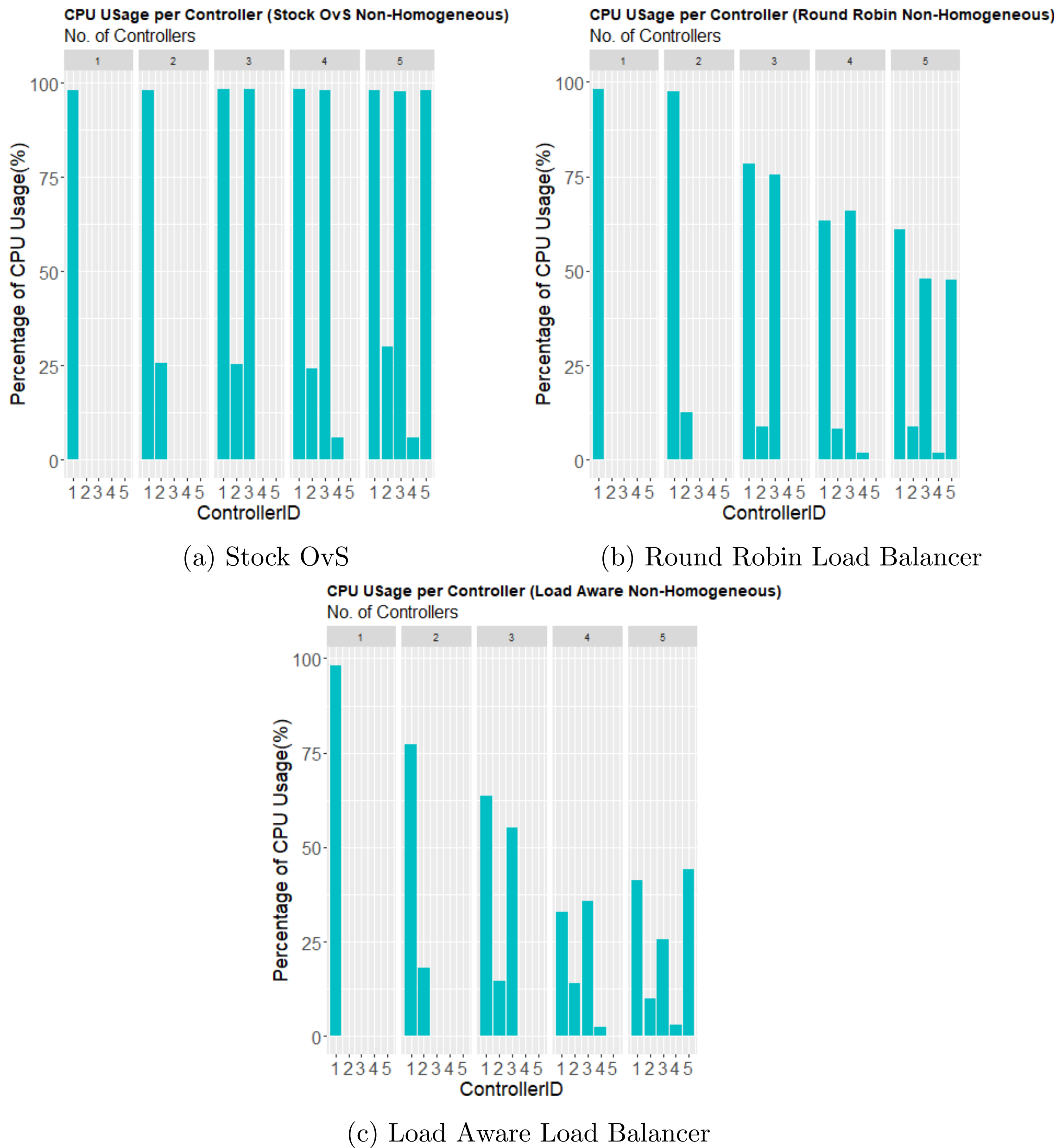


(a) Stock OvS

(b) Round Robin Load Balancer

(c) Load Aware Load Balancer

Figure 5.4: CPU Load per controller (Homogeneous Controller Pool)



(a) Stock OvS

(b) Round Robin Load Balancer

(c) Load Aware Load Balancer

Figure 5.5: CPU Load per controller (Non Homogeneous Controller Pool)

The results of the experiments show that the CPU usage of the Homogeneous controllers under the Stock OvS is high, regardless of the number of controllers connected (Figure 5.4a). This is consistent with the results of the packet_in count

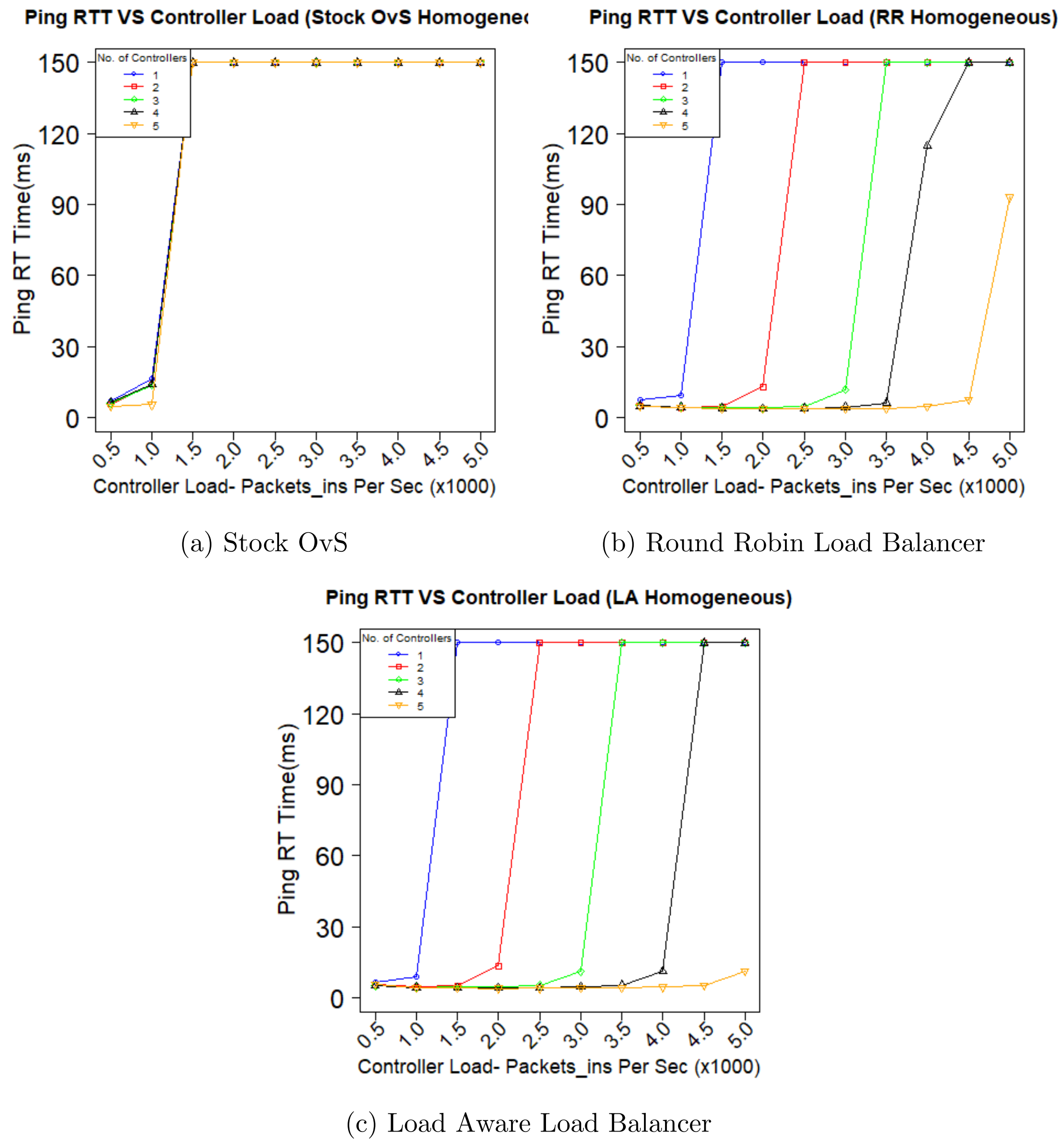
experiment in the previous section which indicated that the Stock OvS sends all packet_ins to all controllers. We see here as well that from three controllers upward, the CPU usage of the controllers begins to fall as the switch distributes the flow requests, ensuring that no single controller is overloaded. Figures 5.4b and 5.4c show that both the Round Robin and the Load Aware load distribution in the switch produce similar results here due to the controllers being the same (though controller 5 appears to require slightly less CPU power than the others for the same number of packets in the Round Robin).

With the Non Homogeneous controller pools, the controller CPU usage shows vast disparities. The Floodlight controllers, which we have seen to provide better service, record the lowest CPU usage in all cases while Ryu controllers are high. While the Stock OvS switch keeps the Ryu controllers at maximum CPU usage, both the Round Robin and Load Aware distribution in the switch show the CPU usage of its Ryu controllers consistently fall as more controllers are added. The Load Aware load balancer again performs better than the Round Robin as in each round, the CPU usage levels of its Ryu controllers are less than the Round Robin counterparts.

5.2.3 Latency Tests

We test the responsiveness of the controller pool under varying loads by measuring the Round Trip Times of pings sent between the client and server. The flow rules in the switch have a hard-timeout of 1 second and the pings are spaced 2 seconds apart, ensuring that each new ping packet goes to the controller. Thus, the path of the Ping and response is $client \rightarrow switch \rightarrow controller \rightarrow switch \rightarrow server \rightarrow switch \rightarrow controller \rightarrow switch \rightarrow client$. This allows us to measure the additional latency the ping experiences due to controllers under heavy loads. We send 20 pings and find the average of the pings' RTT (Round Trip Time). According to a recent study [186], the maximum tolerable latency for an interactive user based service such as voice communication is 150 milliseconds. We therefore set a maximum wait time of 150 milliseconds on the pings and all unresponsive pings are recorded as 150ms, indicating they have surpassed the maximum value. We perform these experiments using the Stock OvS and switches with both load distribution algorithms implemented. For each switch, we perform the ping test under controller loads of

between 500 and 5000 packets per second. We vary as well the size of the controller pool in each experiment and display the results in Figures 5.6 and Figure 5.7.

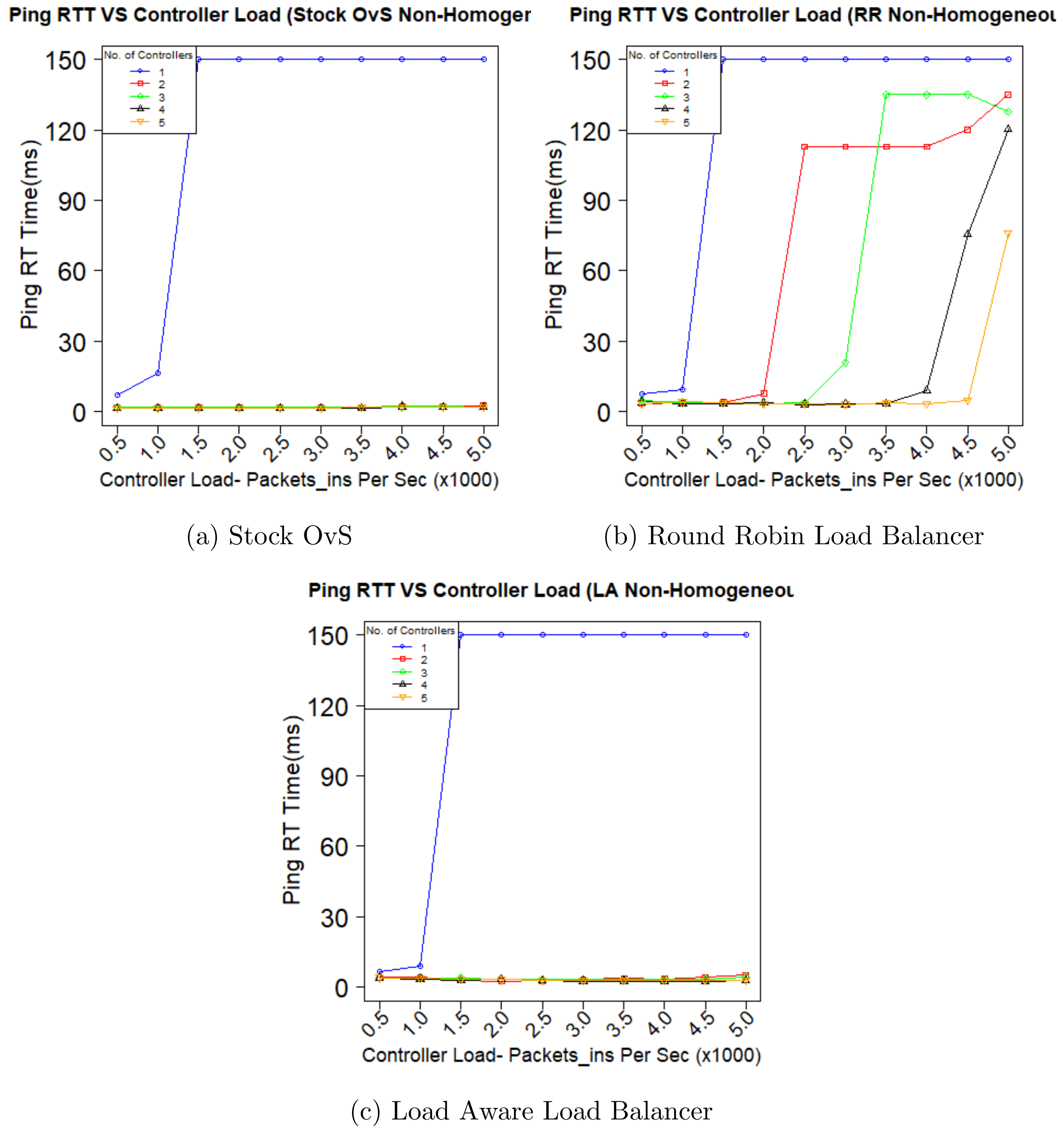


(a) Stock OvS

(b) Round Robin Load Balancer

(c) Load Aware Load Balancer

Figure 5.6: Latency of Pings with multiple controllers (Homogeneous Controller Pool)



(a) Stock OvS

(b) Round Robin Load Balancer

(c) Load Aware Load Balancer

Figure 5.7: Latency of Pings with multiple controllers (Non Homogeneous Controller Pool)

We first note that with the switch using either load distribution algorithm, increasing the controller pool size results in better response times from the pings, however regardless of the Homogeneous controller pool size, the Stock OvS registers

ping RTTs 5ms-16ms up to 1000pps but registers poor connectivity over 1000pps (Figure 5.6a). With one controller, a load of 1500pps or more quickly overwhelms the controller. Since the Stock OvS sends all requests to all controllers, this means that the responsiveness of the controller pool is only as good as its best controller which is capped at 1500pps here. Using the Homogeneous controller pool, the Round Robin approach which evenly distributes the requests among all the controllers shows a gradual increase in performance as the controller pool grows (Figure 5.6b). For 1 controller, it maintains RTT at approximately 4ms up to 1000pps with 2 controllers RTT of less than 4ms up to 1500pps. At 5 controllers, it maintains an RTT of 4ms up to 4500pps. The Load Aware distribution algorithm shows similar results with the same controller pool (Figure 5.6c) but performs slightly better at 5000pps. With 5 controllers, the Round Robin registers an average RTT of 93 ms while the the Load Aware Load Balancer maintained an RTT of approximately 11 ms.

With the Non-Homogeneous controller pool, the Stock OvS's policy of sending all requests to all controllers works out well in this case as it is able to find at least one controller (the Floodlight controllers) which provides good service. Thus, it is able to with 2 or more controllers, it is able to maintain an average RTT of 1-2 ms. In a different network, with less well provisioned controllers, this may not be the case (as we saw with the Homogeneous pool). As stated before, the controller pool is as good as its best controller under the Stock OvS but it does not make the best use of resources. The Round Robin distribution Algorithm in Figure 5.7b shows a notable increase in performance when used with the Non-Homogeneous controller pool, providing better RTTs than the Homogeneous controller pool (for example with 2 controllers is approximately 112ms between 2500pps and 4000pps while there was no connectivity over 2000pps for the Homogeneous pool). With the introduction of the Floodlight controllers to the Round Robin distribution by the switch, the service of the pings becomes a game of chance- whether the pings will be sent to a Floodlight controller and receive good service, or be sent to a Ryu controller and receive delayed service. With 2 Ryu controllers (Figure 5.6b), the Round Robin load distribution stopped receiving service after 2000pps, however with 1 Ryu and 1 Floodlight controller, some were sent to the Floodlight controller which reduced the overall average RTT of the 20 pings. The pings actually have a better chance of receiving service with 2 controllers than with 3 as there is a 50% chance of being sent to the Floodlight controller vs a 33% chance with 3 controllers. This may explain why

between 3000 and 4500pps, 2 controllers provide better service (112ms-120ms) than 3 (approximately 135ms)-more pings went to the Floodlight controller. As shown in the Controller Performance tests (Section 5.2.1), the Load Aware distribution algorithm in the switch is able to exploit the disparity in performance by weighting the better performing controllers with more flow requests. The result of this here is that it provides the best response times and latencies for multiple controllers giving the minimum latency in the RTTs in each case- 2ms-6ms (Figure 5.7c). It produces optimal network performance due to its ability to intuitively detect which controller is best suited for the incoming flow request ensuring the fastest responses and fewest dropped requests.

5.2.4 Processing Time

Finally, we measure the additional processing time the algorithms inflict upon the flow requests. While we have evidenced the performance increases that these algorithms produce in the control plane, the solutions here should not negatively impact the switch's efficiency in forwarding the flow requests to the controller. Thus, in measuring the additional time taken to run the implemented algorithms, we aim to assess the practicality of this solution in deployment. We record the time taken by each algorithm to select the appropriate controller for each packet_in and calculate the average time taken for this process over 10000 packet_ins. This is done for each controller pool size and the average times for each are recorded in Table 5.3.

Round Robin Load Balancer

Number of Controllers	**Average Time**
1	2925μs
2	2986μs
3	2614μs
4	2398μs
5	2460μs

Load Aware Load Balancer

Number of Controllers	**Average Time**
1	4014μs
2	4067μs
3	4687μs
4	4741μs
5	4921μs

Table 5.3: Load Balancer Processing Times

The results of the experiment indicate that the additional processing time added by the algorithms in selecting an appropriate controller is minimal- $<5\mu$s on all

counts for both the Load Aware and Round Robin Load Balancers. We note that the Round Robin algorithm, due to its lack of complexity, takes less time than the Load Aware Load Balancer algorithm in each instance. We also note that while the Load Aware Load Balancer registers a trend of marginal time increases as the number of controllers increase, no such discernable trend is seen with the Round Robin Load Balancer as its measurements appear to fluctuate between 2.3μs and 2.9μs. However, in both cases, the algorithms do not prohibitively increase the amount of time taken to process and forward the packet_ins.

5.2.5 Summary

We investigate here the benefits of enabling the SDN switch to perform flow request distribution to balance the load among the available controllers in its control plane. We demonstrate that this affords significant improvements to the network performance under high loads. The unmodified SDN switch forwards all of its flow requests to all of the controllers connected to it, causing redundancy in the tasks and making inefficient use of resources in the network. By increasing the intelligence of the switch, we are able to extract better performance from the network and make efficient use of the available resources, ensuring no one controller is overloaded while others are under-utilized. The Load Aware distribution algorithm implemented in the switch in particular enabled the switch to infer the loads on each controller connected to it at any given time and apportion the flow requests to the one with the lightest load. This brought out the best performance among the switches evaluated and clearly demonstrated how intelligence in the switch can bring about better performance.

5.3 Malicious Packet_in Filter

We examine here the benefits of enabling the switch to distinguish between malicious and legitimate flow requests from a port. We aim to show here that network performance is significantly improved and the control plane offered an additional layer of protection by increasing the switch intelligence such that it automatically drops flow requests deemed to be intentionally harmful to the network. We first evaluate Random Forest Classification conceptually, looking at its ability to classify datasets based on the attributes extracted from the traffic. We then integrate the

Random Forest classification and filtering module into Open vSwitch and examine its ability to block attacks from entering the core and prevent the controller from becoming overwhelmed by an attacker.

5.3.1 Passive Evaluation

5.3.1.1 Training Phase:

We first conduct an evaluation of our Machine Learning attributes on attack and benign traffic datasets. We train the *ranger* classifier using the Enterprise network dataset [182], along with a generated attack dataset that produces approximately 2000 flow requests per second. Using the java script described in the previous chapter, we simulate a Flow Table of 500 spaces and extract from both datasets the flow requests that would be generated by the recorded traffic. For each flow request, we extract each of the features described in the previous chapter, treating each unique source IP address in the benign dataset as coming from a unique port and all of the attack traffic as having come from a single port. The output is a file that contains the following information for each source:

- TimePeriod is the time taken for 25 requests
- Entropy is the entropy of the packet headers over the last 25 requests
- NumberOfRequests is the number of requests made by the port in the last 60 seconds

Imbalancing and Overfitting We feed the csv file, which consists of approximately 122000 data points, into the *ranger* Machine Learning classifier, configured to use 100 Decision Trees, and save the tree output to a file which will be passed into the classification phase. The dataset consists of 2000 benign packet_ins and 120000 malicious flow requests. The disparity in volumes occurs because the attack, by nature, generates exponentially more packet_ins within the same timeframe as the benign traffic trace. This creates an imbalanced dataset, a problem in Machine Learning data sampling in which one class within the sample dataset is rare in comparison to the others. This can lead to poor training if the decision trees only sample from the malicious class in the dataset or do not sample enough of the benign class to accurately distinguish between the two classes. However, in this dataset, the sample size for each tree is also large enough (66%- approximately 88000 datapoints)

to render “no benign samples selected” a highly unlikely circumstance. Also, the difference between the malicious and benign attributes are so distinct that even a few datapoints of the benign class are enough to allow the trees to accurately train and classify. This is evidenced in the evaluation in Table 5.5 which shows that even with a few trees, the algorithm accurately classifies the dataset.

Another concern for data with such distribution is Overfitting. Overfitting occurs when the Machine Learning algorithm learns the noise in the data and assumes it is normal which causes misclassifications. The algorithm learns the noisy data and may erroneously classify some of the new data in accordance with the anomalies or outliers it has learnt. These anomalies or outliers often do not apply to the new data and negatively impacts the results giving a greater level of inaccuracy. Within the training dataset used here, there is a high distinction between the malicious and benign datasets such that the data contains very little noise. Only, the initial packets of the attack create a small amount of noise as they appear to the algorithm to be benign since the “Entropy” and “Number of Request” attributes in particular remain below malicious thresholds for the first few packets of the attack. Random Forest also decreases the chance of overfitting by increasing the number of trees and its random feature selection for the trees. The experiments illustrated in Table 5.5 show that even with a few trees, indicating that overfitting is also not a problem.

5.3.1.2 Classification Phase

We evaluate the accuracy of the classification on 3 points

- True Positive: the number of malicious flow requests accurately classified as malicious
- False Positive: the number of benign flow requests mis-classified as malicious
- False Negative: the number of malicious flow requests misclassified as benign

Using the output tree file of the Training phase, we run the *ranger* classifier first on the same file it was trained on (which held both the malicious data and the legitimate data from [182]), then on two other legitimate datasets [184][183] combined with the malicious dataset. We give the aforementioned values in percentages in Table 5.4. The results demonstrate that despite varying legitimate data, we are

able to determine with high accuracy, which flow requests are legitimate and which are malicious. The number of false negatives remains constant across all datasets. These are the initial packets at the beginning of the attack which appear to be legitimate because the attack is just beginning.

Dataset	**True Positives**	**False Positives**	**False Negatives**
Enterprise Network Dataset [182]	99.998%	0%	0.002%
Bot Network Dataset [184]	99.880%	0.118%	0.002%
ICS Lab Network Dataset [183]	99.969%	0.029%	0.002%

Table 5.4: Random Forest Accuracy Evaluation

Number of Decision Trees: We also look at the effect of the number of trees used in learning on the classification. This is a key factor for implementation into the switch. The more trees that are used during classification, the longer the classification process in the switch takes. Since processing time within a network is a premium resource (we do not want to add large delays to the packets), any additional delays added by the classification process should be minimal. We aim to learn here if there is a tradeoff between time and accuracy which must be considered and what the optimum number of trees is which will allow us a fast and accurate classification. We use the same dataset for learning as before (Enterprise network dataset [182] for benign traffic & generated malicious dataset). We take the dataset which performed the worst in the initial tests (Bot Network Dataset [184] & generated malicious dataset) for classification testing and adjust the number of trees used in each round. The results are displayed in Table 5.5. The results demonstrate that while 100 trees give the highest accuracy, even with as few as 10 trees, we are able to distinguish with extremely high accuracy malicious from legitimate flow requests.

Number of trees	**True Positives**	**False Positives**	**False Negatives**
1000	99.856%	0.142%	0.002%
500	99.855%	0.143%	0.002%
100	99.879%	0.119%	0.002%
50	99.855%	0.144%	0.001%
10	99.810%	0.188%	0.002%

Table 5.5: Effect of the Number of Decision Trees on Random Forest Accuracy

Attribute Weighting

To better understand the weight of the attributes selected for training, we measure the variable importance of these attributes. The variable importance is a significant factor within Machine Learning as it indicates which variables have the most predictive power. Variables with low importance have little effect on the outcome of the classification while the values of variables with high importance have a significant impact on the classification made. We measure the variable importance using the Mean Decrease in Accuracy (MDA) measurement, in which a variable is selected and its values are randomly permutated. The MDA is a measurement of the decrease in accuracy of the classification as a result of these permutations. Variables which register a high MDA carry more weighting in the classification than those which score low MDAs.

The *ranger* library in R (built on the C++ code used in our implementation) provides an "importance" function which calculates the importance value of the attributes using the MDA measurement in the dataset. We train the data 10 times and record the results of the importance function each time. The results, given in Table 5.6 indicate that the Time Period attribute is of greatest importance, followed by the Entropy attribute which is almost equal in weighting. The Number of Requests attribute is significantly less important in the classification process. Each number in the "Measured Weighting" column shows the reduction in the Random Forest's performance (MDA) when the variables were changed, thus the greater the performance change, the more important that feature is.

Attribute	**Measured Weighting**
TimePeriod	0.01626592965
Entropy	0.0154981306
NumberOfRequests	0.00067799137

Table 5.6: Weighting of Attributes in Classification

5.3.2 Active/Live evaluation

Having determined that we can classify with high accuracy, malicious and legitimate flow requests, we implement in the switch the functions to extract the necessary attributes in Open vSwitch as described in the previous chapter. We implement

the module using 11 trees for classification having seen that as few as 10 trees can provide accurate classifications. We use an odd number of trees to avoid a tie in the binary ("malicious" or "benign") voting among trees. We examine here the switch's ability to identify and drop malicious flow requests before they are sent to the controller and the overall effect this has on the network performance. For our experiments, we use the network setup shown in Figure 5.8, using the "attacker" to generate malicious flow requests and the Client and Server to test the network performance for legitimate traffic where necessary. We use a single controller for these experiments which is a Ryu instance hosted on a quad-core desktop machine with 32GB of RAM. On a similarly provisioned dedicated host (32GB RAM, 4-core CPU), we run an Open vSwitch instance, using both the Stock Open vSwitch and the Random Forest Enabled Open vSwitch instances and comparing their outputs across a number experiments. The attacks performed here involve sending packets with randomized Destination IP Addresses to the switch at varying rates.

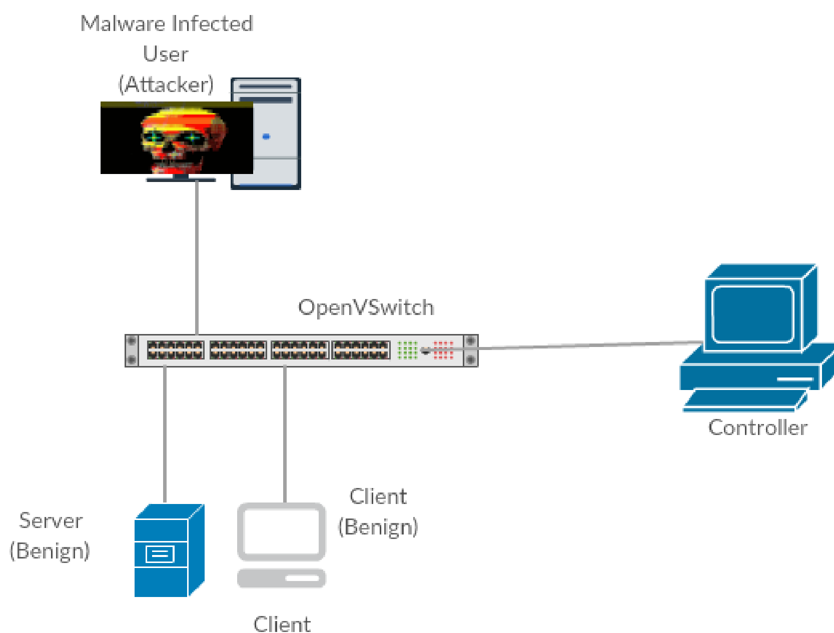


Figure 5.8: Network Setup

5.3.2.1 Network Connectivity

We compare the difference in network connectivity when under high loads using the Random Forest Enabled (RFE) switch and the Stock OvS switch. We use similar Ping experiments as used to evaluate the Load Balancer. We perform 20 Pings while varying the frequency of attack packets which generate flow requests from the attacker to determine the response times with and without the classifier. In each round of experimentation, the client attempts to send 20 pings to the server and we record the response times of each. Flow rules in the switch flow table are set to have a hard timeout of 1 second and the pings are spaced 2 seconds apart to ensure subsequent pings must return to the controller. We set again a timeout of 150 milliseconds and any pings that receive no response or take longer than this are recorded as 150 milliseconds. We then find the average of the times recorded for the 20 pings under each attack frequency both with and without the Random Forest Filter enabled and display the results in Figure 5.9.

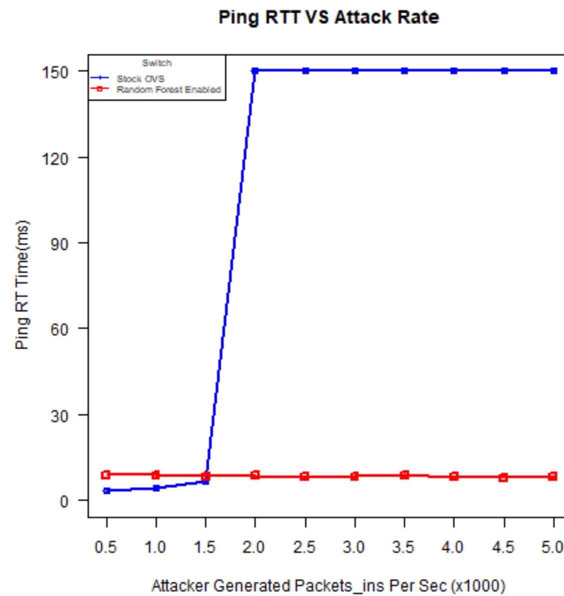


Figure 5.9: Network Connectivity under attack

With the Stock OvS switch, the pings receive no response after 2000pps. The RFE switch correctly classifies and filters out the attack packets freeing the controller to provide the best service to the legitimate ping packets. The legitimate pings under RFE remain at the minimal RTT value of around 7 milliseconds re-

gardless of the attack rate on the network.

5.3.2.2 Bandwidth

We examine here the effects of the attack on the bandwidth experienced between the client and the server both with and without the Random Forest Filter enabled. We enable the FIFO eviction feature on the Switch to mimic the attack proposed previously. This causes both a controller saturation and regular eviction of the legitimate flow rule due to malicious rules being inserted into the switch, both of which contribute to a lower bandwidth experienced by the legitimate flow rule. We again perform the experiments under a range of attack frequencies. For each malicious flow request frequency generated by the attacker, we measure the bandwidth using *iperf* [138] over a 60 second flow. We repeat each *iperf* flow 5 times, gaining 5 results for the bandwidth under each attack rate and find the average. This is performed both using the Stock OVS switch and the RFE switch and the results are displayed in Figure 5.10.

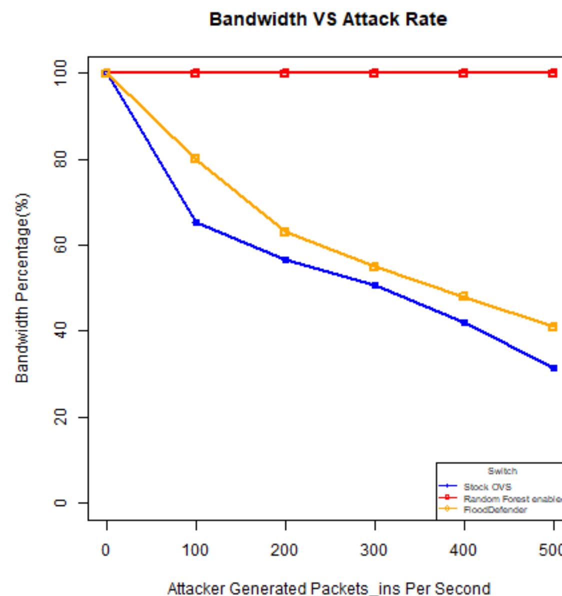


Figure 5.10: Network Bandwidth under attack

We compare our results here to those achieved by the FloodDefender system [99]. FloodDefender applies a "rough" threshold based filter to remove a portion of the

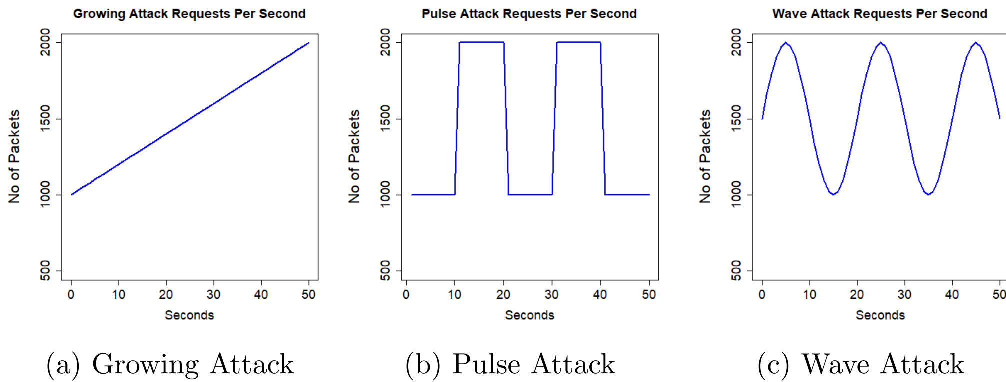
attack requests by using a low level controller application which examines the frequency of incoming flow requests. The threshold, based on the frequency of requests from a source must be set to a value that does not sacrifice legitimate requests attempting to reach the controller. This inevitably allows some attack traffic through to the controller as well. In their experiments, their hardware environment produces a bandwidth reduction of 75% under peak attack rates (before their system is implemented). We scale our attack rates to produce a graph comparable to theirs in our own network setup, registering a gradual bandwidth reduction from 100% to 30% as the attack rate increases. Under attack, their system reduces the rate at which the attack reduces the bandwidth experienced in the network. Despite this, under peak attack rates, the bandwidth is reduced to 40% of its original value with the FloodDefender system enabled (hardware environment). We demonstrate here that our RFE switch, having filtered out the attack packets, enables the client and server to maintain 100% bandwidth regardless of the attack rates. The threshold chosen FloodDefender does not protect the controller as fully as the RFE switch solution which removes almost all attack traffic before it gets to the controller resulting in the significant performance improvements.

5.3.2.3 False Negative Count

Just as we evaluate the False Negatives in the Random Forest classification on the datasets, we evaluate here the number of malicious flow requests which bypass the filter and arrive at the controller. We perform the attack at varying frequencies, and monitor at the controller, the number of malicious flow requests it receives with both the RFE and Stock Open vSwitch.

We compare our results here with the FlowRanger [112] system which uses queues in an attempt to prevent malicious flow requests from monopolizing the controller. We mimic their simulated experiments in our own network setup. While they stipulate that their network controller is able handle 100 requests per time slot, we adjust for our own network (using 1 second as 1 time slot), which the Ping experiment showed stops receiving service at 2000 requests per second. Their simulated experiments made use of 3 different attacks patterns which varied the attack rate between 50 and 150 packets per time slot over 50 time slots. We perform similar experiments, using a range instead of 1000 to 2000 packets per second. We perform

the three different attacks, Growing, Wave and Pulse, with varying rates of attack requests per second and display attack rates over the 50 second attack in Figures 5.11a-5.11c. For each attack, we record at the controller the number of attack requests it receives using both the Stock OvS switch and the RFE switch and display the results in Figure 5.12.



(a) Growing Attack (b) Pulse Attack (c) Wave Attack

Figure 5.11: Attack Patterns

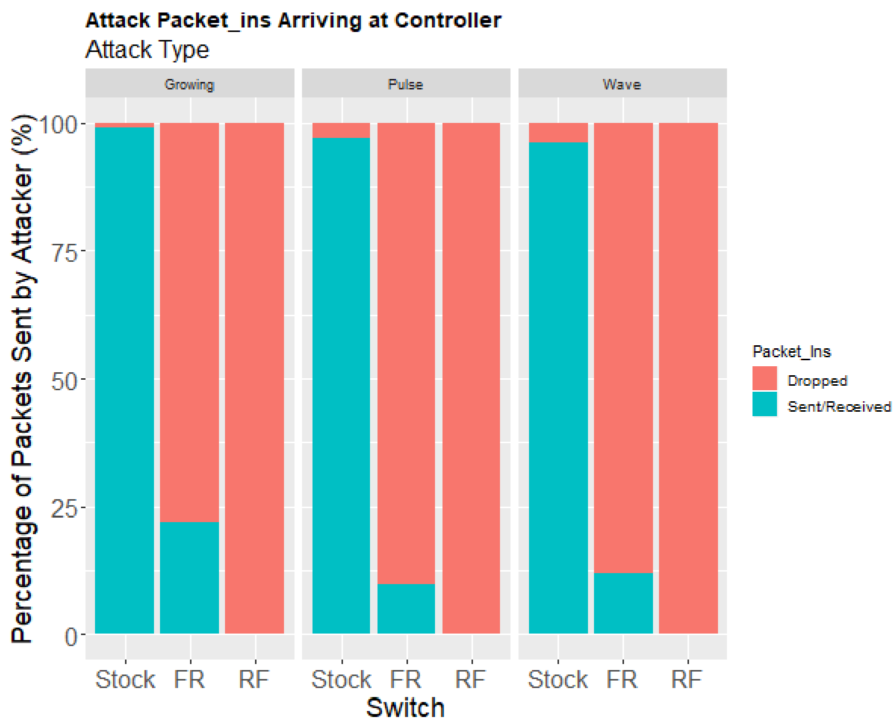


Figure 5.12: Attack Flow Requests Serviced by the controller

The controller receives a high number of attack requests from the Stock OvS

switch as this switch forwards packets to the controller indiscriminately, only dropping a few requests when the controller queue is filled to capacity. The RFE switch by contrast blocks nearly 100% of the attack requests going to the controller. The controller registers no more than 9 received attack requests regardless of attack rate or pattern. This performance significantly improves upon the FlowRanger system which services 1000 malicious requests out of 5000 in the Growing attack, approximately 400 out of 4500 in the Pulse attack and 500 malicious requests out of 4400 in the Wave attack at far lower attack rates.

5.3.2.4 Connectivity Restoration

Our attacker model assumes the infection of a legitimate host with malware which causes it to behave maliciously in the network. Since the host itself is not inherently malicious, we aim to provide regular service to the host when it behaves legitimately while restricting its connectivity when it behaves maliciously. Our experiments here focus on this aspect, examining how quickly a maliciously behaving host is allowed to reestablish connectivity within the network after it stops behaving maliciously.

We use pings from the attacking host to the server to represent legitimate behavior of the infected host. In this experiment, the infected host behaves normally at first, sending pings to the server once per second. At time T20, the malware begins its attack which lasts for 10 seconds, after which it goes to sleep and reawakens to perform another 10 second attack at T40. Throughout this time, the attacker is continuously sending pings- from T0 to T60. We record the connectivity the attacking host receives during this time by monitoring the responsiveness of the pings to the server and display the results in Figure 5.13. The controller sets a hard timeout of 1 second on each rule, requiring every other ping to go to the controller (though the pings are 1 second apart, the 2nd ping manages to catch the rule in the switch just before it expires).

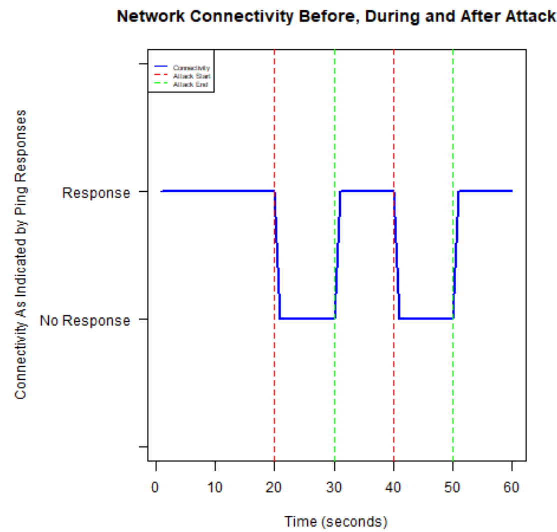


Figure 5.13: Connectivity of the Attacking Host around attacks

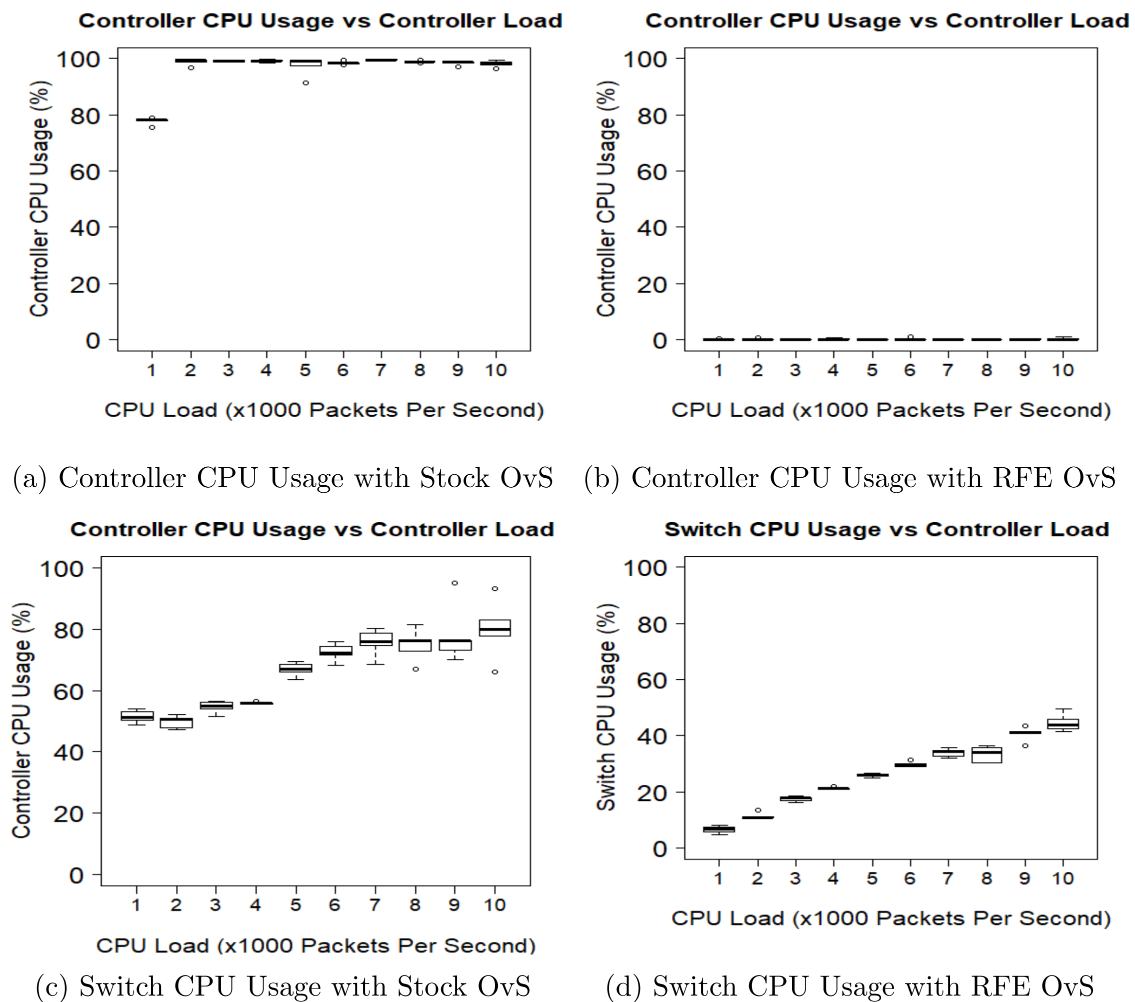
We see that as soon as the attack begins, the attacker's requests are filtered out, including the flow requests caused by the legitimate pings (evidenced by the lack of response). The attack is stopped at T30 and one second later, the pings receive controller service again and connectivity is restored to the infected host. The same result is seen when the attack is restarted at T40 and stopped at T50. At T51, the infected host is once again able to connect to the server. It takes no more than a second for connectivity to be restored to the offending host, but its requests are immediately filtered again when the attack restarts.

In extracting attributes for packet_in classification, the entropy of the requests arriving from a port is monitored in 1 second windows. A flow request which arrives <1second after the attack will have a low entropy as it will exist in its own entropic window. Additionally, since the attack sends several hundred packets per second, the Time Period for 25 requests during the attack is very small (approximately 12ms for a 2000pps attack). A legitimate flow request which arrives one second after the last attack packet will register a Time Period attribute of 1 second + the time taken for the last 24 packets of the attack- which makes the Time Period attribute much larger than 12ms seen for malicious flow requests. Wile the final attribute (Number of requests in 60 seconds) at this point is still "maliciously" high, connectivity being restored 1 second after the attack stops indicates that these values place 2 of the 3 attributes far enough outside the "malicious" range of values to swing the tree votes

towards "benign" 1 second after the attack.

5.3.2.5 Processing and Storage Overhead

We examine here the computational resources used during attack on both the controller and the switch. We perform the attack for 30 seconds in each round using the Stock OvS Switch and the RFE Switch and vary the rate of flow requests sent in the attack in each round. We record the average CPU usage of both the Open vSwitch process and the Ryu controller process during the 30 seconds of attack. At each flow request rate, the experiment is performed 5 times and the results are displayed in Figure 5.14.



(a) Controller CPU Usage with Stock OvS (b) Controller CPU Usage with RFE OvS

(c) Switch CPU Usage with Stock OvS (d) Switch CPU Usage with RFE OvS

Figure 5.14: Switch and Controller CPU Usage

The results of the experiments with the Stock OvS switch indicate that for all but the lowest attack rate, the controller utilizes 100% of the CPU power available to it as the switch forwards all attack requests to it (Figure 5.14a). We also see the CPU usage of the switch rising as it processes higher numbers of flow requests, peaking at approximately 80% CPU usage under 10000 flow requests per second (Figure 5.14c). By contrast, the controller CPU usage with the RFE switch remains at 0% with the exception of a few spikes (Figure 5.14b), which is likely caused by false positive, misclassified packets. The RFE switch even improves the switch CPU Usage as the average CPU usage ranges between 10% and 40% (Figure 5.14d). This is likely because the malicious packets are dropped before the flow requests are created and so the switch is able to save processing power by not going through the flow request creation and sending for the malicious packets.

We also evaluate the additional time required for the classification process. We record the time each packet takes to be classified for 2000 malicious packets and 2000 legitimate packets. We use using Pings spaced 2 seconds apart for the legitimate traffic and generate the attack traffic from the attacking host at a rate of 2000 packets per second. From the recorded times, we determine the maximum, minimum and average times taken to classify the packet_ins as malicious or benign. As shown in the results displayed in Table 5.7, we find that the classification process adds minimal delay to the packet_ins, considering RTTs for pings visiting the controller register around 7 milliseconds. At its peak in the RFE switch took approximately 20 microseconds to classify a malicious packet and approximately 40 microseconds to classify a benign one. This disparity can be explained by the classification trees. The trees require fewer node visits to classify packets as malicious than to classify as benign. The averages and minimums, however fall within a few microseconds of each other.

	Malicious Packet_in	**Legitimate Packet_in**
Maximum Time	19.539μs	38.372μs
Minimum Time	7.082μs	7.959μs
Average Time	8.554μs	11.824μs

Table 5.7: Times taken for packet_in classification

5.3.3 Summary

We examined here the benefits of an increase in switch intelligence which enabled it to filter malicious packet_ins, protecting the controller from intentional overloading. Our evaluation leads us to the conclusion that the addition of a packet classifier to increase the intelligence of the switch presents no obvious drawbacks, including the processing required. It demonstrates the ability to increase the performance of the network at least 75% in bandwidth and connectivity experiments and effectively protects the controller from malicious packet_ins which seek to overwhelm it without negatively affecting the legitimate activity of the network. The increase in switch intelligence also demonstrably reduced the CPU usage of the switch itself under high loads by dropping the flow requests before they formed into packet_ins and sent to the controller, a benefit controller based solutions would be unable to achieve.

5.4 Conclusion

In the previous chapter, we propose several additional functionalities for the SDN switch which increase its intelligence. We detail the implementations for two modules in support of the Intelligent Flow Request Handling functionality and examine the effects on the network of these modules in this chapter.

The evaluations in this chapter demonstrated that both the Load Distribution and Malicious Packet_in Filtration modules placed in the switch significantly improve the network performance, optimising use of and protecting the control plane resources. By implementing these modules into the switch, the switch no longer blindly forwards all flow requests to the controller but intelligently manages its flow requests to prevent control plane overloading.

In section 5.2 we evaluate the Load Distribution modules of the Intelligent Flow Request Forwarding. We found that both the Round Robin and the Load Aware Load Balancers increase the control plane efficiency, allowing the controllers to service more flow requests and reduce the CPU load of each controller and the latency sent by packets to the controller. Additionally, when compared with each other, the Load Aware Load Balancer outperformed the Round Robin Load Balancer in most scenarios, but particularly when the controllers in the control plane were not uniform. In Section 5.3, we evaluated the Random Forest Packet_in filter and de-

termined that it was approximately 99% accurate at distinguishing malicious and benign flow requests and significantly improved the network bandwidth, flow request latency, and the load registered on both the CPU and Switch, among other improvements.

This, along with the Switch Based Flow Rule Eviction functionality discussed in Chapter 3 empirically show the merits of increasing switch intelligence to enable it to perform tasks without controller intervention. The functionality outlined and evaluated here clearly shows that smarter SDN switches is an area which merits serious investigation. As we conclude in the following chapter, we summarize out findings and further discuss additional work in the area of switch intelligence to improve the network.

Chapter 6

Conclusion

In this thesis, we propose that the distribution of intelligence in SDN networks should not be mutually exclusive. In separating the network planes, the traditional SDN paradigm concentrates the intelligence in the control plane and regards the network switches as "dumb forwarding devices". We propose in this thesis that increasing the intelligence of the network switches by enabling them to perform tasks autonomously can improve the network performance and security. While the controller remains the "brains" of the network, the switch can perform some tasks which can take some of the load off the controller, enabling it to focus on more critical tasks such as making routing decisions.

6.1 Thesis Contributions

6.1.1 Analysis of initial steps toward increasing switch intelligence

6.1.1.1 Analysis of flow rule eviction vulnerabilities

To reduce the impact of Table Overflow issues in SDN, OpenFlow developers included a configuration which allows switches to evict rules in their flow table in favour of new rules. If the controller attempts to insert a flow rule into a full flow table, the switch can automatically remove the oldest rule in its table to make space for the new one instead of replying to the controller with an error message (Switch Based Flow Rule Eviction). This augmentation of switch functionality represents an increase in switch intelligence by enabling it to perform a task (flow rule eviction)

without needing controller intervention. This thesis outlines the benefits of this and takes a closer look at the implementation of this functionality. Analysis of this implementation indicated that while beneficial to the network, it opens a new vector for DoS attacks on the users of the SDN network. We described and demonstrated a potential attack on this implementation of switch flow rule eviction. We highlighted the issue that a malicious user in the network can repeatedly force legitimate rules to be removed from the switch flow table by creating new rules of their own after the legitimate rules have been inserted, forcing the older legitimate rules to be removed to make space for their rules. We demonstrated that using this method, an attacker can effectively control the bandwidth of legitimate users in the network since packets which must be diverted to the controller for instruction due to lack of matching rules in the switch flow table accrue additional latency. By controlling the frequency at which they push the legitimate rules out of the flow table, the attacker is able to control the number of legitimate user packets which must go to the controller which is directly related to the throughput the legitimate users experience.

6.1.1.2 Evaluation of alternative flow rule eviction implementations

While the FIFO flow rule eviction policy has some demonstrable shortcomings, the idea of increasing the switch intelligence through the ability to perform evictions is worth pursuing as it takes some load off the controller. We therefore examined several alternative flow rule eviction policies which could be implemented in place of FIFO. We derived 3 other potential policies from studies done on CPU cache-replacement: Least Frequently Used, Least Recently Used and Random Removals. Through simulation of the flow table activity under attack across a number of scenarios, we were able to perform an in-depth analysis of the effects of factors such as the intensity of the attacks, the size of the flow table and the background traffic on an attacker's ability to remove the legitimate flow rules from the switch across each of the eviction policies. Through the evaluation we came to the conclusion that among the policies evaluated, the Least Recently Used flow eviction policy performs best against attacks attempting to remove long lasting, heavy hitter flow rules from the flow table. Implementation of this algorithm into the switch achieves the intended goal of autonomous flow rule eviction without opening a vector for attack as the FIFO implementation does.

6.1.2 Further Switch Intelligence Augmentation

In seeing the merits of switch intelligence in taking load off the controller and contributing to better network performance through switch based flow rule eviction, we propose that switch intelligence be increased to enable SDN switches to perform tasks autonomously. In doing so, some of the burden is taken off the controller, allowing it to focus on other critical tasks and the performance and security of the network can be improved. To illustrate the merits of this, we design an intelligent SDN switch, focusing on the problem of Denial of Service within the network and outlining high level designs which increase the switch intelligence, allowing it to defend against DoS. We demonstrate the tangible benefits of such smarter switches by implementing modules which realise some of the high level design goals. We focus on addressing the problem of high control plane loads in SDN networks, caused by the "dumb" switch's policy of blindly forwarding all flow requests to the control plane.

Due to SDN switches' blind forwarding of flow requests, it has previously been shown that an attacker can generate flow requests to the controller at a high rate by sending large amounts of packets with randomized packet headers to the switch. Additionally, a high control plane load may be generated under benign circumstances such as a flash flood of traffic causing a large number of new flows in the network. Both of these force the controller to map a high number of flow routes and install the associated rules, overloading the controller and preventing some legitimate flows from being serviced. We therefore proposed to increase the intelligence of the switch to include intelligent forwarding of flow requests to reduce the problem of high control plane loads and show how smartter switches can aid the resiliency of the network. We realise this concept of intelligent flow request forwarding by designing, implementing and evaluating modules enabling the switch to effectively relieve the network of control plane overloading as much as possible. These modules, implemented in Open vSwitch (a popular SDN software switch), increased the switch intelligence by enabling the switch to accurately filter out malicious flow requests aiming to DoS the controller or switch flow table and efficiently distribute the legitimate flow requests among the available controllers to achieve the best service.

6.1.2.1 Controller Load Distribution

To implement flow request distribution, we proposed two designs of a Load Balancing module which allows a switch operating in a multi-controller architecture SDN network to efficiently partition its flow requests among the available controllers. The first design, a Round Robin selection system, enabled the switch to send a flow request to each of its connected controllers in turn, ensuring that each controller received an equal number of flow requests. This was particularly aimed at controllers which are equally provisioned in terms of CPU and memory resources and distance from the switch. In deployment this is rarely the scenario and the second design assumes that, not being perfectly equal, some controllers can better handle flow requests than others at any given time. The second design, a Load Aware load distribution system, allowed the switch to infer the load on each of the controllers connected to the switch at any given time by recording the number of outstanding or unanswered flow requests of each controller. For each flow request, the switch selected the controller with the least outstanding flow requests, inferring it to be the controller with the lightest load.

Our evaluations of both designs showed that both systems can effectively enable the switch to distribute its control plane load among multiple controllers achieving better service than the native Open vSwitch connected to multiple controllers. The Load Aware load distribution system in the switch was shown to work best however, as its ability to infer the ability of a controller to handle more flow requests caused it to outperform the Round Robin even under ideal circumstances for the Round Robin.

6.1.2.2 Filtration of DoS attack packets

To filter malicious flow requests maliciously intended to saturate the controller or frequently remove flow rules from the flow table, we implemented a Random Forest classifier into the switch to enable the switch to distinguish between malicious and legitimate flow requests. We first analysed the flow creation properties of legitimate traffic using several publicly available datasets. From this analysis, we extracted three core attributes of the attack which were used to classify malicious and legitimate flow requests. The *ranger* Random Forest implementation was used for

training and the trees generated by the program were extracted and fed into Open vSwitch at boot time. The algorithm for classification was also implemented into Open vSwitch which used the trees to classify a packet_in as malicious or legitimate. Using this module, the switch allowed legitimate flow requests through to the control plane while malicious flow requests were dropped.

Evaluation of this module showed significant improvements to the network performance while under attack when compared with the native Open vSwitch program. The intelligent switch was able to accurately recognize malicious flow requests and drop them, protecting the controller and enabling it to provide good service to the legitimate flow requests. The switch filter performed better than several other works proposed to defend against controller saturation attacks when the results were compared, and it was shown to reduce the CPU load on both the switch and the controller while under attack, incurring minimal latency on the flow requests due to the classification processing. All of these improvements make a strong case for increasing the switch intelligence, enabling it to perform tasks autonomously to protect the controller and improve network performance.

6.1.3 Research Impacts

The research in this thesis firstly demonstrates the need for security evaluation of protocols before deployment. While the Switch Based Flow Rule Eviction at first glance seems to be an excellent step forward in reducing the effects of the limited TCAM in SDN, a closer look showed that while it closed one vector for attack, it opened another. This portion of the thesis lends its support to the Security research community calling for a serious examination of the vulnerability impacts of both protocol and software before deployment rather than considering the security as an afterthought.

Secondly, this thesis pushes for the inclusion of intelligence in the SDN switch which would allow it to work in conjunction with the controller for the benefit of the network. The switch prototypes proposed and evaluated here effectively blocked DoS attacks aimed at the switch and controller and helped distribute legitimate traffic for better service at little to no cost. SDN's vulnerabilities has been one of the major hindrances to its widespread deployment. While this type of network has

demonstrated many benefits, its underlying infrastructure is open to many attacks such as the DoS attacks discussed in this thesis. With the proven benefits of smarter SDN switches, we push SDN closer to safe industrial deployment.

6.1.4 Summary

In summary, the thesis makes the following contributions

- Analysed initial attempts to increase SDN switch intelligence through switch based flow rule eviction and demonstrated a potential attack on the current implementation of the flow rule eviction policy which allows an attacker to control the throughput of legitimate users with which he shares the network
- Evaluated alternative eviction policies for their attack resilience concluding that removing the Least Recently Used rule provides the most resilience against attacks attempting to forcibly remove long lasting flows
- Proposed to further increase switch intelligence to increase network resilience by allowing the switch to perform tasks without the need for controller intervention and offered a design of such a switch resilient to DoS attacks on the network infrastructure
- Designed, implemented and evaluated modules enabling the switch to distribute flow requests among several controllers with the aim of providing better service to SDN networks under high controller loads through smarter switches.
- Designed, implemented and evaluated a packet_in filter module enabling the switch to autonomously distinguish malicious from legitimate flow requests and protect the controller from controller saturation DoS attacks

6.2 Future Work

In this thesis, we evaluate the resiliency of various switch flow rule eviction policies. We demonstrate the attack vector exposed by the First in First Out (FIFO) eviction policy and examine for attack resiliency, three other policies that could be implemented in place of FIFO. Extending the concept of switch intelligence, demonstrate that increasing the intelligence of the SDN switch can improves the resiliency of the

network against both malicious and legitimate controller saturation.

While we present a prototype here, we do not propose that the switch augmentations presented here represent a complete solution for an intelligent switch. In this section we acknowledge the space for further improvements, suggesting further work that could be done in line with what is presented within this thesis and outside of that scope as well.

6.2.1 Switch Flow Rule Eviction

We expand the concept of Switch Based flow rule eviction to look at other policies that could be implemented in place of the FIFO policy of OpenFlow 1.4. We evaluate the Least Recently Used, Least Frequently Used and Random Replacement policies, concluding that the Least Recently Used policy presents the most resilience to an attacker aiming to remove legitimate flow rules from the switch.

We draw inspiration for the flow rule eviction policies from the CPU cache replacement problem. The policies evaluated here are by no means the totality of the proposals for cache replacement. Several other policies have been implemented and evaluated within the CPU cache sphere which could be ported to the switch flow rule replacement problem [147][187][188]. One such example is the Time Aware Least Recently Used [189], a variation of LRU which considers the life span of cache content in deciding eviction. Since in most cases in SDN, rules either have an idle or hard timeout in the switch, this may prove to be a viable and well suited alternative policy. Alternatively, hybrid flow rule eviction policies which combine attributes from one or more eviction policies could also be implemented. This concept would seek to combine the best attributes of multiple policies. One example of such is a Least Frequently Recently Used policy, which could combine the best of both worlds in the Least Frequently Used and Least Recently Used policies by considering the popularity of a flow rule but balancing it with the recency of its use so that “dead” flows which were once heavy hitters do not remain in the switch longer than they should.

6.2.2 Load Distribution

To demonstrate the viability and benefits of smarter SDN switches, we implement and evaluate two designs for control plane load distribution performed by the switch. In both designs, the switch selects a single controller out of the pool of connected controllers for its next flow request. This reduces the redundancy in the system (the switch is no longer forwarding all requests to all controllers) and enables the switch to get better service out of the controllers with which it is connected.

The proposals in this thesis focus solely on the aspect of flow requests being sent into the control plane. With this focus, the switch ignores the issue of propagation of network updates between the controllers. All controllers should be able to maintain an accurate view of the flow table states of each switch under its control at all times. Our system does not currently consider this factor as it is outside of the scope of this project. While we specify that there should exist an inter controller communication system within the control plane to handle this, additional work on this module could look at its own communication protocol with the controllers to inform the others when one installs a flow into the switch table. Such a communication system is also useful for integration of the switch load distribution system with other controller-based defenses. Defense systems for the SDN network which assume all flow requests pass through it (e.g [99] and [106])could be subverted by a switch based load distribution system which actively chooses where to send flow requests. It may cause them to miss out on key statistics when monitoring. This could be handled within the control plane defense system by having the controllers propagate information among each other to ensure the decisions are made using accurate information. Careful consideration must be given to the risk of removing the benefits of the load balancing by bombarding the controllers with updates within such an implementation, however.

6.2.3 Malicious Packet_in Filter

The Random Forest Packet_in Classifier and Filter presents the largest opportunity for expansion of the work done in this thesis. As part of our switch intelligence augmentation proposal, we implemented and evaluated a packet_in filter in the switch which enables it to distinguish malicious from legitimate flow requests using Ran-

dom Forest classification and drops the malicious flow requests.

We highlighted the necessity of extracting from the attack immutable characteristics which an internal attacker cannot easily subvert without compromising the effectiveness of his attack. Training data for legitimate requests was gathered using several publicly available network traces and training was performed offline. This was done in absence of network generated training data specific to the network the filter would be employed on. The "training" portion of this filter concept is yet to be fully explored. The opportunity remains for implementation of a module which records the network behaviour under normal circumstances, allowing it to quickly detect when the behaviour of its own network has changed. Such a module could periodically record the behaviour of the ports it is connected to as "legitimate activity" and use this as its training data (online training). In this way, the switch is not reliant on another network's "good behaviour" to determine good behaviour in its own environment.

Additionally, since the switch performs its filtering and protection of the network in relative isolation (without interacting with the controller), further work in this area should include updates to the controller regarding switch activity. This may take the form of periodic messages to the controller informing it of the number of attack packets blocked, or may involve sending a sample of the attack packets to the controller for verification of accurate filtering. This is to ensure the controller has an accurate view of the traffic in the network and can take its own actions in keeping with network policies to address the attack if necessary.

6.2.3.1 Limitations

The attacker model our Packet_in Filter aims to protect against is an internal host which has been compromised or infected by malware. Detecting traditional DoS attacks or external attackers attempting to perform SDN focused attacks are both outside the scope of this thesis. There are several other systems which focus on both of these attacks within the SDN sphere and we propose that the filter in this system be made to work in conjunction with these other systems. We exclude ingress ports and ports connected to servers (e.g web server) from classification as they, by nature, generate a large number of flow requests within an abnormally short space of time

when compared with a host connected port. Systems such as [104] which monitor for external attackers by examining the entropy of destination addresses would be useful to implement alongside.

6.2.4 Further Switch Intelligence

We present in this thesis several augmentations for an SDN switch, increasing its intelligence to increase the resilience of the network. We by no means claim to present a complete solution here however, and the opportunity exists for further increases in functionality of the switch. With a focus on relieving the burden on the controller, many new ideas can be implemented into the switch. Previous examples include AVANT-GUARD's TCP attack filter [117] and Statesec [122] which detects DDoS attacks on hosts in the network. Further augmentation could allow the switch to monitor for Man In The Middle attacks or for packet signatures of malware. The switch could be envisioned not only as a forwarding device, but as a gatekeeper for the SDN network, blocking malicious activity of all sorts before it has chance to enter the network.

Controller Malfunction Detection, one attribute of an intelligent switch not implemented in this thesis, could be carried out by an extension of the Load Aware load distributor, which infers the load on a controller. It can also be expanded upon to make further inferences about a controller. A controller repeatedly assessed by several switches to be under high loads could be indicative of an attack in progress that the controller is unable to take action against or a Byzantine failure of some sort within the controller which is causing it to respond slowly. The switches could then inform the other controllers that feedback from this controller has been low and should be checked. Low response rates could also imply a controller behaving maliciously. Such a controller may appear to have a high number of unanswered requests to the switches, which could then alert other controllers to potential malicious behaviour from that controller.

Despite the potential benefits presented by augmenting the switch's intelligence, the greatest care must be taken to preserve the switch's core functionality: packet forwarding. Whatever additional functioning the switch is enabled to do should not

compromise its ability to quickly forward packets to their destinations.

Finally, we acknowledge the difficulty of hardware deployment for these and any other ideas which augment the switch. Within the SDN paradigm, the controller is meant to be easily configurable, with switches acting almost like an embedded system with limited functionality. The idea of intelligent switches flies in the face of this concept as it calls for switches that can be easily modified to increase their abilities. This is easily done in software switches such as Open vSwitch. It is significantly more difficult to convince vendors to include custom functionalities in hardware switches. Similar adoption difficulties were also acknowledged when the ideas of separated control and forwarding planes in hardware switches were first conceived [18]. Nevertheless, just as the OpenFlow protocol created a pseudo-standard which vendors could implement and allow for switch interaction with a centralised control plane, this presents an opportunity for a standard framework to be implemented in switches which enables modification of the functionality post deployment. Such a framework would allow network admins and programmers to easily alter what a switch does upon receiving a packet.

6.3 Concluding Remarks

In this thesis, we analyse initial steps to increase the intelligence of the SDN switch and propose further increases to switch intelligence which can help defend the network. Through an extensive evaluation, we prove the merit of the augmentations, showing the clear advantages to the implementation of intelligence in the switch.

While barriers to introduction exist in implementing such technology into hardware switches, the versatility of software switches allows us to try different approaches and prototypes to determine what works and what doesn't to increase the performance and resilience of the network before moving these frameworks into hardware.

Bibliography

[1] Office for National Statistics, "E-commerce and ICT activity, UK: 2016", 30 Nov. 2017 `https://www.ons.gov.uk/` [Cited on page 24]

[2] Chen, Yan, Toni Farley, and Nong Ye. "QoS requirements of network applications on the Internet." Information Knowledge Systems Management 4.1 (2004): 55-76. [Cited on pages 25 and 104]

[3] Prasad, Ravi, et al. "Bandwidth estimation: metrics, measurement techniques, and tools." IEEE network 17.6 (2003): 27-35. [Cited on page 25]

[4] Bodin, Lawrence D., Lawrence A. Gordon, and Martin P. Loeb. "Evaluating information security investments using the analytic hierarchy process." Communications of the ACM 48.2 (2005): 78-83. [Cited on page 25]

[5] Mirkovic, Jelena, and Peter Reiher. "A taxonomy of DDoS attack and DDoS defense mechanisms." ACM SIGCOMM Computer Communication Review 34.2 (2004): 39-53. [Cited on pages 25 and 47]

[6] McKeay, Martin. "SUMMER SOTI - DDOS BY THE NUMBERS", The Akamai Blog, June 19, 2018. `https://blogs.akamai.com/2018/06/summer-soti---ddos-by-the-numbers.html` [Cited on page 26]

[7] Matthews, Tim. "Incapsula survey: What DDoS attacks really cost businesses." Technical report, Incapsula (2014). [Cited on page 26]

[8] Sterbenz, James PG, et al. "Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines." Computer Networks 54.8 (2010): 1245-1265. [Cited on page 28]

[9] Leiner, Barry M., et al. "A brief history of the Internet." ACM SIGCOMM Computer Communication Review 39.5 (2009): 22-31. [Cited on page 33]

[10] Townsend, Anthony M. "The Internet and the rise of the new network cities, 19691999." Environment and planning B: Planning and Design 28.1 (2001): 39-58. [Cited on page 33]

[11] Hedrick, Charles L. Routing information protocol. No. RFC 1058. 1988. [Cited on page 33]

[12] Rutgers, Charles L. Hedrick. "An introduction to igrp." The State University of New Jersey, Center for Computers and Information Services, Laboratory for Computer Science Research (1991). [Cited on page 33]

[13] Albrightson, R., J. J. Garcia-Luna-Aceves, and Joanne Boyle. "EIGRP–A fast routing protocol based on distance vectors." (1994). [Cited on page 33]

[14] Moy, John. "RFC 1131OSPF specification." Network Working Group (1989). [Cited on page 34]

[15] Oran, D. "IETF RFC 1142: OSI IS-IS intra-domain routing protocol." (1990). [Cited on page 34]

[16] Mills, D. L. Exterior Gateway Protocol Formal Specification, DARPA Network Working Group Report RFC-904. Vol. 8. M/A-COM Linkabit, 1984. [Cited on page 34]

[17] Rekhter, Y., T. Li, and S. Hares. "RFC 4271." Internet Engineering Task Force, http://www.rfc-editor.org/rfc/rfc4271.txt, access on 6 (2014). [Cited on page 34]

[18] Campbell, Andrew T., et al. "Open signaling for ATM, internet and mobile networks (OPENSIG'98)." ACM SIGCOMM Computer Communication Review 29.1 (1999): 97-108. [Cited on pages 35 and 186]

[19] Doria, A., et al. "RFC 3292: General Switch Management Protocol (GSMP) V3." (2002). [Cited on page 35]

[20] Tennenhouse, David L., and David J. Wetherall. "Towards an active network architecture." DARPA Active Networks Conference and Exposition, 2002. Proceedings. IEEE, 2002. [Cited on page 35]

[21] Moore, Jonathan T., and Scott M. Nettles. "Towards practical programmable packets." Proceedings of the 20th Conference on Computer Communications (INFOCOM). Citeseer. 2001. [Cited on page 36]

[22] Greenberg, Albert, et al. "A clean slate 4D approach to network control and management." ACM SIGCOMM Computer Communication Review 35.5 (2005): 41-54. [Cited on page 36]

[23] Rexford, Jennifer, et al. "Network-wide decision making: Toward a wafer-thin control plane." Proc. HotNets. 2004. [Cited on page 36]

[24] Caesar, Matthew, et al. "Design and implementation of a routing control platform." Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005. [Cited on page 36]

[25] Enns, R. "Bjorklund, M." and A. Bierman, Ed.," Network Configuration Protocol (NETCONF)", RFC 6241 (2011). [Cited on page 36]

[26] Case, Jeffrey D., et al. Simple network management protocol (SNMP). No. RFC 1157. 1990. [Cited on page 36]

[27] Casado, Martin, et al. "Ethane: Taking control of the enterprise." ACM SIGCOMM Computer Communication Review. Vol. 37. No. 4. ACM, 2007. [Cited on page 36]

[28] McKeown, Nick, et al. "OpenFlow: enabling innovation in campus networks." ACM SIGCOMM Computer Communication Review 38.2 (2008): 69-74. [Cited on page 37]

[29] Alvizu, Rodolfo, and Guido Maier. "Can open flow make transport networks smarter and dynamic? An overview on transport SDN." Smart Communications in Network Technologies (SaCoNeT), 2014 International Conference on. IEEE, 2014. [Cited on page 37]

[30] Doria, Avri, et al. Forwarding and control element separation (ForCES) protocol specification. No. RFC 5810. 2010. [Cited on page 38]

[31] Open Networking Foundation. "Software-defined networking: The new norm for networks." ONF White Paper 2 (2012): 2-6. [Cited on page 38]

[32] Sherwood, Rob, et al. "Carving research slices out of your production networks with OpenFlow." ACM SIGCOMM Computer Communication Review 40.1 (2010): 129-130. [Cited on page 38]

[33] Patel, Parveen, et al. "Ananta: Cloud scale load balancing." ACM SIGCOMM Computer Communication Review 43.4 (2013): 207-218. [Cited on page 38]

[34] Jain, Sushant, et al. "B4: Experience with a globally-deployed software defined WAN." ACM SIGCOMM Computer Communication Review. Vol. 43. No. 4. ACM, 2013. [Cited on page 38]

[35] Gude, Natasha, et al. "NOX: towards an operating system for networks." ACM SIGCOMM Computer Communication Review38.3 (2008): 105-110. [Cited on pages 39 and 44]

[36] Ryu: Component-based Software Defined Networking Framework. https://osrg.github.io/ryu/. Accessed: 18/05/2018. [Cited on pages 39 and 44]

[37] OpenDaylight Consortium. http://www.opendaylight.org/. Accessed: 18/05/2018. [Cited on pages 39, 44, and 66]

[38] Floodlight OpenFlow Controller: Open Source Software for Building Software-defined Networks. http://www.projectfloodlight.org/floodlight/. Accessed: 18/05/2015 [Cited on page 39]

[39] Sezer, Sakir, et al. "Are we ready for SDN? Implementation challenges for software-defined networks." IEEE Communications Magazine 51.7 (2013): 36-43. [Cited on pages 40, 42, 50, 51, and 86]

[40] OpenFlow Switch Specification: Version 1.0.0. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf Accessed 18/05/2018 [Cited on pages 40, 41, and 42]

[41] Dely, Peter, Andreas Kassler, and Nico Bayer. "OpenFlow for wireless mesh networks." Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on. IEEE, 2011. [Cited on page 40]

[42] Petry, Tobias, Rafael da Fonte Lopes da Silva, and Marinho P. Barcellos. "Off the wire control: Improving the control plane resilience through cellular networks." Communications (ICC), 2015 IEEE International Conference on. IEEE, 2015. [Cited on page 40]

[43] Zhang, Peng, et al. "On denial of service attacks in software defined networks." IEEE Network 30.6 (2016): 28-33. [Cited on pages 41, 51, 53, 54, 58, 135, and 137]

[44] Banerjee, Subhasis, and Kalapriya Kannan. "Tag-in-tag: Efficient flow table management in sdn switches." Network and Service Management (CNSM), 2014 10th International Conference on. IEEE, 2014. [Cited on page 42]

[45] OpenFlow Switch Specification: Version 1.1.0. https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf Accessed 18/08/2019 [Cited on page 43]

[46] OpenFlow Switch Specification: Version 1.2.0. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf Accessed 28/10/2018 [Cited on pages 43 and 118]

[47] OpenFlow Switch Specification: Version 1.3.0. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf Accessed 18/05/2018 [Cited on page 43]

[48] The Benefits of Multiple Flow Tables and TTPs https://www.opennetworking.org/wp-content/uploads/2014/10/TR_Multiple_Flow_Tables_and_TTPs.pdf Accessed 18/05/2018 [Cited on page 44]

[49] Erickson, David. "The beacon OpenFlow controller." Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013. [Cited on page 44]

[50] OpenFlow Switch Specification: Version 1.4.0. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf Accessed 18/05/2018 [Cited on page 44]

[51] Cohen, Rami, et al. "On the effect of forwarding table size on SDN network utilization." INFOCOM, 2014 Proceedings IEEE. IEEE, 2014. [Cited on page 44]

[52] Pica8, Inc. https://www.pica8.com/ Accessed 18/05/2018 [Cited on page 45]

[53] OpenFlow Switch Specification: Version 1.4.0. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf Accessed 18/08/2019 [Cited on page 45]

[54] Bianchi, Giuseppe, et al. "OpenState: programming platform-independent stateful openflow applications inside the switch." ACM SIGCOMM Computer Communication Review 44.2 (2014): 44-51. [Cited on page 46]

[55] Zhu, Shuyong, et al. "Sdpa: Enhancing stateful forwarding for software-defined networking." 2015 IEEE 23rd International Conference on Network Protocols (ICNP). IEEE, 2015. [Cited on page 46]

[56] Moshref, Masoud, et al. "Flow-level state transition as a new switch primitive for SDN." Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014. [Cited on page 46]

[57] Bosshart, Pat, et al. "P4: Programming protocol-independent packet processors." ACM SIGCOMM Computer Communication Review 44.3 (2014): 87-95. [Cited on page 46]

[58] Sivaraman, Anirudh, et al. "Packet transactions: High-level programming for line-rate switches." Proceedings of the 2016 ACM SIGCOMM Conference. ACM, 2016. [Cited on page 46]

[59] Carl, Glenn, et al. "Denial-of-service attack-detection techniques." IEEE Internet computing 10.1 (2006): 82-89. [Cited on page 47]

[60] Garber, Lee. "Denial-of-service attacks rip the Internet." Computer 33.4 (2000): 12-17. [Cited on page 47]

[61] Moore, David, et al. "Inferring internet denial-of-service activity." ACM Transactions on Computer Systems (TOCS)24.2 (2006): 115-139. [Cited on page 47]

[62] Wang, Haining, Danlu Zhang, and Kang G. Shin. "Detecting SYN flooding attacks." INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. Vol. 3. IEEE, 2002. [Cited on page 47]

[63] Studer, Ahren, and Adrian Perrig. "The coremelt attack." European Symposium on Research in Computer Security. Springer, Berlin, Heidelberg, 2009. [Cited on page 47]

[64] Kang, Min Suk, Soo Bum Lee, and Virgil D. Gligor. "The crossfire attack." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013. [Cited on page 47]

[65] Zargar, Saman Taghavi, James Joshi, and David Tipper. "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks." IEEE communications surveys & tutorials 15.4 (2013): 2046-2069. [Cited on page 48]

[66] Kumar, Sanjeev. "Smurf-based distributed denial of service (ddos) attack amplification in internet." Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on. IEEE, 2007. [Cited on page 48]

[67] Du, Ping, and Shunji Abe. "Detecting DoS attacks using packet size distribution." Bio-Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd. IEEE, 2007. [Cited on page 48]

[68] Leiwo, Jussipekka, and Yuliang Zheng. "A method to implement a denial of service protection base." Australasian Conference on Information Security and Privacy. Springer, Berlin, Heidelberg, 1997. [Cited on page 48]

[69] Spatscheck, Oliver, and Larry L. Peterson. "Defending against denial of service attacks in Scout." OSDI. Vol. 99. 1999. [Cited on page 48]

[70] Mahajan, Ratul, et al. "Controlling high bandwidth aggregates in the network." ACM SIGCOMM Computer Communication Review 32.3 (2002): 62-73. [Cited on pages 48 and 49]

[71] Gil, Thomer M., and Massimiliano Poletto. "MULTOPS: A Data-Structure for Bandwidth Attack Detection." USENIX Security Symposium. 2001. [Cited on page 48]

[72] Barford, Paul, et al. "A signal analysis of network traffic anomalies." Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment. ACM, 2002. [Cited on page 48]

[73] Andersen, David, et al. "Resilient overlay networks." ACM SIGCOMM Computer Communication Review 32.1 (2002): 66-66. [Cited on page 48]

[74] Yan, Jianxin, Stephen Early, and Ross Anderson. "The xenoservice-a distributed defeat for distributed denial of service." Proceedings of ISW. Vol. 2000. sn, 2000. [Cited on page 48]

[75] Senie, D., and P. Ferguson. "Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing." Network (1998). [Cited on page 48]

[76] Macia-Fernandez, Gabriel, Rafael A. Rodriguez-Gomez, and Jess E. Daz-Verdejo. "Defense techniques for low-rate DoS attacks against application servers." Computer Networks54.15 (2010): 2711-2727. [Cited on page 49]

[77] Ioannidis, John, and Steven M. Bellovin. "Pushback: Router-based defense against ddos attacks." (2001). [Cited on page 49]

[78] The Reverse Firewall: Defeating DDOS Attacks Emanating from a Local Area Network http://www.cs3-inc.com/pubs/Reverse_FireWall.pdf Accessed 18/05/2018 [Cited on page 49]

[79] Mirkovic, Jelena, Gregory Prier, and Peter Reiher. "Attacking DDoS at the source." Network Protocols, 2002. Proceedings. 10th IEEE International Conference on. IEEE, 2002. [Cited on page 49]

[80] Shirali-Shahreza, Sajad, and Yashar Ganjali. "Efficient implementation of security applications in OpenFlow controller with flexam." High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on. IEEE, 2013. [Cited on page 49]

[81] Yu, Minlan, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch." NSDI. Vol. 13. 2013. [Cited on page 49]

[82] Chin, Tommy, et al. "Selective packet inspection to detect DoS flooding using software defined networking (SDN)." Distributed Computing Systems Workshops (ICDCSW), 2015 IEEE 35th International Conference on. IEEE, 2015. [Cited on page 49]

[83] Braga, Rodrigo, Edjard Mota, and Alexandre Passito. "Lightweight DDoS flooding attack detection using NOX/OpenFlow." Local Computer Networks (LCN), 2010 IEEE 35th Conference on. IEEE, 2010. [Cited on page 49]

[84] Giotis, Kostas, et al. "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments." Computer Networks 62 (2014): 122-136. [Cited on page 49]

[85] Gkounis, Dimitrios. "Cross-domain DoS link-flooding attack detection and mitigation using SDN principles." MSc Th., ETH(2014). [Cited on page 49]

[86] Scott-Hayward, Sandra, Sriram Natarajan, and Sakir Sezer. "A survey of security in software defined networks." IEEE Communications Surveys & Tutorials 18.1 (2016): 623-654. [Cited on page 49]

[87] Benton, Kevin, L. Jean Camp, and Chris Small. "OpenFlow vulnerability assessment." Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013. [Cited on pages 50 and 51]

[88] Kreutz, Diego, Fernando Ramos, and Paulo Verissimo. "Towards secure and dependable software-defined networks." Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013. [Cited on pages 50, 51, and 114]

[89] Kloti, Rowan, Vasileios Kotronis, and Paul Smith. "OpenFlow: A security analysis." Network Protocols (ICNP), 2013 21st IEEE International Conference on. IEEE, 2013. [Cited on pages 50, 51, and 113]

[90] Pascoal, Tlio A., et al. "Slow TCAM Exhaustion DDoS Attack." IFIP International Conference on ICT Systems Security and Privacy Protection. Springer, Cham, 2017. [Cited on pages 52 and 55]

[91] Shin, Seungwon, and Guofei Gu. "Attacking software-defined networks: A first feasibility study." Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013. [Cited on page 52]

[92] Cao, Jiahao, et al. "Disrupting SDN via the data plane: a low-rate flow table overflow attack." International Conference on Security and Privacy in Communication Systems. Springer, Cham, 2017. [Cited on pages 52, 54, and 80]

[93] Sonchack, John, Adam J. Aviv, and Eric Keller. "Timing SDN control planes to infer network configurations." Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization. ACM, 2016. [Cited on pages 52 and 80]

[94] Leng, Junyuan, et al. "An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network." arXiv preprint arXiv:1504.03095 (2015). [Cited on pages 52 and 80]

[95] Dhawan, Mohan, et al. "SPHINX: Detecting Security Attacks in Software-Defined Networks." NDSS. 2015. [Cited on pages 52, 70, and 137]

[96] Ambrosin, Moreno, et al. "Amplified Distributed Denial of Service Attack in Software Defined Networking." New Technologies, Mobility and Security (NTMS), 2016 8th IFIP International Conference on. IEEE, 2016. [Cited on pages 53 and 86]

[97] Alharbi, Talal, Siamak Layeghy, and Marius Portmann. "Experimental evaluation of the impact of DoS attacks in SDN." Telecommunication Networks and Applications Conference (ITNAC), 2017 27th International. IEEE, 2017. [Cited on pages 53, 54, 86, and 137]

[98] Kandoi, Rajat, and Markku Antikainen. "Denial-of-service attacks in OpenFlow SDN networks." Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on. IEEE, 2015. [Cited on pages 53 and 55]

[99] Shang, Gao, et al. "FloodDefender: protecting data and control plane resources under SDN-aimed DoS attacks." INFOCOM 2017-IEEE Conference on Computer Communications, IEEE. IEEE, 2017. [Cited on pages 53, 56, 57, 113, 130, 134, 167, and 183]

[100] Wang, An, et al. "Scotch: Elastically scaling up sdn control-plane using vswitch based overlay." Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. ACM, 2014. [Cited on pages 53, 76, 82, and 83]

[101] Lee, Ming-Chieh, and Jang-Ping Sheu. "An efficient routing algorithm based on segment routing in software-defined networking." Computer Networks 103 (2016): 44-55. [Cited on page 54]

[102] Tri, Hiep T. Nguyen, and Kyungbaek Kim. "Assessing the impact of resource attack in software defined network." Information Networking (ICOIN), 2015 International Conference on. IEEE, 2015. [Cited on page 54]

[103] Qian, Ying, Wanqing You, and Kai Qian. "OpenFlow flow table overflow attacks and countermeasures." Networks and Communications (EuCNC), 2016 European Conference on. IEEE, 2016. [Cited on page 54]

[104] Mousavi, Seyed Mohammad, and Marc St-Hilaire. "Early detection of DDoS attacks against SDN controllers." Computing, Networking and Communications (ICNC), 2015 International Conference on. IEEE, 2015. [Cited on pages 55, 129, 133, and 185]

[105] Wang, Mingxin, et al. "An Approach for Protecting the OpenFlow Switch from the Saturation Attack." 4th National Conference on Electrical, Electronics and Computer Engineering (NCEECE 2015). 2016. [Cited on page 56]

[106] Kokila, R. T., S. Thamarai Selvi, and Kannan Govindarajan. "DDoS detection and analysis in SDN-based environment using support vector machine classifier." Advanced Computing (ICoAC), 2014 Sixth International Conference on. IEEE, 2014. [Cited on pages 56, 129, 132, and 183]

[107] Phan, Trung V., Nguyen Khac Bao, and Minho Park. "A novel hybrid flow-based handler with DDoS attacks in software-defined networking." Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences. IEEE, 2016. [Cited on pages 56 and 130]

[108] Cui, Yunhe, et al. "SD-Anti-DDoS: Fast and efficient DDoS defense in software-defined networks." Journal of Network and Computer Applications 68 (2016): 65-79. [Cited on pages 57 and 131]

[109] Gao, Shang, et al. "Security Threats in the Data Plane of Software-Defined Networks." IEEE Network (2018). [Cited on page 57]

[110] Dao, Nhu-Ngoc, et al. "A feasible method to combat against DDoS attack in SDN network." Information Networking (ICOIN), 2015 International Conference on. IEEE, 2015. [Cited on page 58]

[111] Wang, Haopei, Lei Xu, and Guofei Gu. "Floodguard: A dos attack prevention extension in software-defined networks." Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on. IEEE, 2015. [Cited on pages 58, 59, and 133]

[112] Wei, Lei, and Carol Fung. "FlowRanger: A request prioritizing algorithm for controller DoS attacks in Software Defined Networks." Communications (ICC), 2015 IEEE International Conference on. IEEE, 2015. [Cited on pages 58, 133, 135, and 168]

[113] Wolf, Tilman, and Jingrui Li. "Denial-of-Service Prevention for Software-Defined Network Controllers." Computer Communication and Networks (ICCCN), 2016 25th International Conference on. IEEE, 2016. [Cited on page 59]

[114] Yuan, Bin, et al. "Defending against flow table overloading attack in software-defined networks." IEEE Transactions on Services Computing (2016). [Cited on page 59]

[115] Bahaa-Eldin, Ayman M., Ebada Essam-Eldin ElDessouky, and Hasan Da. "Protecting OpenFlow switches against denial of service attacks." Computer Engineering and Systems (ICCES), 2017 12th International Conference on. IEEE, 2017. [Cited on page 60]

[116] Xu, Tong, et al. "Mitigating the Table-Overflow Attack in Software-Defined Networking." IEEE Transactions on Network and Service Management 14.4 (2017): 1086-1097. [Cited on page 60]

[117] Shin, Seungwon, et al. "Avant-guard: Scalable and vigilant switch flow management in software-defined networks." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013. [Cited on pages 61 and 185]

[118] Ambrosin, Moreno, et al. "Lineswitch: Efficiently managing switch flow in software-defined networking while effectively tackling dos attacks." Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. ACM, 2015. [Cited on pages 61, 110, and 133]

[119] Atli, A. Volkan, et al. "Protecting SDN controller with per-flow buffering inside OpenFlow switches." Black Sea Conference on Communications and Networking (BlackSeaCom), 2017 IEEE International. IEEE, 2017. [Cited on pages 62 and 132]

[120] Kotani, Daisuke, and Yasuo Okabe. "A packet-in message filtering mechanism for protection of control plane in OpenFlow networks." Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems. ACM, 2014. [Cited on page 62]

[121] Huang, Xueli, Xiaojiang Du, and Bin Song. "An effective DDoS defense scheme for SDN." Communications (ICC), 2017 IEEE International Conference on. IEEE, 2017. [Cited on pages 62, 63, and 133]

[122] Boite, Julien, et al. "Statesec: Stateful monitoring for DDoS protection in software defined networks." Network Softwarization (NetSoft), 2017 IEEE Conference on. IEEE, 2017. [Cited on pages 63 and 185]

[123] Wang, Rui, Zhiping Jia, and Lei Ju. "An entropy-based distributed DDoS detection mechanism in software-defined networking." Trustcom/BigDataSE/ISPA, 2015 IEEE. Vol. 1. IEEE, 2015. [Cited on page 63]

[124] Kalkan, Kbra, Grkan Gr, and Fatih Alagz. "SDNScore: A statistical defense mechanism against DDoS attacks in SDN environment." Computers and Communications (ISCC), 2017 IEEE Symposium on. IEEE, 2017. [Cited on pages 63 and 132]

[125] Park, Taejune, Yeonkeun Kim, and Seungwon Shin. "UNISAFE: A union of security actions for software switches." Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization. ACM, 2016. [Cited on page 64]

[126] Blial, Othmane, Mouad Ben Mamoun, and Redouane Benaini. "An overview on SDN architectures with multiple controllers." Journal of Computer Networks and Communications 2016 (2016). [Cited on pages 65 and 115]

[127] Abdelaziz, Ahmed, et al. "Distributed controller clustering in software defined networks." PloS one 12.4 (2017): e0174715. [Cited on pages 65 and 115]

[128] Fonseca, Paulo, et al. "A replication component for resilient OpenFlow-based networking." Network Operations and Management Symposium (NOMS), 2012 IEEE. IEEE, 2012. [Cited on page 65]

[129] Hassas Yeganeh, Soheil, and Yashar Ganjali. "Kandoo: a framework for efficient and scalable offloading of control applications." Proceedings of the first workshop on Hot topics in software defined networks. ACM, 2012. [Cited on page 65]

[130] Tootoonchian, Amin, and Yashar Ganjali. "Hyperflow: A distributed control plane for OpenFlow." Proceedings of the 2010 internet network management conference on Research on enterprise networking. 2010. [Cited on pages 65 and 116]

[131] Berde, Pankaj, et al. "ONOS: towards an open, distributed SDN OS." Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014. [Cited on page 66]

[132] Koponen, Teemu, et al. "Onix: A distributed control platform for large-scale production networks." OSDI. Vol. 10. 2010. [Cited on pages 66 and 116]

[133] Ma, Duohe, Zhen Xu, and Dongdai Lin. "Defending blind DDoS attack on SDN based on moving target defense." International Conference on Security and Privacy in Communication Systems. Springer, Cham, 2014. [Cited on pages 66 and 67]

[134] Dixit, Advait, et al. "Towards an elastic distributed SDN controller." ACM SIGCOMM Computer Communication Review. Vol. 43. No. 4. ACM, 2013. [Cited on pages 66, 117, and 118]

[135] Yazici, Volkan, M. Oguz Sunay, and Ali O. Ercan. "Controlling a software-defined network via distributed controllers." arXiv preprint arXiv:1401.7651 (2014). [Cited on pages 66 and 67]

[136] Kaur, Sukhveer, Japinder Singh, and Navtej Singh Ghumman. "Network programmability using POX controller." ICCCS International Conference on Communication, Computing & Systems, IEEE. Vol. 138. 2014. [Cited on page 70]

[137] Tavakoli, Arsalan, et al. "Applying NOX to the Datacenter." HotNets. 2009. [Cited on page 73]

[138] Dugan Jon et al. 2018 iperf. [online] iperf.fr. Available at: `https://iperf.fr/` [Accessed 14 Oct. 2018]. [Cited on pages 76 and 167]

[139] Laissaoui, C., et al. "A measurement of the response times of various OpenFlow/SDN controllers with CBench." Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of. IEEE, 2015. [Cited on pages 78 and 151]

[140] Salman, Ola, et al. "SDN controllers: a comparative study." Electrotechnical Conference (MELECON), 2016 18th Mediterranean. IEEE, 2016. [Cited on pages 78 and 151]

[141] Mori, Tatsuya, et al. "Identifying elephant flows through periodically sampled packets." Proceedings of the 4th ACM SIGCOMM conference on Internet measurement. ACM, 2004. [Cited on page 80]

[142] Biondi, Philippe et al. 2018 Scapy. [online] Scapy.net. Available at: `https://scapy.net/` [Accessed 14 Oct. 2018]. [Cited on page 80]

[143] Benson, Theophilus, Aditya Akella, and David A. Maltz. "Network traffic characteristics of data centers in the wild." Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, 2010. [Cited on page 82]

[144] Tootoonchian, Amin, et al. "On Controller Performance in Software-Defined Networks." Hot-ICE 12 (2012): 1-6. [Cited on page 82]

[145] Sleator, Daniel D., and Robert E. Tarjan. "Amortized efficiency of list update and paging rules." Communications of the ACM 28.2 (1985): 202-208. [Cited on page 83]

[146] Al-Zoubi, Hussein, Aleksandar Milenkovic, and Milena Milenkovic. "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite." Proceedings of the 42nd annual Southeast regional conference. ACM, 2004. [Cited on page 83]

[147] Jaleel, Aamer, et al. "High performance cache replacement using re-reference interval prediction (RRIP)." ACM SIGARCH Computer Architecture News. Vol. 38. No. 3. ACM, 2010. [Cited on pages 83 and 182]

[148] Katta, Naga, et al. "Cacheflow: Dependency-aware rule-caching for software-defined networks." Proceedings of the Symposium on SDN Research. ACM, 2016. [Cited on page 84]

[149] netsniff-ng toolkit. [online] netsniff-ng.org. Available at: http://netsniff-ng.org/ [Accessed 14 Oct. 2018]. [Cited on page 85]

[150] "The CAIDA UCSD Anonymized Internet Traces - [20th March 2014]", http://www.caida.org/data/passive/passive_dataset.xml Accessed 24/10/2018 [Cited on pages 87, 136, 137, 138, and 142]

[151] Curtis, Andrew R., et al. "DevoFlow: scaling flow management for high-performance networks." ACM SIGCOMM Computer Communication Review. Vol. 41. No. 4. ACM, 2011. [Cited on page 89]

[152] Stephens, Brent, et al. "PAST: Scalable Ethernet for data centers." Proceedings of the 8th international conference on Emerging networking experiments and technologies. ACM, 2012. [Cited on page 89]

[153] Dell Inc. "Work Flexibility." Dell Careers, jobs.dell.com/work-flexibility. Accessed on 12/11/2018 [Cited on page 104]

[154] "VOIP Services Market: Global Industry Analysis and Opportunity Assessment 2015 - 2025",Future Market Insights, https://www.futuremarketinsights.com/

reports/global-voip-services-market. Accessed on 12/11/2018 [Cited on page 104]

[155] Muralidharan,Baktha and Kumar, Anoop, Video Quality of Service (QOS) Tutorial, Cisco Limited, September 18, 2017, https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-video/212134-Video-Quality-of-Service-QOS-Tutorial.html. Accessed on 12/11/2018 [Cited on page 104]

[156] Cisco, Visual Networking Index. "The zettabyte era: Trends and analysis." Updated (07/06/2017), https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html. Accessed on 12/11/2018 [Cited on page 104]

[157] Westcott, Kevin, et al. Deloitte Insights, "Digital media trends survey A new world of choice for digital consumers". Updated (19/03/2018), https://www2.deloitte.com/insights/us/en/industry/technology/digital-media-trends-consumption-habits-survey.html. Accessed on 12/11/2018 [Cited on page 104]

[158] Netflix, Inc. Netflix INVESTORS, "2017 Quarterly Earnings", Updated (22/01/2018), https://ir.netflix.com/financials/quarterly-earnings/default.aspx. Accessed on 12/11/2018 [Cited on page 105]

[159] Wu, Wei, et al. "Improving data center energy efficiency using a cyber-physical systems approach: integration of building information modeling and wireless sensor networks." Procedia engineering 118 (2015): 1266-1273. [Cited on page 106]

[160] Schmidt, Mischa, et al. "Cyber-Physical System for Energy-Efficient Stadium Operation: Methodology and Experimental Validation." ACM Transactions on Cyber-Physical Systems 2.4 (2018): 25. [Cited on page 106]

[161] Dogaru, Delia Ioana, and Ioan Dumitrache. "Cyber-physical systems in healthcare networks." E-Health and Bioengineering Conference (EHB), 2015. IEEE, 2015. [Cited on page 106]

[162] Min, Dugki. "Medical cyber physical systems and bigdata platforms." (2013). [Cited on page 106]

[163] Giordano, Andrea, et al. "A cyber-physical system for distributed real-time control of urban drainage networks in smart cities." International Conference on Internet and Distributed Computing Systems. Springer, Cham, 2014. [Cited on page 106]

[164] Xia, Feng, et al. "Network QoS management in cyber-physical systems." Embedded Software and Systems Symposia, 2008. ICESS Symposia'08. International Conference on. IEEE, 2008. [Cited on pages 106 and 107]

[165] Mostafaei, Habib, and Michael Menth. "Software-defined wireless sensor networks: A survey." Journal of Network and Computer Applications (2018). [Cited on page 106]

[166] Shin, Seungwon, et al. "Avant-guard: Scalable and vigilant switch flow management in software-defined networks." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013. [Cited on pages 110 and 133]

[167] Medved, Jan, et al. "Opendaylight: Towards a model-driven sdn controller architecture." 2014 IEEE 15th International Symposium on. IEEE, 2014. [Cited on page 116]

[168] Ma, Yi-Wei, et al. "Load-balancing multiple controllers mechanism for software-defined networking." Wireless Personal Communications 94.4 (2017): 3549-3574. [Cited on page 116]

[169] Hu, Yannan, et al. "Balanceflow: controller load balancing for OpenFlow networks." Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on. Vol. 2. IEEE, 2012. [Cited on page 116]

[170] Zhou, Yuanhao, et al. "A load balancing strategy of sdn controller based on distributed decision." Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on. IEEE, 2014. [Cited on page 117]

[171] Yu, Jinke, et al. "A load balancing mechanism for multiple SDN controllers based on load informing strategy." Network Operations and Management Symposium (APNOMS), 2016 18th Asia-Pacific. IEEE, 2016. [Cited on page 117]

[172] Filali, Abderrahime, et al. "SDN Controller Assignment and Load Balancing with Minimum Quota of Processing Capacity." 2018 IEEE International Conference on Communications (ICC). IEEE, 2018. [Cited on page 117]

[173] Wang, Ping, et al. "An efficient flow control approach for SDN-based network threat detection and migration using support vector machine." 2016 IEEE 13th International Conference on e-Business Engineering (ICEBE). IEEE, 2016. [Cited on page 129]

[174] Barki, Lohit, et al. ”Detection of distributed denial of service attacks in software defined networks.” 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2016. [Cited on page 130]

[175] Alshamrani, Adel, et al. ”A defense system for defeating DDoS attacks in SDN based networks.” Proceedings of the 15th ACM International Symposium on Mobility Management and Wireless Access. ACM, 2017. [Cited on page 130]

[176] Report: State of the Web. https://httparchive.org/reports/state-of-the-web Accessed 28/10/2018 [Cited on page 132]

[177] Breiman, Leo. ”Random forests.” Machine learning 45.1 (2001): 5-32. [Cited on page 135]

[178] Bosch, Anna, Andrew Zisserman, and Xavier Munoz. ”Image classification using random forests and ferns.” Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on. IEEE, 2007. [Cited on page 135]

[179] Kruppa, Jochen, et al. ”Consumer credit risk: Individual probability estimates using machine learning.” Expert Systems with Applications 40.13 (2013): 5125-5131. [Cited on page 135]

[180] Qi, Yanjun. ”Random forest for bioinformatics.” Ensemble machine learning. Springer, Boston, MA, 2012. 307-323. [Cited on page 135]

[181] “Data Set for IMC 2010 Data Center Measurement” http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html Accessed 24/10/2018 [Cited on pages 136, 137, 138, and 142]

[182] “LBNL/ICSI Enterprise Tracing Project” http://www.icir.org/enterprise-tracing/ Accessed 24/10/2018 [Cited on pages 136, 137, 138, 142, 161, 162, and 163]

[183] “Netresec: Capture files from 4SICS Geek Lounge” https://www.netresec.com/?page=PCAP4SICS Accessed 24/10/2018 [Cited on pages 136, 137, 138, 142, 162, and 163]

[184] “CRATE datasets,” ftp://download.iwlab.foi.se/dataset/smia2011/Network_traffic/ Accessed 24/10/2018 [Cited on pages 136, 137, 138, 142, 162, and 163]

[185] Wright, M. N. & Ziegler, A. (2017). ranger: A fast implementation of random forests for high dimensional data in C++ and R. Journal of Statistical Software 77:1-17 [Cited on page 143]

[186] Suznjevic, M., and J. Saldana. "Delay limits for real-time services." draft-suznjevic-dispatch-delay-limits-00 (work in progress) (2015). [Cited on page 155]

[187] Jiang, Song, and Xiaodong Zhang. "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance." ACM SIGMETRICS Performance Evaluation Review 30.1 (2002): 31-42. [Cited on page 182]

[188] Megiddo, Nimrod, and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." FAST. Vol. 3. No. 2003. 2003. [Cited on page 182]

[189] Bilal, Muhammad, and Shin-Gak Kang. "Time aware least recent used (TLRU) cache management policy in ICN." Advanced Communication Technology (ICACT), 2014 16th International Conference on. IEEE, 2014. [Cited on page 182]

Appendix

A Results of varying the attack parameters

A.1 Results of Varying the number of flushes in the Spray Attack

Flushes/sec	Total No of Removals	Removals/sec	Throughput Mb/s	TableMisses
100	5944	99	888	30067
200	11825	197	751	75127
300	17670	294	722	93646
400	23391	389	653	117763
500	27278	454	603	139044
750	28070	467	587	143867
1000	28781	479	583	145917

Table 1a: Spray Attack Power under **FIFO** eviction

Flushes/sec	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	1177	19	929	20551
200	1733	28	928	12300
300	2144	35	925	16066
400	2578	42	921	16846
500	2856	47	925	14254
750	3053	50	922	15306
1000	2975	49	922	18624

Table 1b: Spray Attack Power under **Random** eviction

Flushes/sec	Total No of Removals	Removals/sec	Throughput Mb/s	TableMisses
100	2	0	940	408
200	2	0	940	408
300	2	0	940	408
400	2	0	940	408
500	2	0	940	408
750	2	0	940	408
1000	2	0	940	408

Table 1c: Spray Attack Power under **LFU** eviction

Flushes/sec	Total No of Removals	Removals/sec	Throughput Mb/s	TableMisses
100	7	0	940	618
200	7	0	940	618
300	7	0	940	618
400	7	0	940	618
500	7	0	940	618
750	7	0	940	618
1000	7	0	940	618

Table 1d: Spray Attack Power under **LRU** eviction

A.2 Results of Varying the number of malicious flow rules in the Clog Attack

No of Rules clogged	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
495	7820	130	844	47445
496	8242	137	839	49660
497	8757	145	814	60087
498	9264	154	806	62047
499	10218	170	787	68501
500	11003	183	777	73674

Table 2a: Clog Attack Power under **FIFO** eviction

No of Rules clogged	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
495	7563	126	853	44608
496	8039	133	832	51236
497	8502	141	834	52972
498	8889	148	816	58220
499	9768	162	804	63163
500	10347	172	795	66445

Table 2b: Clog Attack Power under **Random** eviction

No of Rules clogged	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
495	236	3	931	9786
496	505	8	928	13888
497	1680	28	925	15475
498	5299	88	898	26873
499	16412	273	735	89582
500	27930	465	568	151499

Table 2c: Clog Attack Power under **LFU** eviction

No of Rules clogged	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
495	33	0	939	3397
496	68	1	934	7095
497	112	1	932	8792
498	153	2	929	13455
499	194	3	930	10431
500	207	3	927	13646

Table 2d: Clog Attack Power under **LRU** eviction

Packets/Sec	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
1000pps	17960	299	694	93541
2000pps	19832	330	675	99677
3000pps	21738	362	654	107730
4000pps	23585	393	621	106344
5000pps	25415	423	606	116050

Table 2e: Clog Attack Power under **LFU** eviction with last rule packet rates varied

B Results of varying the number of attackers

B.1 Results of Varying the number of Attackers in the Spray Attack

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
1	17673	294	722	93646
2	34004	566	472	177770
3	49159	819	300	244551
4	63905	1065	241	271587
5	78385	1306	214	285416

Table 3a: Increasing Spray Attackers under **FIFO** eviction

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
1	2135	35	927	12711
2	27239	453	616	134947
3	46453	774	324	238781
4	62661	1044	240	272124
5	77712	1295	224	280969

Table 3b: Increasing Spray Attackers under **Random** eviction

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
1	2	0	940	408
2	2	0	940	408
3	2	0	940	408
4	2	0	940	408
5	2	0	940	408

Table 3c: Increasing Spray Attackers under **LFU** eviction

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
1	7	0	940	618
2	7	0	940	618
3	9	0	940	864
4	14	0	940	2376
5	36	0	939	2866

Table 3d: Increasing Spray Attackers under **LRU** eviction

B.2 Results of Varying the number of attackers in the Clog Attack

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
5	8860	147	817	58079
6	9575	159	798	65301
7	10218	170	787	68501
8	10851	180	791	69545
9	11337	188	772	75507

Table 4a: Increasing Clog Attackers under **FIFO** eviction

No of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
5	8143	135	830	52516
6	9037	150	818	58422
7	9768	162	804	63163
8	10449	174	787	69281
9	11044	184	773	75637

Table 4b: Increasing Clog Attackers under **Random** eviction

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
5	274	4	930	10026
6	934	15	927	14423
7	16412	273	735	89582
8	136866	2281	177	305796
9	191133	3185	150	315822

Table 4c: Increasing Clog Attackers under **LFU** eviction

No. of Attackers	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
5	56	0	935	6619
6	119	1	931	9117
7	194	3	930	10431
8	287	4	930	9860
9	360	6	929	13220

Table 4d: Increasing Clog Attackers under **LRU** eviction

C Results of varying the flow table size

C.1 Results of Varying the size of the flow table in the Spray Attack

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	17559	292	713	98181
250	17603	293	721	94426
500	17670	294	722	93646
750	34117	568	475	180614
1000	30362	506	574	147025

Table 5a: Spray Attack on Varying Flow Table Sizes: **FIFO** eviction

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	5574	92	894	29619
250	3380	56	916	21228
500	2144	35	925	16066
750	27561	459	587	146533
1000	23840	397	641	124087

Table 5b: Spray Attack on Varying Flow Table Sizes: **Random** eviction

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	3	0	940	408
250	2	0	940	408
500	2	0	940	408
750	2	0	940	408
1000	2	0	940	408

Table 5c: Spray Attack on Varying Flow Table Sizes: **LFU** eviction

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	9	0	940	864
250	7	0	940	618
500	7	0	940	618
750	7	0	940	618
1000	7	0	940	618

Table 5d: Spray Attack on Varying Flow Table Sizes: **LRU** eviction

C.2 Results of Varying the size of the flow table in the Clog Attack

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	40294	671	457	186304
250	16132	268	690	94298
500	10218	170	787	68501
750	7757	129	848	45757
1000	6594	109	871	37248

Table 6a: Clog Attack on Varying Flow Table Sizes: **FIFO** eviction

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	28313	471	588	142536
250	14032	233	738	89010
500	9768	162	804	63163
750	8222	137	830	52081
1000	6788	113	865	39046

Table 6b: Clog Attack on Varying Flow Table Sizes: **Random** eviction

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	28469	474	572	148538
250	26438	440	743	87839
500	16412	273	735	89582
750	5016	83	902	26391
1000	3237	53	919	19879

Table 6c: Clog Attack on Varying Flow Table Sizes: **LFU** eviction

Flow Table Size	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
100	2280	38	927	12363
250	264	4	925	13179
500	194	3	930	10431
750	150	2	928	12931
1000	138	2	928	13197

Table 6d: Clog Attack on Varying Flow Table Sizes: **LRU** eviction

D Results of varying the surrounding network traffic during the attack

D.1 Results of Varying the surrounding network traffic in the Spray Attack

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	17667	294	716	95975
Caida_50	17670	294	722	93646
Caida_100	17670	294	722	93646
Caida_200	18171	302	719	94931
Caida_500	19601	326	699	102853
Caida_750	20300	338	691	104907

Table 7a: Spray Attack with varying BG traffic: **FIFO** eviction

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	1527	25	926	15053
Caida_50	2144	35	925	16066
Caida_100	3095	51	919	18796
Caida_200	5102	85	901	25883
Caida_500	8366	139	828	52922
Caida_750	9911	165	795	66852

Table 7b: Spray Attack with varying BG traffic: **Random** eviction

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	2	0	940	408
Caida_50	2	0	940	408
Caida_100	2	0	940	408
Caida_200	2	0	940	408
Caida_500	2	0	940	408
Caida_750	2	0	940	408

Table 7c: Spray Attack with varying BG traffic: **LFU** eviction

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	7	0	940	618
Caida_50	7	0	940	618
Caida_100	7	0	940	618
Caida_200	7	0	940	618
Caida_500	7	0	940	618
Caida_750	7	0	940	618

Table 7d: Spray Attack with varying BG traffic: **LRU** eviction

D.2 Results of Varying the surrounding network traffic in the Clog Attack

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	7931	132	833	50902
Caida_50	10218	170	787	68501
Caida_100	15020	250	739	88769
Caida_200	23591	393	656	118106
Caida_500	36405	606	470	186568
Caida_750	41755	695	387	211983

Table 8a: Clog Attack with varying BG traffic: **FIFO** eviction

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	7499	124	853	44436
Caida_50	9768	162	804	63163
Caida_100	14228	237	742	88470
Caida_200	22540	375	659	116062
Caida_500	35089	584	499	176730
Caida_750	40527	675	428	199172

Table 8b: Clog Attack with varying BG traffic: **Random** eviction

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	15171	252	741	88736
Caida_50	16412	273	735	89582
Caida_100	23475	391	651	120538
Caida_200	237216	620	431	198373
Caida_500	56179	936	257	262102
Caida_750	63804	1063	248	266678

Table 8c: Clog Attack with varying BG traffic: **LFU** eviction

Traffic	Total No of Evictions	Evictions/sec	Throughput Mb/s	TableMisses
Caida_40	190	3	925	13190
Caida_50	194	3	930	10431
Caida_100	219	3	930	10044
Caida_200	254	4	927	13489
Caida_500	300	5	928	13229
Caida_750	325	5	926	14370

Table 8d: Clog Attack with varying BG traffic: **LRU** eviction