THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Security in Software Defined Networks

Talal Alharbi

B.S. (Computer Science),

M.S. (Network and System Administration)

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2018

School of Information Technology & Electrical Engineering

Abstract

Software Defined Networking (SDN) is an emerging computer network paradigm and represents one of the most promising technologies to simplify network management and configuration through increased network programmability and abstraction. In contrast to traditional networks, in SDN, the control plane, which makes decisions on how to forward traffic, is separated from the data plane, which transmits traffic to selected destinations. That makes network control (via the *SDN controller*) more programmable, dynamic and centralised. With the higher level of abstraction that SDN provides, network administrators can more easily configure network services and manage traffic flows without having to configure a large number of individual network devices (switches and routers). The great potential of SDN has led to significant deployments in data centres, wide area networks, etc., and it is growing at a rapid pace.

Security is a critical aspect of networking in general and is particularly vital in SDN. Due to its fundamentally new architecture, SDN presents new potential security vulnerabilities and risks. Security in SDN has not received much attention yet, given that it is very distinct and unique.

The goal of this PhD was to address this gap and analyse the security of the SDN infrastructure, identify vulnerabilities and weaknesses, and propose corresponding solutions and improvements. The focus was on the fundamental aspects and components of SDN, in particular the building blocks of the control plane components include Topology Discovery, Address Resolution Protocol (ARP) Handling and Virtualisation Layer. Finally, the thesis thoroughly explored and investigated the most common and effective attacks against the SDN architecture.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, financial support and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Publications during candidature

Peer-reviewed Papers:

[1] Talal Alharbi, Marius Portmann, and Farzaneh Pakzad, "**The (in)security of Topology Discovery in Software Defined Networks**," in Proc. of the 40th conference on Local Computer Networks (LCN), IEEE, 2015, pp. 502-505.

[2] Talal Alharbi, Dario Durando, Farzaneh Pakzad, and Marius Portmann, "**Securing ARP in Software Defined Networks**," in Proc. of the 41st conference on Local Computer Networks (LCN), IEEE, 2016, pp. 523-526.

[3] Talal Alharbi, and Marius Portmann, "**SProxy ARP - Efficient ARP Handling in SDN**," in Proc. of the 26th International Telecommunication Networks and Applications Conference (ITNAC), IEEE, 2016, pp. 179-184.

[4] Talal Alharbi, Siamak Layeghy, and Marius Portmann, "**Experimental Evaluation of the Impact of DoS Attacks in SDN**," in Proc. of the 27th International Telecommunication Networks and Applications Conference (ITNAC), IEEE, 2017, pp. 1-6.

Publications included in this thesis

Talal Alharbi, Marius Portmann, and Farzaneh Pakzad, "**The (in)security of Topology Discovery in Software Defined Networks**," in Proc. of the 40th conference on Local Computer Networks (LCN), IEEE, 2015, pp. 502-505. - Incorporated as Chapter 4.

Contributor	Statement of contribution
Author Talal Alharbi (Candidate)	Conception and design (70%) Analysis and interpretation (80%) Drafting and production (70%)
Author Marius Portmann	Conception and design (20%) Analysis and interpretation (10%) Drafting and production (20%)
Author Farzaneh Pakzad	Conception and design (10%) Analysis and interpretation (10%) Drafting and production (10%)

Talal Alharbi, Dario Durando, Farzaneh Pakzad, and Marius Portmann, "**Securing ARP in Software Defined Networks**," in Proc. of the 41st conference on Local Computer Networks (LCN), IEEE, 2016, pp. 523-526. - Incorporated as Chapter 5.

Contributor	Statement of contribution
Author Talal Alharbi (Candidate)	Conception and design (70%) Analysis and interpretation (75%) Drafting and production (70%)
Author Dario Durando	Conception and design (10%) Analysis and interpretation (5%) Drafting and production (10%)
Author Farzaneh Pakzad	Conception and design (10%) Analysis and interpretation (5%) Drafting and production (10%)
Author Marius Portmann	Conception and design (10%) Analysis and interpretation (15%) Drafting and production (10%)

Talal Alharbi, and Marius Portmann, "**SProxy ARP - Efficient ARP Handling in SDN**," in Proc. of the 26th International Telecommunication Networks and Applications Conference (ITNAC), IEEE, 2016, pp. 179-184. - Incorporated as Chapter 6.

Contributor	Statement of contribution
Author Talal Alharbi (Candidate)	Conception and design (80%) Analysis and interpretation (80%) Drafting and production (80%)
Author Marius Portmann	Conception and design (20%) Analysis and interpretation (20%) Drafting and production (20%)

Talal Alharbi, Siamak Layeghy, and Marius Portmann, "**Experimental Evaluation of the Impact of DoS Attacks in SDN**," in Proc. of the 27th International Telecommunication Networks and Applications Conference (ITNAC), IEEE, 2017, pp. 1-6. - Incorporated as Chapter 7.

Contributor	Statement of contribution
Author Talal Alharbi (Candidate)	Conception and design (80%) Analysis and interpretation (80%) Drafting and production (70%)
Author Siamak Layeghy	Conception and design (10%) Analysis and interpretation (10%) Drafting and production (10%)
Author Marius Portmann	Conception and design (10%) Analysis and interpretation (10%) Drafting and production (20%)

Contributions by others to the thesis

A/Prof Marius Portmann had input into the core conception and design of the work presented in this thesis, data analysis and interpretation, and critical revision along with providing a constructive feedback and valuable guidance.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Research Involving Human or Animal Subjects

No animal or human subjects were involved in this research.

Acknowledgements

This thesis would have been impossible without the support and guidance of remarkable individuals whom I am profoundly grateful to and wish to acknowledge.

First and foremost, it is a privilege to express my sincere thanks and profound gratitude to my principal advisor, A/Prof. Marius Portmann for his invaluable guidance and constant encouragement throughout the years I spent to complete my PhD. He has been very supportive and helpful from the day one when we started brainstorming my research topic. During my research tenure, he has provided me with an excellent working atmosphere, continuous mentorship and intelligent critiques that significantly enhance my academic writing skills and critical thinking. I have gained a professional and valuable experience from his vast knowledge, scientific insight and superb skills that he shared with us on a daily basis.

I would like to extend my thanks to Prof. Jadwiga Indulska for allocating valuable time out from her overloaded schedule to serve as my associate advisor and her constant advice and academic supervision. I must express my sincere appreciation to Prof. Neil Bergmann for serving as the committee chair and providing constructive feedback at each milestone during my PhD study. I also want to thank the School of ITEE staff for their kind help and assistance. A very special thank you goes out toward my fellow labmates, Mr Siamak Layeghy, Mr Anees Al-najjar, Ms Farzaneh Pakzad and Mr Furqan Khan for the philosophical arguments and the exchange of knowledge and skills during our regular group meeting. My acknowledgements would be incomplete without thanking Majmaah University for the award of the PhD scholarship which enabled me to undertake my PhD at the University of Queensland.

Most of all, I owe a great deal to my parents, Abdi and Sitah who are the primary source of unstinting support, encouragement and love that helped me survive and afford to undertake this endeavour. My gratitude is equally extended to all my siblings for bearing with me and cheering me up with their wishes whenever needed.

Last but not least, I am eternally grateful to my lovely wife for her forbearance, tolerance, and reassurance during the most stressful time of my PhD journey. My most heartfelt gratitude is for my first-born son, Alwaleed and my sweet and beautiful daughter, Rana. They are the fount of my greatest joy and happiness.

Financial support

This research was supported by Majmaah University through the Saudi Arabian Culture Mission in Australia.

Keywords

software defined networking, sdn, topology discovery, network security, security of topology discovery, security of address resolution protocol, denial of service attack, network virtualisation security

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 100503, Computer Communications Networks, 30%

ANZSRC code: 080503, Networking and Communications, 30%

ANZSRC code: 080303, Computer System Security, 40%

Fields of Research (FoR) Classification

FoR code: 0805, Distributed Computing, 70%

FoR code: 1005, Communications Technologies, 30%

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Research Contributions 3

1.2.1 Security of Topology Discovery 3

1.2.2 Security of ARP 4

1.2.3 Efficient ARP Handling 5

1.2.4 Evaluation of Denial of Service Attacks 5

1.2.5 Security of Network Virtualisation 6

1.3 Research Methodology 6

1.4 Thesis Structure 7

2 Background **9**

2.1 Software Defined Networking (SDN) 9

2.1.1 OpenFlow 11

2.2 Network Security 14

2.2.1 Types of Network Security Attacks 14

2.2.2 Network Security Protocols and Technologies 16

3 Literature Review **18**

3.1 Security via SDN 19

3.2 Security of SDN 22

3.2.1 Security of the Control Plane 23

3.2.2 Security of the Data Plane 26
3.2.3 Security of the Application Layer 27
3.2.4 Security of the Southbound Interface 29

4 Security of Topology Discovery 32

4.1 Introduction 32
4.2 OpenFlow Discovery Protocol (OFDP) 33
4.3 OFDP Link Spoofing - Basic Vulnerability 35
4.3.1 Experimental Validation - Mininet 37
4.3.2 Experimental Validation - OFELIA 39
4.4 Impact on Routing 40
4.4.1 Linear Topology 41
4.4.2 Tree Topology 43
4.4.3 Discussion 45
4.5 Countermeasures 45
4.5.1 Controller Checks 46
4.5.2 LLDP Packet Authentication 46
4.6 Related Works 49
4.7 Conclusions 49

5 Security of Address Resolution Protocol 51

5.1 Introduction 51
5.2 Background 52
5.2.1 Address Resolution Protocol (ARP) 52
5.2.2 ARP Spoofing 54
5.2.3 Traditional ARP Spoofing Countermeasures 54
5.3 ARP Handling in SDN 55
5.4 ARP and NDP (NS/NA) Spoofing in SDN 56

5.4.1 Experimental Platform 56
5.4.2 Spoofing with Regular ARP 57
5.4.3 Spoofing with Proxy ARP 61
5.5 Countermeasure 1: SARP_DAI 62
5.5.1 SARP_DAI Overhead 63
5.6 Countermeasure 2: SARP_NAT 67
5.6.1 ARP Request based Attack 69
5.6.2 ARP Reply based Attack 71
5.6.3 SARP_NAT Overhead 72
5.6.4 Comparison with SARP_DAI 74
5.7 Related Works 75
5.7.1 DAI-like Approaches 76
5.7.2 Proxy ARP-based Approaches 77
5.8 Conclusions 79

6 Efficient Address Resolution Protocol Handling 80

6.1 Introduction 80
6.2 ARP Handling in SDN 81
6.3 Switch-based Proxy ARP (SProxy ARP) 82
6.4 Experimental Platform 85
6.5 Evaluation 86
6.5.1 ARP Response Time 87
6.5.2 Controller Overhead 88
6.6 Switch Memory-Performance Trade-off 90
6.7 Related Works 93
6.8 Conclusions 93

7 Evaluation of Denial of Service Attacks 95

7.1 Introduction 95
7.2 SDN Packet Forwarding 96
7.3 DoS Attacks against SDN 97
7.3.1 Attack on the Control Plane 97
7.3.2 Attack on the Data Plane 99
7.4 Experimental Evaluation 99
7.4.1 Testbed 99
7.4.2 Control Plane Attack 100
7.4.3 Data Plane Attack 105
7.5 Related Works 107
7.6 Conclusions 109

8 Security of Virtualisation **110**

8.1 Introduction 110
8.2 SDN Hypervisor Platforms 111
8.2.1 FlowVisor 111
8.2.2 OpenVirteX 113
8.2.3 Other SDN Hypervisor Platforms 114
8.3 SDN Virtualisation Vulnerabilities 116
8.4 Related Works 117
8.5 Experimental Platform 118
8.6 Security of FlowVisor 119
8.6.1 Topology Discovery 119
8.6.2 Breaking Isolation 122
8.6.3 Ping of Death 125
8.7 Security of OpenVirteX (OVX) 126
8.7.1 Topology Discovery 126
8.7.2 Breaking Isolation 128

8.7.3 Ping of Death 129
8.8 Conclusions 130

9 Conclusion 132

Bibliography 135

List of Figures

2.1 Software Defined Network Architecture [1] 10

2.2 OpenFlow Switch Components [2] 11

2.3 Main Components of OpenFlow Entry 12

2.4 Classification of Network Security Attacks 14

3.1 Active Security Architecture [3] 20

3.2 OrchSec Architecture [4] 21

3.3 LivSec Architecture [5] 22

3.4 Security Threat Vectors Map in SDN [6] 23

4.1 LLDP Frame Structure 33

4.2 Basic OFDP Example Scenario 34

4.3 Basic Attack Scenario (Mininet) 36

4.4 POX Debug Information (Mininet) 38

4.5 Basic Attack Scenario (OFELIA) 39

4.6 POX Debug Information (OFELIA) 40

4.7 Linear Topology for Routing Experiment 41

4.8 Tree Topology for Routing Experiment 44

4.9 Computational Overhead of HMAC in OFDP 48

5.1 ARP Frame Structure 53

5.2 Basic Example Scenario 57

5.3 Poisoned ARP Cache (Mininet) 58

5.4 Poisoned ARP Cache (OFELIA) . . . 59
5.5 ARP Spoofing Attack on ONOS . . . 60
5.6 ARP Spoofing Attack on Floodlight . . . 60
5.7 Poisoned ARP Responder Table . . . 61
5.8 Linear Topology . . . 64
5.9 Tree Topology . . . 64
5.10 Controller CPU Load (SARP_DAI) . . . 65
5.11 RTT in Linear Topology (SARP_DAI) . . . 66
5.12 RTT in Tree Topology (SARP_DAI) . . . 67
5.13 Controller CPU Load (SARP_NAT) . . . 72
5.14 RTT in Linear Topology (SARP_NAT) . . . 73
5.15 RTT in Tree Topology (SARP_NAT) . . . 73

6.1 Example OpenFlow Rule for SProxy ARP . . . 84
6.2 Basic Experiment Scenario . . . 86
6.3 ARP Response Time . . . 87
6.4 ARP Response Time vs. Background Controller CPU Load . . . 89
6.5 CPU Consumption with Sending Rate . . . 90
6.6 Total Packet size with Sending Rate . . . 91
6.7 ARP Request Distribution based on TPA . . . 91

7.1 Basic Attack Scenario . . . 98
7.2 Control Plane Attack, PDR . . . 101
7.3 Control Plane Attack, Controller CPU Load . . . 102
7.4 Linear Topology . . . 102
7.5 Attack Amplification Effect on PDR . . . 103
7.6 Attack Amplification Effect on Controller CPU Load . . . 104
7.7 Data Plane Attack, PDR . . . 105

7.8 Data Plane Attack, CPU Load 107

8.1 FlowVisor Architecture 112

8.2 OpenVirteX Architecture 114

8.3 Network Infrastructure 119

8.4 FlowVisor LLDP Frame Structure 120

8.5 FlowVisor Database Attack 121

8.6 Controller CPU Load 124

8.7 FlowVisor Crash, Ping of Death 126

8.8 OpenVirteX LLDP Frame Structure 127

8.9 OpenVirteX Database Attack 128

8.10 Network Traffic from Another Tenant 129

8.11 OpenVirteX Crash, Ping of Death 130

List of Tables

1.1 Software Tools used for Implementation Experiments in the Thesis 8

3.1 Summary of Proposed Security Measures for SDN Layers (Planes) 31

4.1 Software Tools used for Implementation and Experiments in Chapter 4 37

4.2 Connectivity After Attack 42

4.3 Connectivity Loss Due to Link Spoofing Attack 44

5.1 Software Tools used for Implementation and Experiments in Chapter 5 57

5.2 Controller CPU Load Comparison 75

5.3 Round Trip Time Comparison 75

6.1 Software Tools used for Implementation and Experiments in Chapter 6 86

6.2 Memory Performance Trade-off 92

7.1 Software Tools used for Implementation Experiments in Chapter 7 100

8.1 Classification of Network Virtualisation Threats 117

8.2 Software Tools used for Implementation and Experiments in Chapter 8 118

List of Abbreviations

The abbreviated terms are provided for reference throughout the thesis.

ARP	**A**ddress **R**esolution **P**rotocol
BGP	**B**order **G**ateway **P**rotocol
CPU	**C**entral **P**rocessing **U**nit
DAI	**D**ynamic **A**RP **I**nspection
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol
DoS	**D**enial **o**f **S**ervice
DU	**D**ata **U**nit
GUI	**G**raphical **U**ser **I**nterface
HMAC	**H**ash-based **M**essage **A**uthentication **C**ode
IPsec	**I**nternet **P**rotocol **Sec**urity
IP	**I**nternet **P**rotocol
LLDP	**L**ink **L**ayer **D**iscovery **P**rotocol
MAC	**M**edia **A**ccess **C**ontrol
MAC	**M**essage **A**uthentication **C**ode
MITM	**M**an **i**n **t**he **M**iddle
NA	**N**eighbour **A**dvertisement
NAT	**N**etwork **A**ddress **T**ranslation
NDP	**N**eighbour **D**iscovery **P**rotocol
NOS	**N**etwork **O**perating **S**ystem
NS	**N**eighbour **S**olicitation
OFDP	**O**penFlow **D**iscovery **P**rotocol
ONF	**O**pen **N**etworking **F**oundation
ONOS	**O**pen **N**etwork **O**perating **S**ystem
OVS	**O**pen **vS**witch

OVX	**O**pen**V**irte**X**
PDR	**P**acket **D**elivery **R**atio
PKI	**P**ublic **K**ey **I**nfrastructure
RAM	**R**andom **A**ccess **M**emory
RTT	**R**ound **T**rip **T**ime
SDN	**S**oftware **D**efined **N**etworking
SEND	**Se**cure **N**eighbour **D**iscovery
SHA	**S**ender **H**ardware **A**ddress
SPA	**S**ender **P**rotocol **A**ddress
TCAM	**T**ernary **C**ontent **A**ddressable **M**emory
TCP	**T**ransmission **C**ontrol **P**rotocol
THA	**T**arget **H**ardware **A**ddress
TLS	**T**ransport **L**ayer **S**ecurity
TLV	**T**ype-**l**ength-**v**alue
TPA	**T**arget **P**rotocol **A**ddress
UDP	**U**ser **D**atagram **P**rotocol
VLAN	**V**irtual **L**ocal **A**rea **N**etwork

Chapter 1

Introduction

1.1 Motivation

In traditional networks, the deployment and management of infrastructure and services is a complex task and requires the configuration of a large number of individual network devices such as routers and switches, typically via proprietary interfaces. Packet forwarding is controlled via complex, distributed routing protocols such as OSPF [7], BGP and EGP [8]. The lack of a centralised global view of the network state, as well as the lack of relevant networking abstractions, make it challenging to implement high-level forwarding policies in traditional IP networks. As a result, the network is configured rather than programmed, making innovation and the deployment of new networking services difficult and slow [9, 10].

Software Defined Networking (SDN) is a relatively new approach to manage and configure computer networks, which aims to address this problem through removing the control intelligence from forwarding elements, such as switches and routers, and placing it in a logically centralised node, i.e. the SDN controller [11, 12]. The separation of the control plane and the data plane in SDN makes the network more programmable. The complex and labour-intensive task of configuring individual networking devices is now replaced by the simpler and more efficient task of 'programming' the network.

In SDN, the Network Operating System (NOS) hides the complexity and details of the underlying network infrastructure, by providing an abstraction layer and clearly defined interfaces.

Network services and policies are implemented as applications that sit on top of the Network Operating System, running on the centralised controller. The controller communicates with the forwarding elements, i.e. the data plane, through a well-defined interface, e.g. the OpenFlow protocol [13].

As a result of the separation of the control plane and the data plane as well as the increased level of abstraction, SDN make the network more agile, flexible and programmable, which dramatically simplifies network management and configuration, and enables faster innovation. The implementation and deployment of new network services and policies, which take a considerable amount of time and effort in traditional networks, becomes a comparatively manageable task in an SDN-enabled network.

SDN has gained tremendous momentum, both in the industry and the research community, and has been successfully deployed in data centres and Wide Area Networks (WANs) [14]. For example, Google has deployed an SDN-based WAN in its internal backbone network to globally connect its various data centres. As a result, network performance has been enhanced dramatically, and the link utilisation has increased from 30 $\sim$ 40% to close to 100% [14]. This dramatic achievement came from the more fine-grained control over the forwarding of network flows, and the increased network programmability provided by SDN. It is likely that the growth of SDN will continue in the future, which is supported by the fact that major networking vendors such as Cisco [15], Huawei [16], Juniper [17] and Hewlett Packard [18] are increasingly supporting SDN-based products.

The fundamentally different approach to network management and configuration of SDN has significant implications for network security. There are two separate aspects of this. In the first one, to which we refer to as 'Security via SDN', the logically centralised view and programmability of SDN makes it easier to implement and enforce network-wide security policies.

The second security aspect of SDN, which we refer to as 'Security of SDN', considers the security of SDN platform itself, and is the focus of this PhD. Our hypothesis is that a fundamentally different network architecture such as SDN is likely to have new security vulnerabilities and provides a new range of attack vectors, and that past work on security in traditional networks cannot fully capture the security aspects of SDN.

The goal of this thesis was to analyse the security of the SDN architecture, components and services, and to identify security risks and vulnerabilities. A further goal was to practically demonstrate the feasibility of attacks, discuss and quantify their potential impact, and if possible, propose suitable countermeasures.

The security analysis in this thesis focuses on the following key SDN components, services and aspects: Topology Discovery, Address Resolution Protocol (ARP) Handling and Network Virtualisation. Furthermore, the thesis also specifically considers the problem of Denial of Service (DoS) attacks against the SDN platform and their potential impact. The corresponding research contributions are summarised in the following.

1.2 Research Contributions

1.2.1 Security of Topology Discovery

Topology discovery is a core service in SDN, and it underpins most network applications such as routing, access control, etc., by providing a global view of the network and the abstraction of the network as a graph. All major SDN controllers implement topology discovery using the OpenFlow Discovery Protocol (OFDP), making it the de-facto standard for topology discovery in SDN [19].

OFDP uses the packet format of the Link Layer Discovery Protocol (LLDP) used in traditional Ethernet networks [20], but operates completely differently. Given its important role in SDN, a security analysis of the OFDP protocol is essential for ensuring the overall security of any SDN platform.

In this thesis, key vulnerabilities of SDN's current topology discovery approach are identified, which are mostly due to the lack of authentication and integrity protection of LLDP packets. A link spoofing attack is discussed, implemented and experimentally evaluated, where an attacker can successfully corrupt the controller's topology view of the network by injecting a fabricated LLDP packet. The impact of this attack on higher level services is discussed and demonstrated via the example of routing.

The thesis proposes a countermeasure, which can prevent this type of attack, based on the addition of a Hash-based Message Authentication Code (HMAC) at the controller, which provides integrity protection for LLDP messages. We show that our approach is not vulnerable to replay attacks. Using experiments, we also quantify the computational cost of the proposed security mechanism.

1.2.2 Security of ARP

The Address Resolution Protocol (ARP) is used in computer networks to map an interface's network layer address (typically IP), to its corresponding Layer 2 or Media Access Control (MAC) address [21]. The vulnerability of ARP to spoofing attacks is a well-known problem in traditional computer networks [22], mostly due to its stateless nature, and lack of authentication and integrity protection. ARP spoofing attacks form a critical building block for a lot of Denial of Service (DoS), and Man-in-the-middle (MITM) attacks [23, 24].

In this thesis, we consider ARP security from the specific perspective of SDN, in particular its centralised control, which allows new approaches to ARP handling and providing security for ARP. We initially demonstrate the vulnerability of current SDN platforms to ARP spoofing attacks via experiments, for different approaches to ARP handling used in SDN, i.e. *Regular ARP* and *Proxy ARP*.

We then investigate Dynamic ARP Inspection (DAI) [25], an ARP spoofing protection mechanism used in traditional IP networks, and explore its adoption to SDN. We show that DAI can prevent ARP spoofing attacks in SDN, and we experimentally evaluate its overhead on the SDN control plane. DAI relies on the availability of a trusted database of IP-to-MAC address mappings. Such a database is not always available, and we, therefore, explore a new method to secure ARP without such a requirement. The new method presented in this thesis, called SARP_NAT, does not assume any trusted a-priori information of IP-to-MAC address mappings and leverages SDN's centralised control plane. The basic idea of SARP_NAT is to prevent any potentially spoofed ARP information from coming into contact with end-hosts and ARP handling components at the controller, and hence prevents the poisoning of the corresponding ARP caches and databases. This is achieved by implementing a controller component, which 'sanitises' ARP requests and replies by overwriting potentially spoofed

fields. We demonstrate the viability of this new, SDN specific approach of securing ARP, and present extensive experimental evaluations of its performance and cost.

1.2.3 Efficient ARP Handling

ARP handling in SDN is typically an expensive operation, depending on which approach is chosen, i.e. *Regular ARP* or *Proxy ARP*. In the case of Regular ARP, significant network bandwidth is required to broadcast ARP request messages in the network whereas Proxy ARP imposes a significant computational load on the control plane since the SDN controller handles ARP requests.

We developed a new OpenFlow-based approach for handling ARP in SDN, which achieves much greater efficiency by offloading the task of answering ARP request to the data plane, i.e. the SDN switches. Our experiments show that this approach significantly reduces the time required to handle ARP requests, and significantly reduces the load on the SDN controller, compared to the current state-of-the-art approach.

While this contribution is more of a general nature, and not exclusively focussed on security, it does have important security implications and benefits. By significantly reducing the load of the SDN controller, we can make the controller, and therefore the entire network, less vulnerable to Denial of Service (DoS) attacks, as will be discussed in more detail in Chapter 6.

1.2.4 Evaluation of Denial of Service Attacks

Denial of Service (DoS) attacks are a common problem in traditional networks. According to [26], the estimated annual cost of the impact of DoS attacks is US $113 billion globally. SDN, with its logically centralised control plane, represents a unique target for DoS attacks. If an attacker manages to disable the controller, the entire network can be disrupted.

In this thesis, we provide an extensive experimental evaluation of the impact of DoS attacks on different SDN controller platforms. We also consider the impact of the attacks on the

data plane, i.e. SDN switches. Our results show that an attacker, with relatively minimal effort, can significantly disrupt modern SDN controllers, and their ability to forward legitimate network traffic.

1.2.5 Security of Network Virtualisation

Network virtualisation is a highly desirable feature in today's large-scale computer systems, in particular in data centres. One of the key benefits of SDN is that it enables network virtualisation. Using network virtualisation, multiple SDN controllers can share the same physical network infrastructure. Network virtualisation is widely used in SDN, and the most relevant SDN hypervisors are FlowVisor [27] and OpenVirteX [28]. With the new potential that network virtualisation in SDN brings, they also represent a potential for new security vulnerabilities.

In this thesis, we present a first extensive evaluation and analysis of the security of the network virtualisation layer in SDN, with a focus on FlowVisor and OpenVirteX. By using code analysis and fuzz testing [29], we found a number of new, critical security vulnerabilities in FlowVisor and OpenVirteX. We show how an attacker can exploit these vulnerabilities to break the isolation between virtual networks, and how a node on one virtual network can successfully disrupt another virtual network, or in some cases, completely disable the entire network. From our results, we can conclude that significant further efforts are required to guarantee the security of SDN hypervisors.

1.3 Research Methodology

The research methodology used in this thesis for the evaluation of various security aspects of SDN is largely experimental. This section provides a summary of the software tools, and platforms that formed the basis of our experimental evaluation.

A key platform used for most of our experiments is Mininet [30], a Linux-based network emulator. Mininet allows the creation of a network of virtual SDN switches and hosts, con-

nected via virtual links. An important advantage of Mininet is the ability to run real network code, which allows experiments to be easily transferred to hardware SDN test-beds. Mininet uses Open vSwitch (OVS) [31], a popular and widely supported software OpenFlow switch. Some of our experiments have also been replicated on a hardware SDN test-bed. For this, we used the OFELIA test-bed [32], a federated experimental SDN facility shared between a number of *SDN islands* across various European countries (e.g. UK, Switzerland, Germany, Belgium, Spain, and Italy), as well as Brazil. Each island is equipped with a range of SDN hardware switches, supporting the OpenFlow 1.0 standard [2]. The model of the OFELIA switches used in all our experiments was NEC IP8800//S3640-24T2XW. We used the resources located at the OFELIA island in Trento, Italy. The SDN controller platforms used in this thesis include POX [33], Ryu [34], ONOS [35], FloodLight [36], and OpenDaylight [37].

To implement and analyse different SDN security attacks, we used a range of software tools, such as Scapy [38], Dsniff [39], PackETH [40], Tcpreplay [41], Stress-ng [42], and Netcat [43]. Table 8.2 shows a summary of the key software tools that were used for the experimental evaluations conducted in this thesis.

1.4 Thesis Structure

The structure of the remainder of this thesis is as follows:

- Chapter 2 provides the relevant background on Software Defined Networks, OpenFlow and network security.
- Chapter 3 presents a general overview of the most relevant works on SDN security. The works that are more specifically related to each of the specific contributions presented in this thesis are discussed in more detail in the corresponding chapters.
- Chapter 4 presents our security analysis of SDN's current topology discovery mechanism, as well as the implementation and evaluation of our proposed improvements.
- Chapter 5 presents the security analysis of ARP handling in SDN, as well as our proposed countermeasures against ARP spoofing attacks.

Table 1.1: Software Tools used for Implementation Experiments in the Thesis

Software	**Function**	**Version**
Mininet [30]	Network Emulator	2.1.0-2.2.2
Open vSwitch [31]	Software SDN Switch	2.0.2-2.6.1
OFELIA [32]	Hardware SDN Test-bed	=======
POX [33]	SDN Controller Platform	*dart* branch
Ryu [34]	SDN Controller Platform	3.19-3.22
ONOS [35]	SDN Controller Platform	1.8.5-1.11.1
Floodlight [36]	SDN Controller Platform	1.0
OpenDaylight [37]	SDN Controller Platform	Carbon SR1
Scapy Library [38]	Packet Manipulation Tool	2.2.0
Dsniff Package [39]	Network Sniffing Tool	2.4
PackETH [40]	Packet Generator	1.8.1
Tcpreplay [41]	Traffic Replay Tool	4.2.6
Stress-ng [42]	Control Traffic Tool	0.02.26
Netcat [43]	Network Sniffing Tool	5.59*BETA1*
VM-VirtualBox [44]	Oracle Virtualisation	5.0.10

- Chapter 6 introduces a new, efficient approach to handle ARP in SDN.
- Chapter 7 presents an evaluation of the impact of DoS attacks against the control and data plane in SDN.
- Chapter 8 presents new security vulnerabilities in the two most relevant SDN virtualisation platforms and experimentally demonstrates their impact.
- Chapter 9 concludes the thesis and provides directions for potential future work.

Chapter 2

Background

2.1 Software Defined Networking (SDN)

Software Defined Networking (SDN) is a new approach to managing computer networks and has recently gained tremendous momentum [45]. One of the essential concepts of the SDN technology is the separation of the control plane from the data plane. The control plane (intelligence), which determines how network packets are being forwarded, is removed from the data plane, which is responsible for the actual forwarding of packets. The network control function is (logically) centralised in an entity called the *SDN controller* that allows network operators to programmatically configure network behaviour and directly manage the entire network elements from a single management point [46, 47]. This concept facilitates network evolution, boosts innovation processes, automates network management, and optimise network configurations.

The conceptual architecture of SDN, as illustrated in Figure 2.1, consists of three layers (infrastructure layer, control layer and application layer). The bottom layer, i.e. *the infrastructure layer*, is basically a set of forwarding elements, i.e. SDN switches, which provide basic packet forwarding functionality based on decisions made by the control layer, i.e. forwarding rules provided by the SDN controller.

The middle layer is the *control layer*, consisting of a logically centralised SDN controller, implementing the functionality of a Network Operating System (NOS) [48]. The NOS deals with

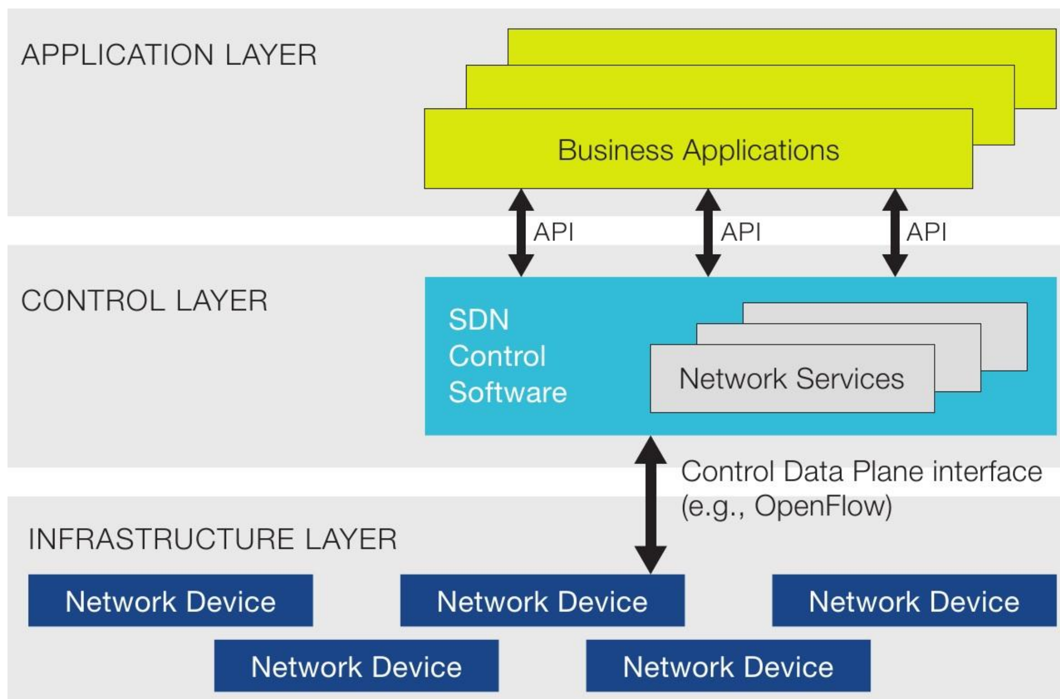


Figure 2.1: Software Defined Network Architecture [1]

and hides the distributed nature of the physical network, and provides the abstraction of a network graph to higher layer services, which sit at the *application layer* of the SDN architecture [49]. This abstraction makes the network much more programmable and simplifies the implementation and deployment of new network services and applications. The SDN controller manages and configures individual SDN switches by installing forwarding rules, via the so-called *southbound interface*. The predominant standard for this is OpenFlow, which allows the SDN controller to manipulate forwarding rules of the OpenFlow switches [2]. OpenFlow is discussed further in the next section.

At the top of the SDN architecture is the *application layer*, where high-level network policy decisions are defined and applications and services such as Traffic Engineering (TE), routing, firewalling, etc., are implemented. The interface between the application layer and the control layer is referred to as the *northbound interface*. In contrast to the southbound interface, there is currently no well-established standard for this, and different SDN controller platforms support different APIs [1, 10, 45, 46].

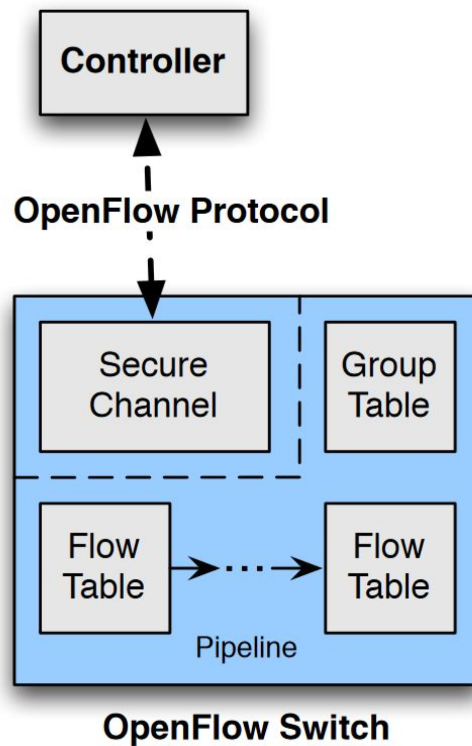


Figure 2.2: OpenFlow Switch Components [2]

2.1.1 OpenFlow

OpenFlow is the predominant southbound interface protocol for SDN. It provides the interface between the infrastructure layer and the control layer as shown in Figure 2.1, which allows the SDN controller to talk to the forwarding elements (switches). The OpenFlow standard is maintained by the Open Networking Foundation (ONF). OpenFlow is a wire protocol that allows the SDN controller to manipulate network traffic across the forwarding elements, i.e. via decisions that are translated into forwarding rules and actions. It also allows switches to notify the controller about special events, e.g. the receipt of a packet that does not match any installed rules [13]. The OpenFlow specification has evolved from version 1.0 to the current version 1.5 at the time of writing this thesis.

The core components of an OpenFlow switch that perform a packet lookup and forwarding operation are illustrated in Figure 2.2. They consist of one flow table (in the case of OpenFlow switch v1.0) or more flow tables in a pipeline (in the case of OpenFlow switch v1.2 - v1.5), a group table, and an OpenFlow channel, connecting the switch to an external SDN controller.

OpenFlow switches are assumed to be configured with the IP address and TCP port num-

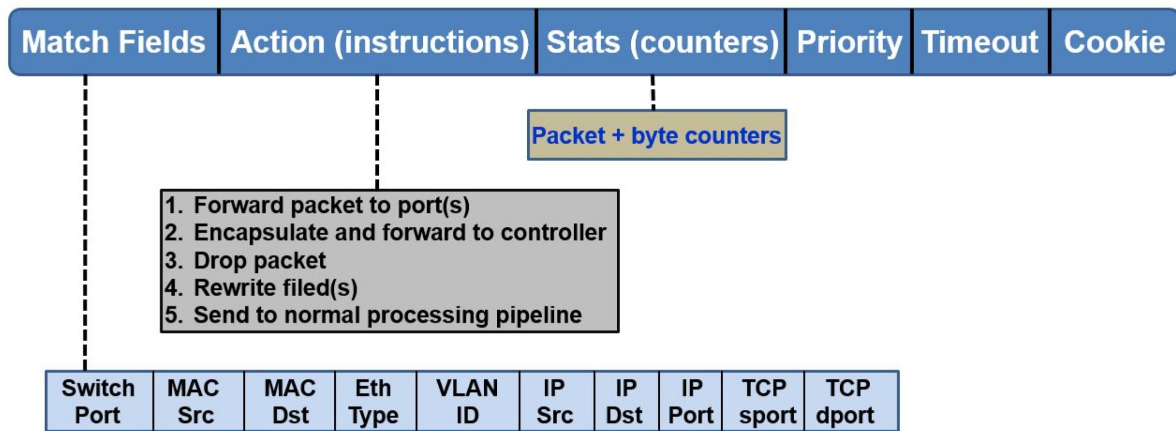


Figure 2.3: Main Components of OpenFlow Entry

ber of their assigned SDN controller. At initialisation, switches contact the SDN controller, via this address and port, and establish a secure Transport Layer Security (TLS) connection. As part of the initial protocol handshake, the controller sends an OpenFlow *OFPT_FEATURES_REQUEST* message to each OpenFlow switch, requesting configuration information, including the number of switch ports and corresponding MAC addresses [50]. This initial handshake informs the controller of the existence of the nodes (switches) in the network, but it does not provide any information about the active inter-switch links, i.e. the network topology. Gathering this information is the role of OFDP, which will be discussed in more detail later.

As mentioned above, OpenFlow allows controllers to access and configure the forwarding rules, i.e. *flow entries* in the flow tables at SDN switches. These rules, which provide fine-grained control over how packets are forwarded through the network, can be installed reactively as a response to received packets, or proactively. Each forwarding rule consists of three main parts: *Match fields* (rules), *Action* (instructions), and *Statistics* (counters), as shown in Figure 2.3.

- The *Match* fields are basically the selectors that OpenFlow switches rely on to filter incoming packets. The supported match fields include the switch ingress port and various packet header fields, such as IP source and destination address, MAC source and destination address, UDP/TCP source and destination port number and more, as shown in the figure. The value of the header fields can be either fixed or set as wildcards, i.e. made to match any value [12, 46].

- The *Action* field defines how packets that match specified match field values are

treated. The main actions supported by an OpenFlow switch include forwarding a packet on a particular switch port, dropping the packet, enqueuing the packet and or modifying the value of a specific field. Switch ports can either be physical ports or one of the following virtual port types: *ALL* (sends the packet out on all physical ports, except the ingress port), *CONTROLLER* (sends the packet to the SDN controller), *FLOOD* (same as *ALL*, excluding the ports disabled by the spanning tree protocol) [1, 9].

- The *Statistics* field is typically a collection of counters, which can be per table, flow, port, and queue to count how many packets and bytes passing through the switch match this forwarding rule.
- The *Priority* field defines the matching precedence of the forwarding rule. For example, a forwarding rule with a high priority is executed before the rest.
- The *Timeout* field specifies the idle and maximum time of the forwarding rule before it is removed from the flow table.
- The *Cookie* field is data value selected by the controller and may be used to filter flow statistics and flow modification.

OpenFlow switches support a basic *match-action* paradigm, where each incoming packet is *matched* against a set of rules, and the corresponding *action* or *action list* is executed. The default behaviour of OpenFlow switches is to send the packet to the controller when it does not match any of the rules.

To send a data packet to the SDN controller, an OpenFlow switch encapsulates the packet in an OpenFlow *Packet-In* message. OpenFlow also supports an OpenFlow *Packet-Out* message, via which the SDN controller can send a data packet to an OpenFlow switch, together with instructions (*action list*) on how to forward the packet [13, 46].

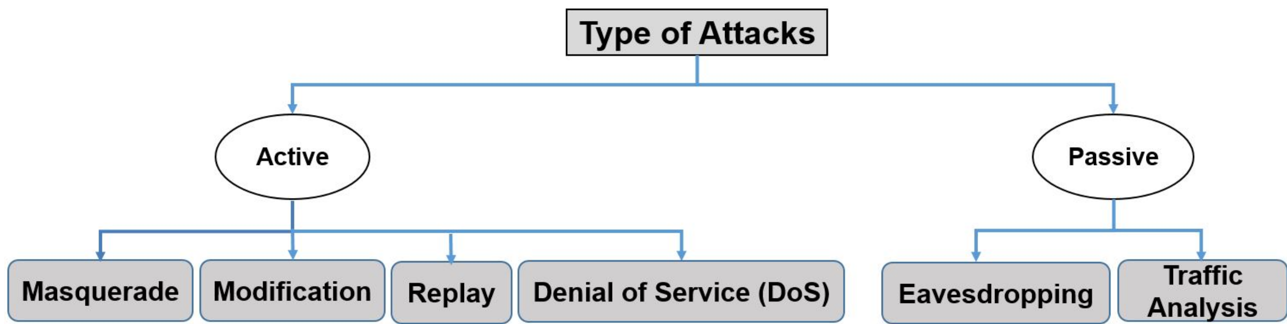


Figure 2.4: Classification of Network Security Attacks

2.2 Network Security

Network security is concerned with providing the core properties of secure communications, such as confidentiality, integrity and availability [51]. In this context, confidentially aims to ensure that network services and information are only accessible to authorised users, while integrity aims to prevent unauthorised modification and deletion of data traversing the network. Availability refers to the guaranteed and reliable access to the information and services by authorised people.

A key tool to implement network security is cryptography, in which electronic data is encrypted in a particular form that makes the data only accessible to authorised entities. Encryption provides confidentiality, by preventing unauthorised entities from reading messages, while cryptographic checksums or Message Authentication Codes (MAC) provide both data integrity and authenticity [52]. Unfortunately, no simple cryptographic solution guarantees availability, e.g. prevents Denial of Services (DoS) attacks.

In this section, we first provide a basic classification of network security attacks and then discuss some of the key security protocols, standards and technologies that are used to secure computer networks.

2.2.1 Types of Network Security Attacks

Network security attacks are classified based on the behaviour of the attacker, as shown in Figure 2.4. The two main categories are *Passive* attacks and *Active* attacks [53, 54].

- Passive attacks are network exploits, in which an attacker passively observes network traffic and captures data that is being transmitted over a network without involving any modification or deletion of the data. Eavesdropping is a type of passive attack, where the attacker aims to read the content of the messages to reveal sensitive information. This can relatively easily be prevented via encryption. An attacker can still gain some information by observing the flow of encrypted information. The traffic pattern can reveal critical information, e.g. who is talking to whom, when, how often, etc. This kind of passive attack is called *traffic analysis*. The detection of passive attacks is generally difficult, due to the nature of the attack. [55].

- In active attacks, an attacker attempts to either modify or delete data. There are four main types of active attacks:

  1. In masquerade attacks, an attacker uses a false identity to gain unauthorised access to services or information. This attack can be prevented by cryptographic means, e.g. message authentication codes or digital signatures.
  2. In modification attacks, an attacker modifies an intercepted message, e.g. the packet header or the payload. This requires the attacker to be in the data path between the sender and receiver, i.e. as a *Man-in-the-Middle*. This attack can also be prevented by the same cryptographic means as used to prevent masquerade attacks, i.e. message authentication codes or digital signatures.
  3. In replay attacks, an attacker passively captures and intercepts a stream of messages transmitted between two legitimate users and fraudulently replays the message back to one of the users. This attack is slightly more challenging to detect, and basic methods based on message authentication codes are vulnerable to reply attacks. This will be discussed in more detail in Chapter 4.
  4. Denial of Service (DoS) attacks aim to remove or reduce the availability of services to legitimate users. This is often done by simply overwhelming a server or a network with illegitimate service requests or packets. As mentioned before, there is no simple cryptographic method to prevent general DoS attacks.

  In contrast to the passive attacks, active attacks can generally be more easily detected, due to the active intervention of the attacker.

A more detailed overview of network security attacks is provided in [52, 55, 56]

2.2.2 Network Security Protocols and Technologies

There are a number of ways in which networks can be protected from different types of attacks. As mentioned above, cryptography is a useful tool and can be implemented in protocols at different layers of the network protocol stack [57]. Two of the most relevant security protocols in this context are Transport Layer Security (TLS) / Secure Socket Layer (SSL) and Internet Protocol Security (IPsec).

TLS (or SSL) [58] is a cryptographic protocol at the transport layer that establishes a secure end-to-end connection across the network. TLS is the most widely used security protocol in computer networks, and it provides data confidentiality, integrity and authenticity via a range of cryptographic algorithms.

IPsec [59] is a suite of security protocols, which provide cryptographic security at the network layer. IPsec supports key negotiation, authentication, encryption and data integrity similar to TLS.

In addition to cryptographic tools, there are a number of other technologies used to provide network security. Some of the key examples include Firewalls, Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS).

A firewall is a device (software or hardware based) that monitors and filters the flow of packets based on some pre-defined security rules. Firewalls can operate at different layers of the protocol stack, typically layers 3 and 4, and can be state-less or state-full. Firewalls can also be host-based or network-based [60].

An Intrusion Detection System (IDS) passively monitors the flow of network traffic with the aim of identifying any policy violations and detecting abnormal and suspicious network traffic. The detection mechanism can either be *signature based*, in which packets are compared against a special signature in the database, or *anomaly based*, in which packets are compared against an established baseline network behaviour [61, 62, 63]. In contrast to typical firewalls, IDSs also look at the packet payload, i.e. they do Deep Packet Inspection (DPI).

An Intrusion Prevention System (IPS) is similar to IDS, but in addition to alerting, it can ac-

tively prevent or mitigate network attacks, e.g. via blocking malicious traffic or disconnecting infected hosts [64].

Chapter 3

Literature Review

The fundamental characteristic of the SDN architecture, i.e. the separation of the control plane from the data plane, represents a two-sided concept from a security perspective. On the one side, SDN allows the implementation of network security functions and policies in a new and more simple approach, which has not been possible in traditional networks [65, 66, 67]. We refer to this as 'Security via SDN'.

On the other hand, the new architecture of SDN brings a range of new potential attack vectors, including attacks against the control plane and the data plane [68, 69, 70]. We refer to this aspect of SDN security, i.e. the security of the SDN platform itself, as 'Security of SDN'.

While there have been quite a lot of works exploring how SDN can be used to implement different network security functionality, i.e. 'Security via SDN', there has been relatively less attention to the security of the SDN architecture and platform, i.e. 'Security of SDN', which is the focus of this thesis.

This chapter first provides a broad overview of key works related to 'Security via SDN', and then discusses key works related to the security of the SDN architecture itself, i.e. 'Security of SDN'. More detailed discussions of works that are specifically relevant to the contributions presented in this thesis are provided in the corresponding chapters.

3.1 Security via SDN

SDN technology has emerged initially with the objective to improve network management. One of the initial papers that laid the foundation of the current form of SDN is [71], where Secure Architecture for the Network Enterprise (SANE) was proposed to ease the network management and configuration of security middle-boxes of legacy networks through centralising the logical network functionality of access control decisions and policies.

SANE primarily concentrates on the registration and authentication mechanisms of network elements to establish communication successfully. Therefore, no access to services is provided unless end-hosts grant an explicit permission, i.e. authenticated and registered at the controller.

This new network architecture paradigm is considered a radical change to the traditional network architecture and a catalyst for the development of Ethane [72], an extension of SANE. Ethane is basically a security management architecture combining special Ethernet switches that are extended to track flows in-progress. In Ethane, the network-wide policy decides the network path that packets are supposed to follow. The proposed solutions resulted in the widespread adoption of the SDN platform. More recently, SDN has been increasingly used to enhance network security and to simplify the deployment of new security services [73, 74]. In this section, we provide a brief overview of the key works in this space.

Active security [3] is an SDN-based architecture that implements advanced security mechanisms through a centralised and unified programming interface to detect complex attacks and protect the network infrastructure. Figure 3.1 shows the Active security architecture, including its two layers: cyber infrastructure and Active security controller. The cyber infrastructure layer is a set of network forwarding devices, end-hosts, and security middle-boxes. The next layer is the Active security controller, which is extended beyond its pivotal role of controlling the network forwarding devices to communicate with the end-hosts and the security middle-boxes. The controller, which constitutes the most significant part of the Active security architecture, continuously and passively monitors network traffic and collects information about the current state of the infrastructure. It additionally gathers forensic evidence on-demand at runtime for attribution, while countering the attack through sophisti-

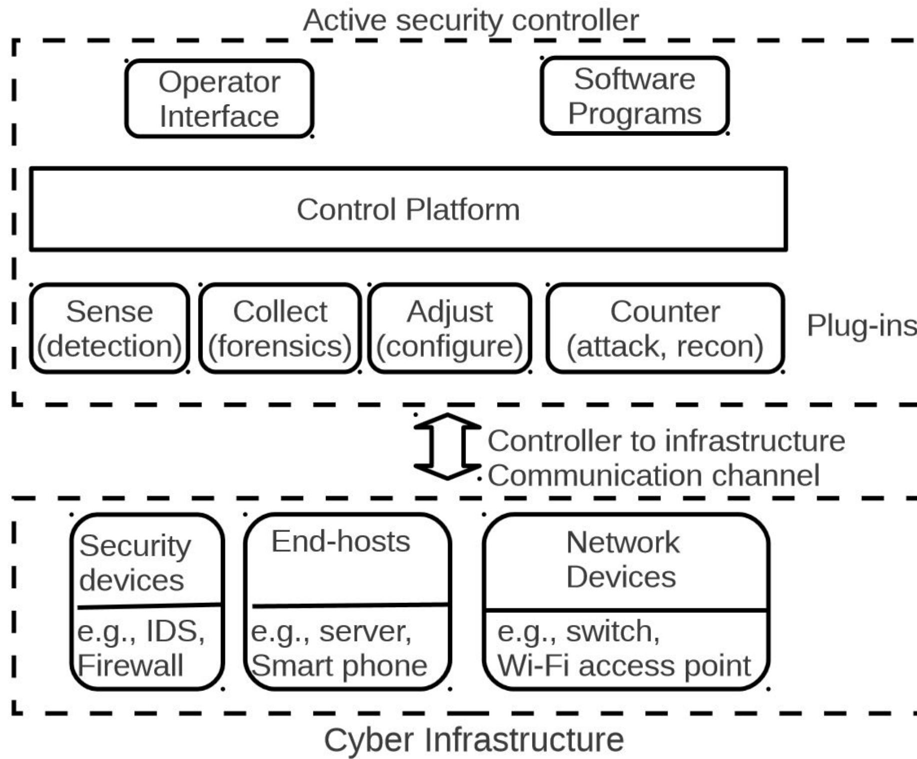


Figure 3.1: Active Security Architecture [3]

cated mechanisms such as moving malicious and dangerous code to a walled-off system. The authors create a preliminary prototype that goes beyond the SDN controller to automatically interact and exchange the attack information with dedicated network security devices such as the Snort Intrusion Detection System (IDS) [63] to detect anomalies and collect forensic evidence.

When anomalies are detected in network traffic flowing across the infrastructure, the Snort IDS reports the activity to the Active security controller, which is capable of analysing any malicious traffic and isolates it from normal traffic to a quarantined machine. Based on the results of collected data and forensic evidence of the attack statistics, the Active security controller immediately takes a proper action and dynamically adjusts the configuration of the infrastructure at runtime.

OrchSec [4] is an orchestrator-based architecture that mainly aims to improve network security and increase the system performance, flexibility, reliability and reliance through abstracting the control and network monitoring functions from the control plane and placing them at an additional layer, i.e. the *Orchestrator*. The OrchSec architecture is composed of three layers: infrastructure layer, control layer and orchestration layer, as shown in Figure 3.2. The

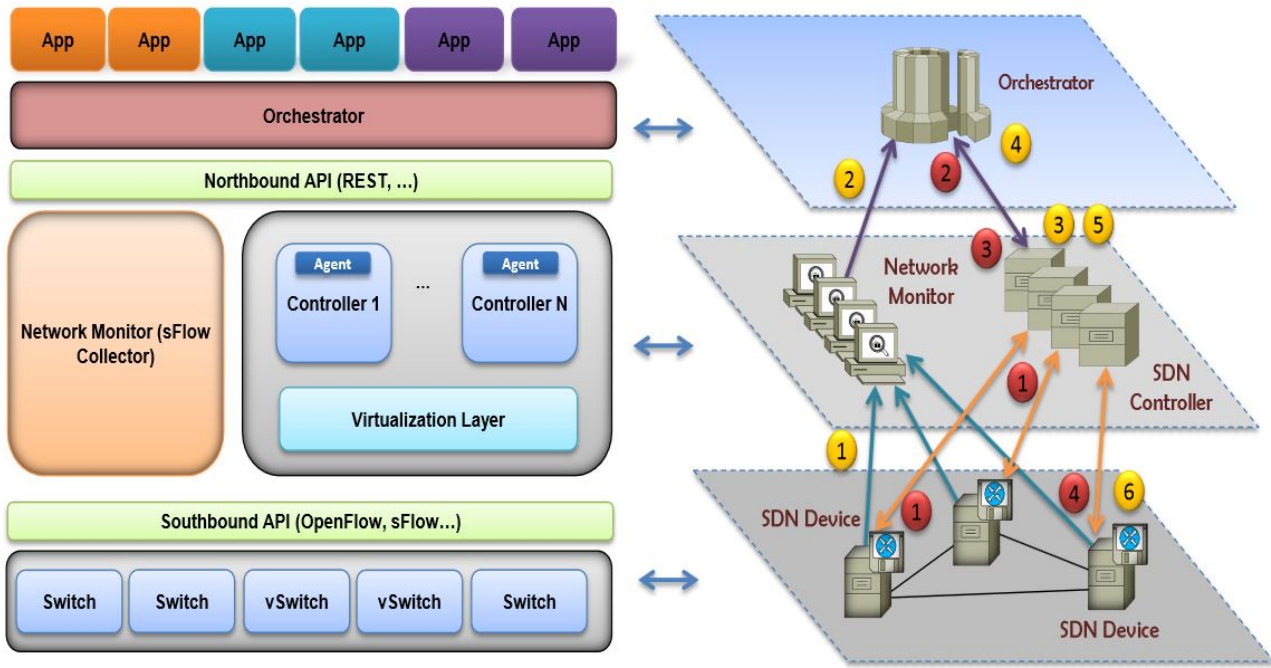


Figure 3.2: OrchSec Architecture [4]

infrastructure layer is basically a collection of SDN forwarding elements, e.g. switches and routers, that are responsible for forwarding network packets, not including any security devices. The control layer includes multiple SDN controllers running over a virtualisation layer and a network monitor that utilises sFlow [75] for network traffic sampling. The main function of the controllers is to coordinate between the Orchestrator and the control plane through a special application installed in all SDN controllers, i.e. *the Orchestrator agent*, and initiate control traffic messages, e.g. flow rules. At the top is the orchestration layer (Orchestrator), the component of the architecture where network security applications are implemented and deployed.

LiveSec [5] is another SDN-based architecture that fundamentally aims to provide scalable and flexible network security management in large-scale production networks through interactive policy-enforcement, real-time traffic monitoring and distributed load-balancing. In contrast to previous architectures, LiveSec basically appends a new layer to the traditional network architecture, and thus the LiveSec architecture consists of three layers: legacy-switching layer, access switching layer and control layer, as shown in Figure 3.3. The access-switching layer is used to interconnect the control layer and the legacy-switching layer through a set of OpenFlow-enabled switches. It provides legitimate interfaces for the control plane, i.e. LiveSec controller to access the legacy-switching layer and the ability to attach various security elements. The legacy-switching layer is a set of Ethernet switches

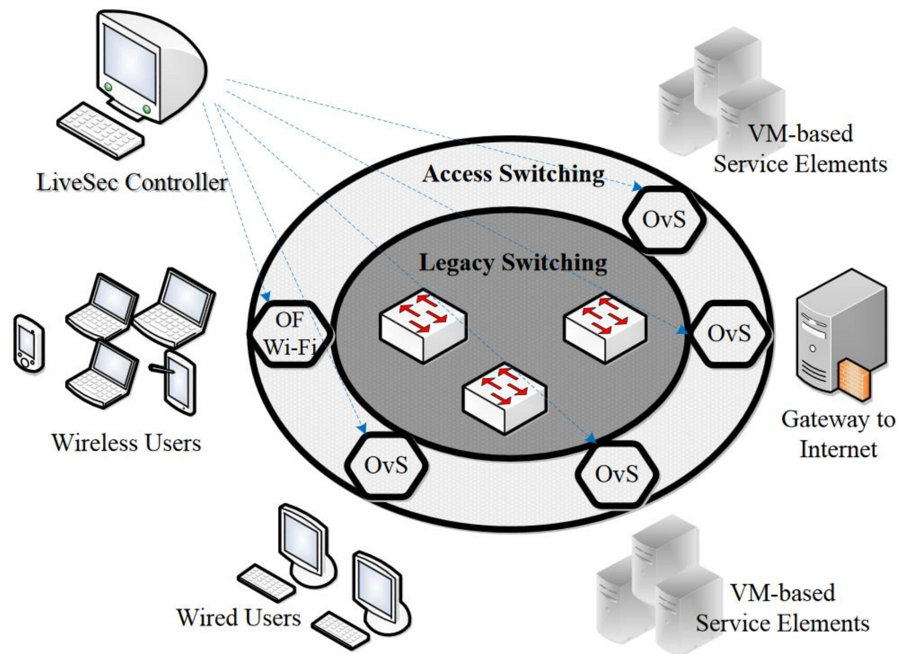


Figure 3.3: LivSec Architecture [5]

that perform layer-2 forwarding. The integral part of the LiveSec architecture is the control plane, i.e. the *LiveSec Controller*, which is responsible for identifying and locating the source of real-time security events.

3.2 Security of SDN

As mentioned before, the different architecture of SDN with its logically centralised control plane creates potentially new vulnerabilities and attack vectors that do not exist in traditional networks, or not to the same extent.

In [6], seven main threat vectors are identified in the SDN design that potentially stand in the way of achieving a secure network environment and keeping all network devices operating properly. Figure 3.4 shows an overview of these threats, which are indicated respectively as (1) forged or faked traffic flows, (2) attacks on vulnerabilities in switches, (3) attacks on control plane communications, (4) attacks on vulnerabilities in controllers, (5) lack of trusted mechanisms between controller and management applications, (6) attacks on vulnerabilities in administrative stations, and (7) lack of trusted resources for forensics and remediation.

Inspired by this, we group our discussion of key works on the security of the SDN architecture

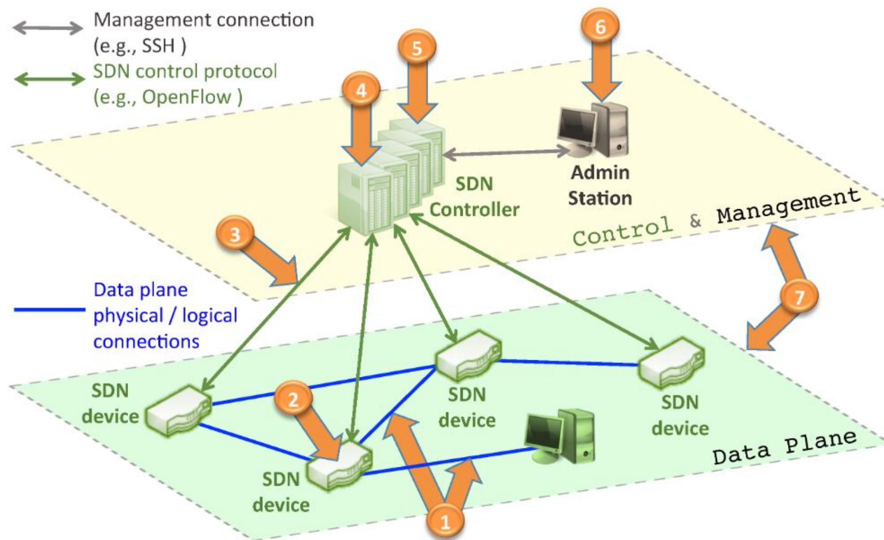


Figure 3.4: Security Threat Vectors Map in SDN [6]

into four categories: security of the control plane, security of the data plane, security of the application layer, and security of the southbound interface.

3.2.1 Security of the Control Plane

Aggregating the whole network function to a centralised entity (the SDN controller) appears to present a double-edged sword to the SDN architecture. It evidently improves network management and operation through the global view of the network [46, 76]. However, the control plane, i.e. the SDN controller, becomes a serious single point of failure risk and a prime target for attackers to exploit. The centralised nature of the control plane of the SDN architecture can potentially degrade the reliability and scalability of SDN, particularly in data centres [77].

To address this problem, Onix [78] was proposed as a logically centralised SDN controller, that is a physically distributed system. HyperFlow [79] extended the idea further by maintaining network control centralisation with distributed decision making, thereby minimising the look-up overhead and the response time of data plane requests imposed by sending them to the control plane. DISCO [80] is another distributed control plane, where each DISCO controller manages and controls its network domain and coordinates with other DISCO controllers to provide network services. A comprehensive analysis of various reactive and proactive SDN controllers, with performance evaluation and scalability measurement, is presented

in [81].

Despite the physical distribution of the SDN controller, its logical centralisation still presents an interesting target for attackers, since bringing down the controller makes the entire network 'headless' and consequently disrupts the entire network. Therefore, the security of the SDN control plane is absolutely critical for the security of SDN overall [82]. In the following, we summarise some of the key works that have been done in this area, with a focus on DoS attacks.

Denial of Service (DoS) and Distributed Denial of Service (DDoS) attacks have recently become a primary concern in SDN, due to the centralised nature of the control plane and the lack of network intelligence at the data plane [68, 83]. Implementing robust DoS and DDoS attack detection can be quite difficult, due to the similarity between normal and abnormal traffic, i.e. malicious packets sent by compromised hosts, in which OpenFlow switches are unable to expose during the packet forwarding process. For this reason, recent DoS and DDoS attack detection solutions exploit traffic flow statistics to produce efficient and stable DoS and DDoS protection system. Braga et al. [84] proposed a lightweight DDoS attack detection method that is configured on traffic flow features to extract information with low overhead. It basically monitors OpenFlow switches and retrieves active flow statistics at regular intervals to increase the rate of detection and lower the rate of false alarms. Based on flow statistics and flow manipulation functions, network traffic is then classified as legitimate or malicious by using Self Organising Maps (SOM), an unsupervised machine learning algorithm [85].

Another DDoS detection method is proposed in [86], where the Locator/ID separation protocol (LISP) [87] is used to analyse the frequency of network traffic and identify the malicious source of the DDoS attack. Suh et al. [88] presented a content-oriented networking architecture (CONA) that is capable of creating flows for the host sending a request to the server and the type of content requested. It then computes the difference between the request rate and a pre-defined value, and a DDoS attack is detected when the request rate exceeds the specified threshold.

The paper [89] discussed another approach to launch a DDoS attack against the control plane of SDN, in which an attacker continually generates various network packets with ran-

dom headers from multiple nodes to make the controller unable to handle normal network traffic. To increase the resilience of SDN, the authors proposed the use of second SDN controller (CPRecovery component) acting as a standby backup that replicates and synchronises with the primary controller to maintain a consistent and up to date global network view. In the event of a primary controller failure, the second controller is activated and becomes the primary Network Operating System (NOS).

In [90], the authors extended the SDN architecture to improve network security by distributing the security functions between the control plane and the data plane. In this architecture, a local security detection agent, Local Frequent Sets Analyser (LFSA), is installed on each SDN switch to primarily analyse network traffic patterns and identify vulnerabilities using data mining. Another security detection agent, Global Frequent Sets Analyser (GFSA), is installed globally on the SDN controller to continuously monitor the LFSA logs, which hold information about detected traffic anomalies and triggers appropriate action as a response to any detected network threat. A key limitation of this approach is that it does not work on standard OpenFlow switches, and it requires a significant extension of the data plane, which goes against the 'philosophy' of SDN, separating control and data plane functionality.

Dotcenko et al. [91] presented a fuzzy logic-based information security management system that performs intrusion detection and prevention and simultaneously evaluates the security level of the network, aiming to provide a more secure network environment. The proposed system at the beginning collects and aggregates statistical network data through real-time capturing. It then relies on the combination of popular network anomaly detection algorithms [92, 93], to distinguish between normal and abnormal traffic. Upon detecting an attack, the system dynamically makes a decision based on fuzzy logic, regarding which countermeasures, e.g. rate limiting should be applied. The authors claim that this requires less computational overhead than other approaches.

Klaedtke et al. [94] discussed the fundamental security concepts of SDN controllers and found that previous access control schemes are insufficient to provide SDN controllers with a secure environment. This resulted in the proposal of a new network-level access control scheme based on the OpenFlow protocol. The proposed policy enforcement is placed at the control plane, which allows the SDN controller to access the flow tables and entries of OpenFlow switches. It mainly aims to protect network flows and resolve conflicts derived

from network component reconfiguration. The recommendation of the paper was that network components (SDN applications), which are shared between users, should be logically separated and users' security requirements should be expressed and enforced.

3.2.2 Security of the Data Plane

In an OpenFlow-based SDN, switches simply forward packets based on match-action rules installed in flow tables. Hardware SDN switches typically use Ternary Content Addressable Memory (TCAM) to maximise forwarding speed. This memory is expensive and therefore limited in size. This represents a target for DoS attacks, in which attackers try to exhaust the TCAM memory resources. Also, OpenFlow switches generally need to buffer data packets that do not match any pre-defined flow rules and wait until the controller finishes processing the packets and issuing the corresponding flow rules. This could ultimately result in exhausting the switches' resources, especially in large-scale networks [70]. In [95], the authors analyse potential security vulnerabilities across the SDN platform, emphasising the impact of the default forwarding process on the data plane. An attacker can readily perform a Denial of Service (DoS) attack against the data plane, via setting up a large number of new unknown flows, which result in the installation of a large number of new flow rules by the controller, thereby exhausting the memory of the switch.

DevoFlow [96] introduced an extension of the current OpenFlow protocol to minimise the frequent interaction between OpenFlow switches and the SDN controller, as well as the number of required TCAM entries through the use of wild-carded OpenFlow clone rules. In the proposed implementation, DevoFlow-based OpenFlow switches replicate the active and pre-installed flow rules for microflow rules that match the header fields of packets and then use the microflow rules for updating and configuring global flow rules. Subsequently, the SDN controller of the DevoFlow architecture is only responsible for the brunt of defining and inserting flow rules and policies for the Quality of Service (QoS), which further devolves centralisation to avoid network overhead and increase scalability. Additionally, DevoFlow-based OpenFlow switches include new mechanisms that allow the switches to perform routing decisions locally, thereby avoiding controller involvement.

Similar to DevoFlow, DIFANE [97] extends the OpenFlow protocol and offloads some of

the control plane functionality to the data plane through authorising a subset of switches with sufficiently large memory and processing capabilities for handling a portion of network traffic. When network traffic does not match any cached flow rules, the receiving switch forwards the packet to one of these authorised switches, which handle the packet locally in the data plane instead of contacting the controller. Like DevoFlow, this eventually results in the reduction of the number of controller-switch interactions and the increase of network scalability. The primary differences between DevoFlow and DIFANE are that the controller in the DIFANE implementation is not involved in the process of installing new flow rules and the network state is distributed among authority switches. As a result, DIFANE-based OpenFlow switches are enabled to detect and learn topology changes, without relying on the controller updates. Both of these approaches rely on putting intelligence in the switches, which violate the philosophy of clear control and data plane separation of SDN.

AVANT-GUARD [98] is a proxy-based solution with an idea similar to DevoFlow and DIFANE, in which the OpenFlow protocol is being extended. The main purpose of AVANT-GUARD is to enhance OpenFlow networks' scalability and resilience under control plane saturation attacks such as TCP-SYN flooding. It adds intelligence to the data plane that enables security applications to respond to network threats dynamically. In AVANT-GUARD, OpenFlow switches are essentially capable of performing forensic analysis on network traffic and inspecting TCP sessions prior to notifying the controller. Thus, only flow requests that include a complete TCP handshake are forwarded to the controller.

As mentioned above, these methods require significant changes to SDN and OpenFlow, and propose moving away from the idea of separating control and data plane, which is an essential concept in SDN.

3.2.3 Security of the Application Layer

The application layer, i.e. the application plane in SDN environment is basically a set of software applications. They are programmed and designed to perform network policy decisions and provide various network services such as Quality of Service (QoS), Traffic Engineering (TE), Intrusion Detection System (IDS), network security monitoring, firewalling, and load-balancing [99]. SDN applications directly interact with the control plane through

the *Northbound* interface and are enabled to manipulate the behaviour of underlying network devices and network functions. In the following, we discuss key SDN applications that provide network security services.

CloudWatcher [100] is an OpenFlow-based monitoring application for large-scale networks. It relies on network traffic analysis technique for offering security and filtering incoming packets. Based on pre-installed security policies and controlling network flows, CloudWatcher re-routes network packets to dedicated network security devices for inspection.

FLOWGUARD [101] is an OpenFlow-based firewall application for detecting global policy violations in real-time and automatically resolving any discovered conflicts between all firewall policies. It actively monitors the flow status in the data plane and records the source and destination of each flow to enforce one consistent global behaviour across the network. When a contradiction occurs with the firewall policies, FLOWGUARD does not simply block and reject the update. Instead, it analyses the context of conflicts and acts accordingly.

Even though the OpenFlow protocol dramatically simplifies the design and the integration of complex network security applications and reduces the complexity of security policy management, the current security policy enforcement in SDN is somewhat inefficient and limited. In particular, malicious OpenFlow applications can contradict and override flow rules created by other OpenFlow applications, which can result in allowing malicious traffic to pass through the firewall. In the following, we discuss recent research focused on addressing this problem, exclusively on security threats raised from malicious applications and address the proposed security policy enforcement.

FortNOX [102] was introduced as a role-based authorisation and security constraint kernel directly integrated into the SDN control plane, offering a security mediation and policy enforcement. The essential aim of the new mechanisms incorporated into FortNOX is to detect potential flow rule contradictions within the data plane in real-time, through regularly checking and analysing the flow table after every update. When a contradiction occurs within a flow rule, FortNOX verifies this rule with flow rules in the security constraints table and applies the rule with the highest priority.

FRESCO [103] is an OpenFlow-based security application development platform specifically

designed to efficiently facilitate and enhance the implementation of OpenFlow-based security applications through defining high-level security policies and creating innovative security functions for threat detection and mitigation. FRESCO has access to all network events and flow statistics, and attack detection is based on a comparison of current network state with a database of historical network state information. When a threat is detected, the system reacts with actions such as *mirror*, *redirect* or *quarantine*. In case the action is *mirror*, the receiving switch explicitly creates a copy of the packet and forwards it to the packet analysis system for conducting additional analysis. If the action is *redirect*, the packet is forwarded directly to the destination host. However, if the action is *quarantine*, the packet can only traverse to certain hosts that are equipped with special tools to keep the infected host isolated from the network. This results in minimising the severity of the threats and simplifies the management of network security functions in SDN.

3.2.4 Security of the Southbound Interface

The OpenFlow control channel, i.e. the *Southbound Interface*, is the interface that links OpenFlow switches to an SDN controller to exchange control messages. Typically, the connection between the SDN controller and OpenFlow switches is encrypted using Transport Layer Security (TLS). The essential benefit of the TLS protocol is preventing attackers from being able to take over the control of network switches without having the required authority. As described in the OpenFlow specification [2], this security feature is optional and there is no official standard defined for the TLS protocol in the OpenFlow implementation. Unfortunately, some vendors of switches and controllers neglect to enforce the use of this protocol.

The lack of adopting of secure cryptographic protocols to the southbound interface makes SDN susceptible to the following types of attacks [83, 104]:

- Man-in-the-middle (MITM) attacks, where the attacker actively intercepts the control messages such as flow rules and applies desired changes before reaching the data plane.
- Denial of Service (DoS) attacks, where the attacker mainly aims to break the connection between the control plane and the data plane and disrupt the installation of flow

rules.

- Eavesdropping attacks, where the attacker sniffs valuable information such as the network topology.
- TCP-level attacks, where the attacker exploits vulnerabilities of TLS and floods the OpenFlow channel with attack packets.

Overall, these vulnerabilities leave avenues for an attacker to proactively and reactively manipulate the configuration of the switches, which ultimately results in modifying or disrupting the network behaviour. It is therefore absolutely critical that the control channel is secured via cryptographic protocols, which provide secure authentication, encryption and integrity protection of control messages.

Table 3.1 summarises some of the key proposals and works that discuss potential security threats of SDN technology, at the control plane, data plane, application layer, indicating the key contribution as well as the proposed attack mitigation methods.

None of the works discussed in this chapter have paid close attention to the potential security vulnerabilities of the fundamental components of SDN, such as Topology Discovery, ARP handling and network virtualisation, which represent the key focus and contribution of this thesis. More closely related works to these specific aspects are discussed in the context of the corresponding chapters, where the contributions and technical details of this thesis are presented.

Table 3.1: Summary of Proposed Security Measures for SDN Layers (Planes)

SDN layer (Plane)	Security Projects	Main Contribution	Mitigation Technique
Control Plane	Hybrid controller [81]	Evaluating OpenFlow controller performance	Add mechanisms to understand the traffic behaviour and set path on-demand
	DDoS Attacks Detection [105]	Intrusion detection and prevention system	Use a window size and threshold
	Dynamic Controller [106]	Traffic management for multiple controllers	Use integer linear programming
	ROSEMARY[107]	Secure and high performance SDN controller	Use resource utilisation monitoring
Data Plane	FlowChecker [108]	Configuration Analyser of OpenFlow switches	Use binary decision diagram
	Monitoring Model [109]	Monitoring function on OpenFlow switches	Place a general message generator and processing function
	Packet-In Filtering [110]	Filtering mechanism on OpenFlow switches	Extend the OpenFlow specification
	Resonance [111]	Dynamic access control on OpenFlow switches	Based on policies the controller installs
Application Plane	PermOF [112]	Resource isolation and access control	Minimise SDN application privileges
	Veriflow [113]	Flow rules checker	Use prefix tree(an ordered tree data structure)
	FLOVER [114]	Flow rules checker	Utilise the satisfiability modulo theories
	Assertion [115]	SDN application debugger	Use VeriFlow verification algorithm
	OF-testing [116]	Automating the testing of SDN applications	Use simple traffic models

Chapter 4

Security of Topology Discovery

4.1 Introduction

In the SDN architecture, it is essential for higher layer services such as routing to have an accurate and up to date view of the network topology. One of the key network services of a Network Operating System (NOS), i.e. SDN control layer, is to provide network topology information to the application layer.

While there is no official standard for an SDN topology discovery mechanism, there is a de-facto standard, which is sometimes informally referred to as Open Flow Discovery Protocol (OFDP) [117, 118]. All major SDN controllers implement it in essentially the same way, most likely due to the fact that it has been adopted from NOX, the original SDN controller [48].

The problem with OFDP is that it is fundamentally insecure, as is demonstrated in this chapter. We show how an attacker can poison the topology view of the SDN controller and create spoofed links by crafting special control packets and injecting them into the network via one or more compromised hosts.

We show the feasibility of the attack, both via network emulation as well as test-bed experiments, for both POX [33] and Ryu [34], two widely used SDN controller platforms. We further demonstrate and evaluate the impact of the link spoofing attack on higher layer services. We use shortest path routing as a case study and show that an attacker can relatively

Preamble	Dst MAC	Src MAC	Ether-type: 0x88CC	Chassis ID TLV	Port ID TLV	Time to live TLV	Opt. TLVs	End of LLDPDU TLV	Frame check seq.

Figure 4.1: LLDP Frame Structure

easily cause significant disruption of network connectivity. The impact, i.e. the level of connectivity disruption is quantified in two example scenarios. A final contribution of the chapter is the discussion and evaluation of countermeasures against the vulnerability.

The remainder of the chapter is organised as follows. Section 4.2 describes in detail the operation of OFDP, the current state-of-the-art topology discovery mechanism in SDN. Section 4.3 describes the vulnerability of OFDP and demonstrates a number of link spoofing attacks. Section 4.4 investigates the impact of the attack on higher layer services, based on the example of routing. Section 4.5 discusses countermeasures, Section 4.6 discusses related works, and Section 4.7 concludes the chapter.

4.2 OpenFlow Discovery Protocol (OFDP)

Topology discovery is an essential service in SDN and underpins many higher layer services. In this context, when we refer to topology discovery, we mean *link discovery*, since the controller learns about the existence of network nodes (switches) by other means, as discussed previously.

OpenFlow switches themselves do not support any topology (link) discovery functionality, and it, therefore, needs to be implemented as a service at the controller. There is currently no official standard for topology discovery in SDNs based on OpenFlow. However, there is a de-facto standard, since all the major SDN controller platforms implement topology discovery in essentially the same way, derived from the topology discovery mechanism in NOX [48]. This mechanism is sometimes referred to informally as OpenFlow Discovery Protocol (OFDP) in [118, 119], and for lack of an official term, we use it in this chapter.

OFDP uses the frame format defined in the Link Layer Discovery Protocol (LLDP) [120], designed for link and neighbour discovery in Ethernet networks. However, except the frame

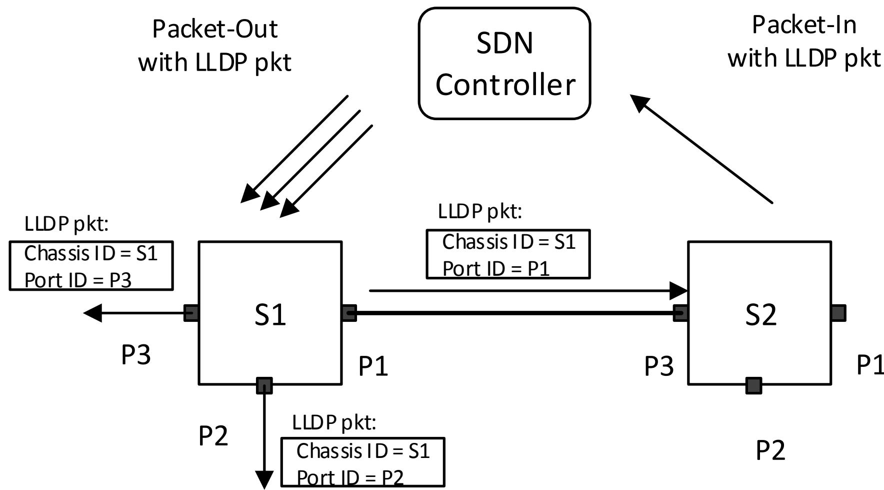


Figure 4.2: Basic OFDP Example Scenario

format, OFDP has not much in common with LLDP and operates quite differently.

The format of an LLDP frame, as used in OFDP, is shown in Figure 4.1. The LLDP payload is encapsulated in an Ethernet frame with the *EtherType* field set to 0x88CC. The Ethernet frame contains an LLDP Data Unit (LLDPDU) (shaded in grey in Figure 4.1), which has a number of type-length-value (TLV) fields. The mandatory TLVs include *Chassis ID*, a unique switch identifier, *Port ID*, a port identifier, and a *Time to live* field. These TLVs can be followed by a number of optional TLVs and an *End of LLDPDU* TLV.

An OpenFlow switch, due to its lack of control intelligence and autonomous operation, cannot initiate the sending and processing of link discovery packets, as is the case for traditional Ethernet switches, and as specified in the LLDP standard [120]. In SDN, link discovery is initiated by the controller. How this works in OFDP is illustrated via a basic example scenario shown in Figure 4.2. Initially, the SDN controller creates a dedicated LLDP packet for each port on each switch, in our example, a packet for port *P1*, a packet for port *P2* and one for port *P3* on switch *S1*. All these LLDP packets have their *Chassis ID*, and *Port ID* TLVs initialised accordingly.

The controller then uses a separate OpenFlow *Packet-Out* message to send each of the LLDP packets to switch *S1*. Every OpenFlow *Packet-Out* message also includes an *action*, which instructs the switch to forward the packet via the corresponding port. For example, the LLDP packet with *Port ID = P1* will be sent out on port *P1*, the packet with *Port ID = P2* on port *P2*, etc.

Switches are pre-configured with a rule which states that any received LLDP packets are to be sent to the controller via an OpenFlow *Packet-In* message. As an example, we consider the LLDP packet which is sent out on port *P1* on switch *S1* and is received by switch *S2* via port *P3* in Figure 4.2. According to the pre-installed rule, switch *S2* sends the LLDP packet to the controller, encapsulated in an OpenFlow *Packet-In* message. This OpenFlow *Packet-In* message also contains additional metadata, such as the ingress port where the packet was received, as well as the *Chassis ID* of the switch sending the OpenFlow *Packet-In* message. This information, combined with information about the origin switch and port, contained in the payload of the LLDP packet (*Chassis ID* and *Port ID* TLVs) can be used by the controller to infer the existence of a link between *(S1, P1)* and *(S2, P3)*.

This process is repeated for every switch in the network, i.e. the controller sends a separate OpenFlow *Packet-Out* message with a dedicated LLDP packet for each port of each switch, allowing it to discover all available links in the network.[1] The entire process is repeated continuously, with a typical discovery interval of 5 seconds [33].

Most current SDN controller platforms such as NOX [48], POX [33], Ryu [34], ONOS [35], OpenDaylight [37], Floodlight [36], and Beacon [121] implement the OFDP discovery mechanism as described above. A study of the source code of the different implementations reveals only very minor variations, for example in regards to the timing of the sending of the *Packet-Out* messages, or the encoding of *Chassis ID* and *Port ID* information in LLDP packets.

4.3 OFDP Link Spoofing - Basic Vulnerability

The basic security problem with the current SDN topology discovery mechanism (OFDP) is that there is no authentication of LLDP control messages. Any LLDP packet received by the controller is accepted, and link information contained in it is used to update the controller's topology view.

More specifically, OFDP lacks the following two checks:

[1] The authors in [117] have demonstrated how the efficiency of this approach can be significantly improved.

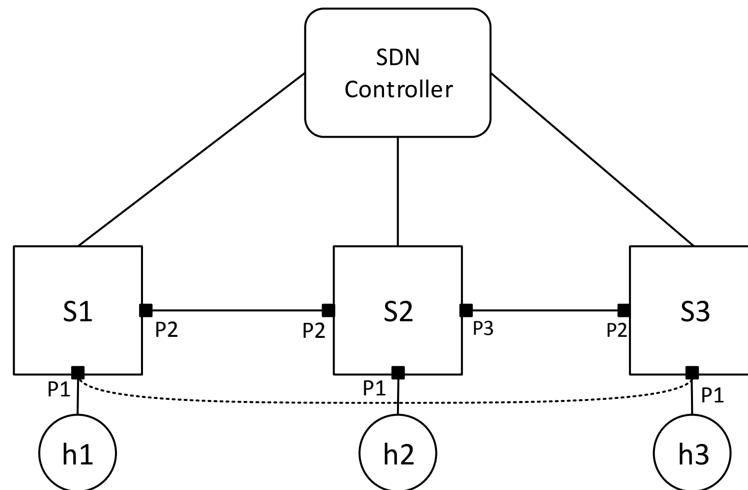


Figure 4.3: Basic Attack Scenario (Mininet)

- OFDP does not check or enforce that only LLDP packets received via switch ports connected to another switch are accepted for processing. Instead, LLDP packets from host ports are also accepted and forwarded to the controller.
- There is no authentication or integrity check of LLDP control messages. The controller has no way of verifying the origin of the packets.

As a result, it is relatively easy for an attacker to inject fabricated (spoofed) LLDP control messages into the network and thereby corrupting the topology information of the controller. We illustrate this via a simple example, shown in Figure 4.3. In this scenario, we assume that host *h1* has been compromised by an attacker, who aims to create a fake link between switches *S1* and *S3*.

The attack can be broken down into the following steps:

1. Host *h1* injects an LLDP packet via port *P1* on switch *S1*, where *h1* is attached. The injected packet follows the structure shown in Figure 4.1, but with the *Chassis ID* TLV set to *S3*, and the *Port ID* set to *P1*.

2. Switch *S1* receives the LLDP packet from *h1*, and following its installed rule, it forwards the packet to the controller, encapsulated in an OpenFlow *Packet-In* message. Switch *S1* adds information to the OpenFlow *Packet-In* message, i.e. its own *Chassis ID* and

Table 4.1: Software Tools used for Implementation and Experiments in Chapter 4

Software	**Function**	**Version**
Mininet [30]	Network Emulator	2.1.0
OFELIA [32]	Hardware SDN Test-bed	=======
Open vSwitch [31]	Virtual SDN Switch	2.0.2
POX [33]	SDN Controller Platform	*dart* branch
Scapy Library [38]	Packet Manipulation Tool	2.2.0

the *Port ID* of the ingress port via which the LLDP packet was received at switch *S1*. In our scenario, this information is *(S1, P1)*.

3. The controller receives the LLDP packet plus the additional information added by switch *S1*. It identifies the source of the LLDP packet, and therefore the origin of the link from the TLVs in the payload is *(S3, P1)*. The information about the other end of the link is taken from the metadata of the *Packet-In* message, and is identified as *(S1, P1)*. From this information, the controller concludes (wrongly) that there exists a link between *(S3, P1)* and *(S1, P1)*.

4.3.1 Experimental Validation - Mininet

To validate the feasibility of the link spoofing attack experimentally, we used Mininet [30] and Open vSwitch (OVS) [31]. For our initial experiment, we used the POX controller platform and its implementation of OFDP, i.e. the *openflow.discovery* component. We wrote a packet generator in Python based on the Scapy library [38] to craft a special LLDP packet for the attack. Table 8.2 summarises the relevant software tools that we used in the experiments of this chapter. The Mininet experiments were run on a standard Dell PC (OptiPlex 780 with a 3 GHz Intel Core 2 Duo CPU and 4 GB of RAM), running Ubuntu Linux with kernel version 3.13.0.

Figure 4.4 shows the debug output of the POX controller, in particular the *openflow.discovery* component which implements OFDP. No other POX component was running in this experiment. From the output, we see that our three switches have connected to the controller,

```
root@mininet-vm:~/pox# ./pox.py openflow.discovery
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
INFO:openflow.of_01:[00-00-00-00-00-03 3] connected
INFO:openflow.discovery:link detected: 00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-02.3 -> 00-00-00-00-00-03.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.2 -> 00-00-00-00-00-02.3
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.1 -> 00-00-00-00-00-01.1
```

Figure 4.4: POX Debug Information (Mininet)

with *Chassis ID* of *00-00-00-00-00-01* for switch *S1*, *00-00-00-00-00-02* for switch *S2* and *00-00-00-00-00-03* for switch *S3*. This debug output is generated by the main POX component.

The last five lines of the output are from the *openflow.discovery* component. Each line indicates the detection of a unidirectional link, caused by the reception of a corresponding LLDP packet at the controller. For example, the first of these lines indicates that a link from *(S1, P2)* to *(S2, P2)*, i.e. from port *P2* on switch *S1* to Port *P2* on switch *S2*, has been detected. The next line indicates a link from *(S2, P3)* to *(S3, P2)*. The following two lines indicate the detection of the same links in the reverse direction. This is consistent with our topology, as shown in Figure 4.3.

The interesting part in Figure 4.4 is the last line (in bold), which appears after we run the attack by injecting the fabricated LLDP packet from host *h1* to switch *S1*. The line indicates that a non-existent link from *(S3, P1)* to *(S1, P1)* is detected by the controller, and hence the link spoofing attack has been successful. When we look at the topology view of the controller, which is stored as a list of links, we see that the link has indeed been added.

It is important to note that the attacker can spoof the origin of the link (switch and port) arbitrarily, simply by setting the relevant LLDP TLVs accordingly. However, the link destination information is added as metadata to the OpenFlow *Packet-In* message by the ingress switch, and hence cannot be changed by the attacker. For our example, this means that the spoofed links, which host *h1* can create are, limited to the set of unidirectional links terminating at port *P1* on switch *S1*.

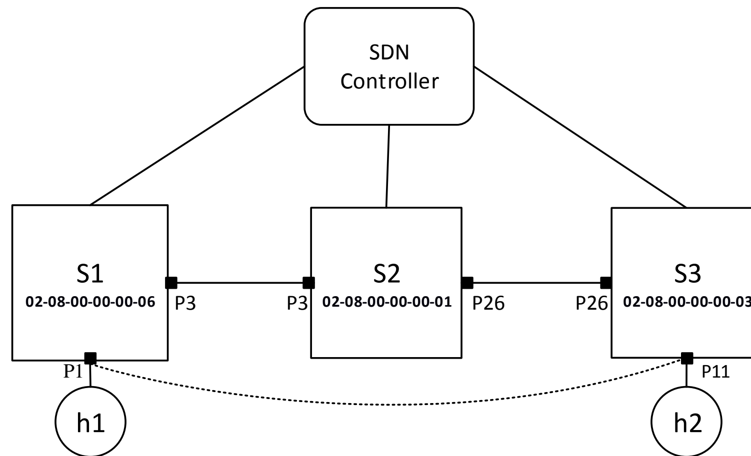


Figure 4.5: Basic Attack Scenario (OFELIA)

If an attacker wants to create a spoofed bidirectional link, for example, from switch *S1* to switch *S3* in our scenario, the attacker needs to control both hosts *h1* and *h3*. We discuss this in more detail in Section 4.4.

We also performed the above attack with the Ryu SDN controller, with identical results. However, we had to slightly modify our LLDP packet generation script, since Ryu uses a different encoding format for the *Chassis ID* and *Port ID* values in the LLDP packet.

4.3.2 Experimental Validation - OFELIA

In addition to our emulation based experiments using Mininet, we also conducted the same experiment on the OFELIA SDN test-bed [32]. Our goal was to replicate the topology shown in Figure 4.3. We managed to do this, with the exception of a small detail. OFELIA provides only three virtual machines to experimenters, of which one is used for the controller, leaving only two for the use as hosts. Figure 4.5 shows our OFELIA topology, with the corresponding *Chassis ID*s and *Port ID*s. The main difference to Figure 4.3 is that the host attached to switch *S2* is missing, which is not a problem, since it does not play an active role in this attack scenario.

After configuring the topology, we conducted the same experiment as discussed in Section 4.3.1, with the same POX controller code and configuration. We also used the same packet injection code, with a small modification to account for the different interface name

```
root@POX:/ofelia/users/██████████/pox# ./pox.py openflow.discovery openflow.debug POX
0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[02-08-00-00-00-06|520 1] connected
INFO:openflow.of_01:[02-08-00-00-00-01|520 3] connected
INFO:openflow.of_01:[02-08-00-00-00-03|520 2] connected
INFO:openflow.discovery:link detected: 02-08-00-00-00-06|520.3 -> 02-08-00-00-00-01|520.3
INFO:openflow.discovery:link detected: 02-08-00-00-00-01|520.3 -> 02-08-00-00-00-06|520.3
INFO:openflow.discovery:link detected: 02-08-00-00-00-01|520.26 -> 02-08-00-00-00-03|520.26
INFO:openflow.discovery:link detected: 02-08-00-00-00-03|520.26 -> 02-08-00-00-00-01|520.26
INFO:openflow.discovery:link detected: 02-08-00-00-00-03|520.11 -> 02-08-00-00-00-06|520.1
```

Figure 4.6: POX Debug Information (OFELIA)

on host *h1*.

Figure 4.6 shows the debug output from POX. We see that bidirectional links are created between port *P3* on switch *S1* (*02-08-00-00-00-06*), and port *P3* on switch *S2* (*02-08-00-00-00-01*), as well as between port *P26* on switch *S2* (*02-08-00-00-00-01*), and port *P26* on switch *S3* (*02-08-00-00-00-03*). The last line again indicates that the attack was successful, and there was a link created between switch *S1* and switch *S3*. We have also verified the creation of the fake link in the controller's topology view. As with Mininet, we have replicated the attack for Ryu, with the same results.

Our experiments have demonstrated the basic vulnerability of OFDP, the predominant SDN topology discovery mechanism. As mentioned earlier, topology discovery is an essential network service provided in SDN, upon which a lot of other services and applications rely. In the following section, we discuss the potential impact of the vulnerability on such higher layer services, using the example of shortest path routing.

4.4 Impact on Routing

Routing is a key network application that relies on the controller having an up to date and accurate topology view. Shortest path routing in SDN is relatively trivial, compared to traditional networks. The challenge of dealing with a physically distributed system is done by the topology discovery component, implemented by the controller platform. Given a network topology as a graph, shortest path routing is essentially just computing the shortest path between the source and destination nodes, e.g. via Dijkstra's algorithm [122]. In the following,

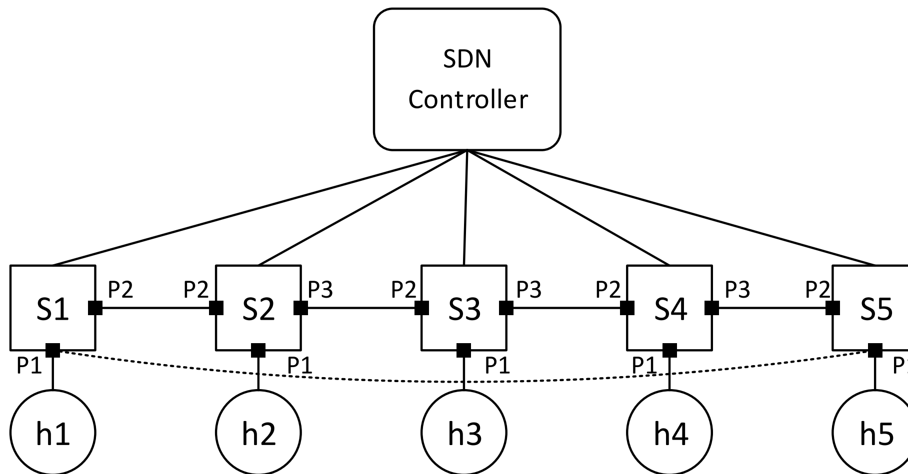


Figure 4.7: Linear Topology for Routing Experiment

we evaluate and quantify the impact of the attack on routing via two topology examples.

4.4.1 Linear Topology

For our next experiment, we consider a simple linear topology with five switches and a single host attached to each switch, as shown in Figure 4.7. As before, we assume that host *h1* is the attacker, which in this case injects a fabricated LLDP packet with the aim of creating a false (unidirectional) link between *(S5, P1)* and *(S1, P1)*. This spoofed link is shown as a dashed line in Figure 4.7. As described in the previous section, the attacker simply needs to set the *Chassis ID* to *S5*, and the *Port ID* to *P1* in the fabricated LLDP packet for this attack.

We created this topology in Mininet and used the layer 2 shortest path routing component in POX (*l2_multi.py*) for our experiment. This POX component computes shortest paths between node pairs using the Floyd-Warshall algorithm [123].

Prior to launching the attack, we performed a ping test among all host pairs to verify the connectivity. This was achieved via the *pingall* command in Mininet. We saw that we had 100% connectivity, and each host could reach every other host. After launching the attack from host *h1*, which injected the fabricated LLDP packet, we verified that the topology discovery service had indeed added a link from *(S5, P1)* to *(S1, P1)* to the controller's topology database.

Table 4.2: Connectivity After Attack

ping src \ ping dst	h1	h2	h3	h4	h5
h1		1	1	0	0
h2	1		1	1	0
h3	1	1		1	1
h4	0	1	1		1
h5	0	0	1	1	

However, after running *pingall* again, we still saw a connectivity of 100%. Inspection of the source code of *l2_multi.py* reveals that only bidirectional links are considered for path computation, which is why the attack was unsuccessful.

In this case, to be able to disrupt network connectivity, the attacker needs to spoof a bidirectional link. This is impossible to achieve with control over only a single host, due to the fact that only one end point of the spoofed link, i.e. the source, can be chosen by the attacker. The other end of the link is determined by the ingress port and switch where the LLDP packet is injected by the attacker. Another way of expressing this is that an attacker controlling a single host can create unidirectional links starting at any source, but they all have to terminate at the switch and port, where the attacking host is connected to.

To establish a bidirectional link, the attacker needs to control at least two hosts. In this new attack scenario, we assume the attacker has compromised and controlled hosts *h1* and *h5*. As in the previous case, *h1* injects an LLDP packet with the *Chassis ID and Port ID* set to *S5* and *P1*, creating the unidirectional link from *(S5, P1)* to *(S1, P1)*. In addition, *h5* injects an LLDP packet with the source set to *(S1, P1)*, creating the link in the reverse direction.

We ran the pairwise ping test again, and the result is shown in Table 4.2. The leftmost column indicates the source, and the topmost row indicates the destination of the ICMP echo request message sent by *ping*. A '1' in the table indicates that there is connectivity between the two hosts in the corresponding row and column, and a '0' indicates a lack of connectivity. We can see that connectivity between a pair of hosts is disrupted when the spoofed link *(S1-S5)* is part of the shortest path between the two hosts, as expected.

This attack on the topology discovery mechanism significantly disrupts connectivity for routing. In this scenario, almost 30% of all links are disrupted by the creation of a single spoofed bidirectional link. We have also replicated the experiment in OFELIA, with an identical outcome.

While often shortest path routing assumes bidirectional links, such as the POX component we have considered here, this is a design decision rather an absolute requirement. Other implementations, such as [124], also consider unidirectional links in their computation of shortest paths. In this case, the attacker only needs to control a single host and can disrupt connectivity via creating unidirectional spoofed links.

4.4.2 Tree Topology

We also considered a tree topology, as shown in Figure 4.8. The topology consists of 15 switches, organised in a binary tree of depth 4 and fan-out 2, with a host attached to each leaf switch. In this experiment, we tried to quantify the connectivity disruption impact of the link spoofing attack.

We created bidirectional fake links between all host pairs, one at the time. For each of those scenarios, i.e. for each case with a different fake link in the network, we ran a complete ping test (pingall) between all host pairs, resulting in a total of 56 pings. We then considered how many of those pings failed, due to the creation of each individual fabricated link.

The results are shown in Table 4.3, which shows the percentage (rounded to the nearest integer) of connectivity loss, i.e. failed pings, due to each of the possible fabricated links. For example, we see that if a single fake link is generated between hosts *h1* and *h8*, 29% of the connectivity between host pairs is disrupted.

We observe that there are only three distinct values in the table, 4%, 11% and 29%, which correspond to 3 different scenarios in our example. We get 4% (rounded) in the case where the fake link is between 2 hosts that are attached to the same switch at level 3 in the topology, for example hosts *h1* and *h2*, or hosts *h3* and *h4*, etc. In this case, the spoofed link forms part of the shortest path for only 2 out of the total of 56 ping paths, i.e. from *h1* to *h2*, and *h2*

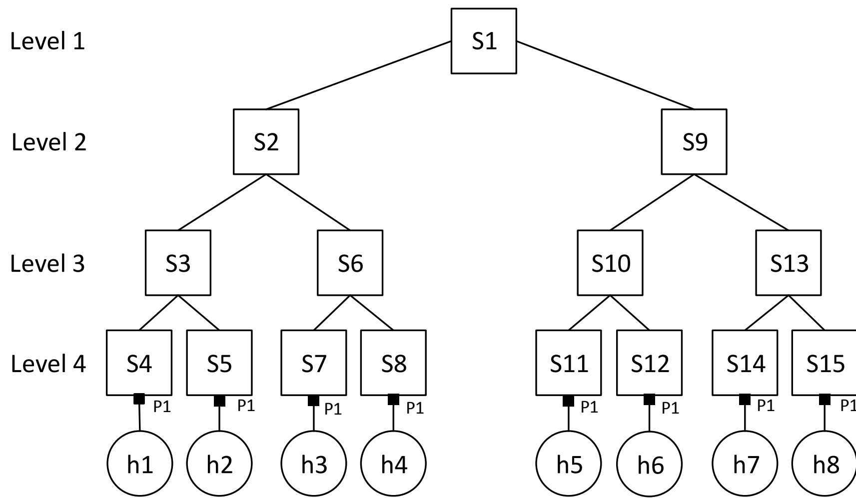


Figure 4.8: Tree Topology for Routing Experiment

Table 4.3: Connectivity Loss Due to Link Spoofing Attack

fake link src \ fake link dst	h1	h2	h3	h4	h5	h6	h7	h8
h1		4%	11%	11%	29%	29%	29%	29%
h2	4%		11%	11%	29%	29%	29%	29%
h3	11%	11%		4%	29%	29%	29%	29%
h4	11%	11%	4%		29%	29%	29%	29%
h5	29%	29%	29%	29%		4%	11%	11%
h6	29%	29%	29%	29%	4%		11%	11%
h7	29%	29%	29%	29%	11%	11%		4%
h8	29%	29%	29%	29%	11%	11%	4%	

to *h1*.

For the case where the fake link is between two hosts connected via a common switch at level 2 of the topology, e.g. hosts *h1* and *h3* via switch *S2*, we observe that the fake link is part of the shortest path of 6 out of the total of 56 source-destination pairs, resulting in a loss of 11% connections.

Finally, if a fake link is created between hosts that are located in different main branches of the tree, i.e. if they are connected via switch *S1* at level 1 of the topology, 29% (16 out of 56) connections are disrupted. This is due to the fact that the spoofed link provides a shorter path or shortcut for a larger number of source-destination pairs.

The exact numbers are obviously specific to this particular example. However, it demonstrates the potential power of the link spoofing attack in terms of connectivity disruption for routing in SDN. By creating multiple spoofed links, the impact would obviously be compounded. The example also shows that by targeting the location of the spoofed link(s) in the network topology, an attacker can maximise the impact.

4.4.3 Discussion

We have demonstrated that the vulnerability of OFDP translates into a vulnerability of routing, which relies heavily on the topology discovery service provided by the controller. An attacker can relatively easily disrupt network connectivity since the attacker only needs to gain control over a small number of hosts. This is typically much easier to achieve than gaining control over network infrastructure devices, i.e. switches or controllers. Since all the open source controller platforms that we have investigated implement OFDP in essentially the same way, the vulnerability discussed in this chapter is shared by all of them, and hence the problem is significant.

We have used the example of routing to demonstrate the potential impact of the link spoofing attack, but it is clear that other services, which rely on the topology discovery service, are also vulnerable to the attack. We have done some preliminary investigation into the spanning tree component in POX and found it to be vulnerable as well. This is critical, given the fact that many services rely on the spanning tree mechanism, such as the L2_learning switch component in POX.

4.5 Countermeasures

As mentioned previously, the vulnerability of OFDP is due to the lack of any checks about the origin of received LLDP packets. We discuss two basic approaches to address this.

4.5.1 Controller Checks

The link spoofing attack, as presented in this chapter, could be rendered impossible if LLDP packets were only accepted via switch ports that connected to other switches.

This check could be implemented via installing a simple rule on each switch, or alternatively, it could be performed at the controller. The problem with this approach is that it assumes knowledge about each port on each switch, and to what type of node it is connected to. In a network with a dynamic topology, there is no simple and secure way to keep track of that, to the best of our knowledge. For example, services that keep track of hosts, such as the *host_tracker* component in POX, are themselves vulnerable to attacks [125], and hence cannot be assumed to provide reliable information.

4.5.2 LLDP Packet Authentication

The problem of lacking the authenticity of LLDP messages can relatively easily be overcome by adding a cryptographic Message Authentication Code (MAC) to each LLDP packet that the controller sends out. Such a MAC provides authentication as well as integrity for each packet. We have implemented this mechanism in POX using a Hash-based Message Authentication Code (HMAC) [126]. The MAC is computed as follows:

$$HMAC(K, m) = h((K \oplus opad)|h(K \oplus |ipad)|m)$$

K is the secret key, and m is the message over which the HMAC is calculated. In our case, m consists of the relevant LLDP TLVs, i.e. the *Chassis ID* and the *Port ID*. $h()$ is a cryptographic hash function, '$|$' denotes concatenation and '$\oplus$' denotes the XOR operation. $opad$ and $ipad$ are constant padding values [126].

It is important to note that the basic HMAC is vulnerable to replay attacks. In the scenario shown in Figure 4.3, a replay attack can be used against a topology discovery mechanism protected with a basic HMAC. The attack requires control over two hosts, e.g. hosts *h1* and *h3*. As part of the normal OFDP protocol, host *h1* will receive LLDP packets with *Chassis ID* set to *S1* and *Port ID* set to *P1*. Here, we assume that the LLDP packet is secured with

a HMAC, computed over the relevant LLDP TLVs, using the secret key K. Host *h1* can then send this LLDP packet to its colluding partner host *h3* via an out-of-band channel. Host *h3* then injects the packet to switch *S3* via port *P1*. Since the packet has a valid MAC, the controller accepts it, and a spoofed link from *(S1, P1)* to *(S3, P1)* is successfully created. The reverse link can be created in the same way. We have implemented this attack and verified its feasibility.

The traditional approach to prevent replay attacks in the HMAC is via the use of a unique message identifier (or *nonce*) over which the HMAC is computed, to ensure that each HMAC value is unique. This message identifier needs to be sent as cleartext to the receiver as part of the message, causing additional overhead.

We, therefore, use an alternative approach in our implementation. Instead of using a unique message identifier, we replace the static secret key K with a dynamic value $K_{i,j}$, which is randomly chosen for every single LLDP packet i, in every topology discovery round j. The best chance for an attacker to compute a valid MAC and launch a successful link spoofing attack is via guessing the correct value of the random numbers $K_{i,j}$. This is virtually impossible if we use a high-quality random number generator that provides sufficient entropy. Any wrong guess by an attacker can easily be detected by the controller.

To verify the authenticity of a received LLDP packet and compute its HMAC value, the controller needs to know the corresponding value of $K_{i,j}$. This is achieved by the controller keeping track of which key is used for which packet. The combination of *Chassis ID* and *Port ID* provides the necessary identifier.

We used MD5 as our hash functions. While MD5 has been shown to be vulnerable to a range of collision attacks, it can still be considered sufficiently secure in the context of HMAC [126], since HMAC does not rely on the collision resistance property [127]. [2]

We have implemented this HMAC based mechanism in the topology discovery component in POX. To accommodate the MAC, we defined a new, optional TLV in the LLDP packet. We have conducted extensive tests and have verified that OFDP with the added HMAC (OFDP_HMAC) is indeed able to detect the injection of any fabricated LLDP packets from

[2]It would obviously be trivial to replace MD5 with another hash function such as SHA3.

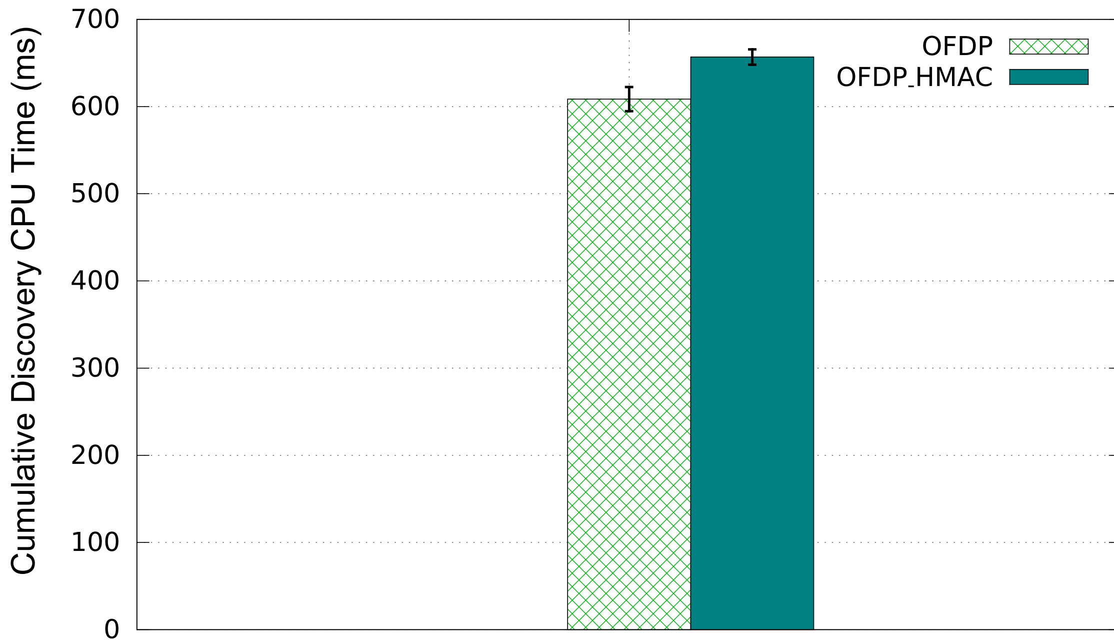


Figure 4.9: Computational Overhead of HMAC in OFDP

an attacker.

When creating an LLDP packet, the controller chooses a value N, computes HMAC, and adds the result to the TLV.

Whenever the controller receives an LLDP packet via an OpenFlow *Packet-In* message, it computes the MAC over the relevant TLVs in the LLDP packet, using the corresponding value of $K_{i,j}$. If the computed value matches the MAC in the packet, the authenticity of the packet is validated, and its content is used to update the controller's topology view. If not, the packet is discarded, and an alarm can be raised.

We have also evaluated the computational overhead on the controller caused by this mechanism. For this, we used a 21 node tree topology in Mininet, with depth 2 and fan-out 4, and ran both the original OFDP mechanism in POX, as well as OFDP_HMAC. Figure 4.9 shows the total cumulative controller CPU time used by each version, over an experiment period of 300 seconds. The experiment was repeated 20 times, and the figure shows the 95% confidence interval. The absolute values are not that interesting since they are hardware dependent. However, we see that in relative terms, the overhead of HMAC adds an extra 8% in CPU load to the low computational cost of the topology discovery mechanism. We believe this is an acceptable cost to pay for the increased level of security.

4.6 Related Works

There have been a number of works that address various security aspects of SDN. However, only very few recent papers have addressed the security of the core SDN service of topology discovery.

The paper [128] discusses a range of attacks against SDN, and proposes *SPHINX*, a generic SDN attack alert system, which compares network behaviour with predefined or learned 'normal' behaviour, defined as policies. The paper also mentions the possibility of attacks against topology discovery via spoofing of LLDP packets, as discussed in this chapter. The paper does not address the specific technical details of the attack, such as provided in this chapter, nor does it explore and quantify the impact of the attack on network connectivity.

In [125], the authors also discuss a range of attacks against SDNs, including ARP spoofing attacks as well as link spoofing attacks. Due to the wide scope of the paper, it does not specifically consider, evaluate or quantify the impact of the link spoofing attack on routing and hence network connectivity, as we have done in this chapter. Furthermore, in contrast to our work, the experiments in [125] are limited to software switches only. The authors of [125] also discuss potential countermeasures against the link spoofing attack, and also suggest a HMAC based packet authentication mechanism. However, their proposed method uses a static secret key, without a nonce, for the computation of the HMAC, and is therefore vulnerable to replay attacks, as discussed in the previous section.

4.7 Conclusions

Topology discovery is an essential service in SDN, and a variety of other services and applications, such as routing, rely on it. In this chapter, we have discussed OFDP, the current de-facto standard of topology discovery in SDN, implemented by most SDN controller platforms. We have discussed the vulnerability of OFDP to link spoofing attacks, which only requires an attacker to have control over one or more hosts (physical or virtual) in the network.

Through experiments in both emulated network scenarios using Mininet, and an SDN test-bed (OFELIA), we have demonstrated the feasibility of the attack. We have shown that the controller's topology view can be successfully poisoned by the attacker, and non-existent links can be added to the controller's topology database. We have further evaluated the impact of this attack on the operation of an SDN, in particular with the example of routing.

Finally, we discussed potential countermeasures and implemented a simple mechanism that provides authentication using a Hash-based Message Authentication Code (HMAC). We have also discussed and verified its ability to prevent the link spoofing attack demonstrated in this chapter. Finally, we have measured the computational cost of implementing the measure.

Chapter 5

Security of Address Resolution Protocol

5.1 Introduction

The resolution of network layer addresses to link-layer addresses is an essential function in packet switched networks. In IPv4 networks, this service is provided by the Address Resolution Protocol (ARP) [21]. When a host wants to send a frame to another node on the local network, it broadcasts an ARP request, specifying the targeted destination node's IP address, as well as the source node's MAC address. Upon receiving the ARP request, the node with the specified (target) IP address, will respond with an ARP reply message, containing its own MAC address. This information is then added to the sending node's ARP cache, and the layer 2 frame can be sent.

In IPv6, this address mapping service is implemented via the Neighbour Discovery Protocol (NDP), in particular via Neighbour Solicitation (NS) and Neighbour Advertisement (NA) [129, 130], which allows the discovery of the link layer address of a node on the same local network. Similar to ARP, an IPv6 node can multicast a Neighbour Solicitation message on the local network, specifying the target IP address. In response, the node with the specified target IP address sends a Neighbour Advertisement message, containing its link-local address.

Both ARP and NDP (NS/NA) are vulnerable to spoofing or poisoning attacks, where an attacker can create false entries in a host's ARP cache in IPv4 or Neighbour Cache in the

case of IPv6 [131]. The vulnerability is due to the fact that both ARP and NDP are stateless protocols and neither of the two (in their basic version) support any cryptographic authentication mechanism. In the context of ARP, an attacker can send an ARP message (request or reply) in the local network, with a false IP-to-MAC address binding. This information is accepted by nodes, receiving the message, and their ARP cache is updated accordingly. As a result, they are poisoned. Similarly, a host can send a fabricated NDP Neighbour Solicitation or Neighbour Advertisement message, thereby associating an IP address with the layer 2 address of another host. As a result of a successful attack, packets are sent to a malicious node instead of the intended destination. This can be used to launch Denial of Service (DoS) and Man-in-the-middle (MITM) attacks. While some of the technical details vary, the basic mechanism of ARP and NDP (NS/NA) and their vulnerabilities to spoofing attacks are very similar.

In this chapter, we explore ARP spoofing detection and mitigation mechanisms in the context of SDN. The outline of the chapter is as follows. Section 5.2 discusses the relevant background of ARP, ARP spoofing, and traditional countermeasures. Section 5.3 discusses the basic approaches for ARP handling in SDN. Section 5.4 demonstrates ARP and NDP spoofing in SDN. Sections 5.5 and 5.6 present our two proposed countermeasures and their evaluation. Section 5.7 discusses related works and Section 5.8 concludes the chapter.

5.2 Background

5.2.1 Address Resolution Protocol (ARP)

Figure 5.1 shows the Ethernet frame structure of ARP packets. As can be seen in the figure, the ARP payload is encapsulated in an Ethernet frame with an *Ether Type* field of *0x0806*. The payload includes *Sender Hardware Address* (SHA) and *Target Hardware Address* (THA) fields, which are the MAC addresses of the sender and the intended receiver, i.e. target. The frame also contains the *Sender Protocol Address* (SPA) and *Target Protocol Addresses* (TPA) fields, which represent the IP addresses of the sender and the target. The *Operation* field indicates if the packet is an ARP request, with a value of 1, or an ARP reply, with a

Preamble	Dest MAC	Src MAC	Ether Type (0x0806)
Hardware Type		Protocol Type	
Hardwre Length	Protocol Length	Operation (Request 1, Reply 2)	
Sender Hardware Address (SHA)			
Sender Protocol Address (SPA)			
Target Hardware Address (THA)			
Target Protocol Address (TPA)			
Frame check sequence			

Figure 5.1: ARP Frame Structure

value of 2.

When a host wants to deliver an IP datagram as an Ethernet frame to another host on the same subnet, whose MAC address is unknown, it broadcasts an ARP request with the SHA field set to its own MAC address and the SPA field set to its own IP address. The TPA field is set to the IP address of the intended destination node, while the THA field is initialised with a dummy value (*00:00:00:00:00:00*), representing the unknown target MAC address. Each node that receives the ARP request will learn about the IP-MAC address mapping of the sending node (SHA and SPA) and will add the information to its ARP cache. Each recipient node will check if the TPA value in the request matches its own IP address, and if so, will respond with an ARP reply message.

The fields of the ARP reply message are initialised by swapping the roles of the sender and target. The SHA field is set to the MAC address of the node that received the ARP request and represents the answer to the question posed in the request. The SPA field is set to the corresponding IP address and is copied from the TPA field in the corresponding ARP request. Finally, the THA and TPA fields of the ARP reply are initialised as the SHA and SPA fields of the corresponding ARP request. The ARP reply is then unicast to the MAC address of the sender of the ARP request (THA in ARP reply = SHA in ARP request). The recipient of the ARP reply will update its ARP cache and add the newly learned SPA-SHA address mapping [132] [133]. ARP also supports *gratuitous* replies, which are unsolicited messages without a corresponding request.

We refer to the ARP handling approach discussed above as *Regular ARP*, in order to contrast it with *Proxy ARP*. Proxy ARP is a mechanism where a node, typically a router, answers ARP requests intended for another host on behalf of that node. Both Regular ARP and Proxy ARP are supported and widely used in SDN, and we cover both methods in this chapter.

5.2.2 ARP Spoofing

The basic security problem with ARP (and NDP) is that it is a stateless protocol, i.e. it treats each request or reply independently from any previous communication. As a consequence, a host will readily accept information from gratuitous ARP replies, without having sent a corresponding request. Furthermore, the ARP protocol has no mechanism to authenticate the sender of an ARP request or reply message or to check the integrity and validity of the provided information. As a result, it is relatively easy for an attacker to poison a host's ARP cache with a false IP-MAC address mapping. All the attacker needs to do is to craft an ARP message with a false SPA field. A node receiving the message will simply trust the content and update its ARP cache accordingly.

ARP spoofing attacks can be done via ARP request messages or ARP reply messages. To launch an ARP request based attack, the attacker sets the SHA and SPA fields to the desired values and broadcasts the message. As a result, the ARP cache of every node on the same subnet will be poisoned. An attacker can use either a gratuitous or non-gratuitous ARP request message for such an attack. The ARP spoofing attacks allow an attacker to redirect traffic from a target host to any arbitrary node, thereby enabling DoS or MITM attacks [23].

5.2.3 Traditional ARP Spoofing Countermeasures

Arguably, the strongest protection against ARP or NDP spoofing is via cryptographic authentication and integrity mechanisms. S-ARP [134] proposes such a cryptographic protection. It uses public key cryptography and relies on each node having a public/private key pair. The ARP packet format is extended and adds a digital signature field, providing message authenticity and integrity. S-ARP only protects ARP reply messages and therefore remains

vulnerable to ARP request based spoofing attacks. The other key limitation of this approach is its reliance on a Public Key Infrastructure (PKI), in particular a Certificate Authority (CA) for its operation. This requirement has proven to be impractical for a low-level protocol such as ARP, as is demonstrated by the lack of adoption of S-ARP and related ideas.

A similar cryptographic solution exists for the protection of the NDP (NS/NA) protocol in IPv6. The Secure Neighbour Discovery (SEND) protocol is a security extension of NDP [135]. This approach shares similar practical challenges with S-ARP, imposed by the problem of bootstrapping and key management [136].

A number of non-cryptographic ARP spoofing detection and prevention have been proposed for traditional IP networks [137, 138, 139, 140, 141, 142, 143]. A survey of these measures is provided in [22]. One of the most prominent and widely used ARP spoofing mitigation approaches is Dynamic ARP Inspection (DAI) [25]. DAI is implemented in Ethernet switches and checks ARP packets against a trusted database of IP-MAC address mappings. Any ARP packets with information (in particular SHA and SPA fields) that is inconsistent with the database is dropped. DAI uses a technique called DHCP-Snooping [144] to build and maintain its trusted IP-MAC address database. While DAI is one of the most promising non-cryptographic solutions to mitigate ARP spoofing attacks, its drawbacks include its proprietary nature and relatively high cost [22, 145].

5.3 ARP Handling in SDN

There are two basic approaches to handling ARP in SDN. The first, which we refer to as *Regular ARP*, is as discussed in Section 5.2.1. Here, a host broadcasts an ARP request on the local network, and the host that has the matching IP address specified in the TPA field responds with a unicast ARP reply containing its own MAC address. The role of SDN and in particular the controller in this context is simply to provide the packet forwarding functionality.

The second approach to handle ARP in SDN is *Proxy ARP*, which is well suited to SDN, due to its centralised control plane. Proxy ARP can easily be implemented in SDN by installing rules on each SDN switch to forward any ARP request to the controller via an OpenFlow

Packet-In message. The controller, with its global view of the network, can generate the corresponding ARP reply message with the required MAC address of the target node. It then encapsulates the ARP reply packet in an OpenFlow *Packet-Out* message and sends it to the switch, where the request was received from together with instructions (actions) to send it back to the sender of the ARP request message via the port where that request was received on.

Proxy ARP handling is implemented in most SDN controllers, including POX [33], Ryu [34], ONOS [35], Floodlight [36] and OpenDaylight [37], which we consider in our experiments. The benefit of Proxy ARP in SDN is a significant reduction in broadcast traffic that decreases the ARP response time, which we discuss in the next chapter. In this chapter, we consider the security of both ARP handling approaches in SDN, i.e. Regular ARP as well as Proxy ARP.

5.4 ARP and NDP (NS/NA) Spoofing in SDN

In order to motivate our work, we demonstrate the vulnerability of SDN to ARP and NDP (NS/NA) spoofing attacks via experiments, for both Regular and Proxy ARP.

5.4.1 Experimental Platform

For this and all our experiments in this chapter, we used Mininet [30] and Open vSwitch (OVS) [31]. We further used POX [33], Ryu [34], ONOS [35], Floodlight [36], and OpenDaylight [37] as our SDN controllers. To craft ARP packets for our attacks, we used the Scapy packet manipulation library [38] and Dsniff package [39]. Table 5.1 summaries the relevant software tools that we used in the experiments of this chapter. All our experiments were run on a PC (OptiPlex 9020 with a 3.6 GHz Intel Core i7-4790 CPU and 16 GB of RAM), running Ubuntu Linux 14.04 with kernel version 3.13.0. For our initial experiments, we use a simple scenario with three switches and three hosts shown in Figure 5.2.

Table 5.1: Software Tools used for Implementation and Experiments in Chapter 5

Software	Function	Version
Mininet [30]	Network Emulator	2.2.0
OFELIA [32]	Hardware SDN Test-bed	=======
Open vSwitch [31]	Virtual SDN Switch	2.1.1
POX [33]	SDN Controller Platform	*dart* branch
Ryu [34]	SDN Controller Platform	3.19-3.22
ONOS [35]	SDN Controller Platform	1.8.5
Floodlight [36]	SDN Controller Platform	1.0
OpenDaylight [37]	SDN Controller Platform	Carbon SR1
Scapy Library [38]	Packet Manipulation Tool	2.2.0

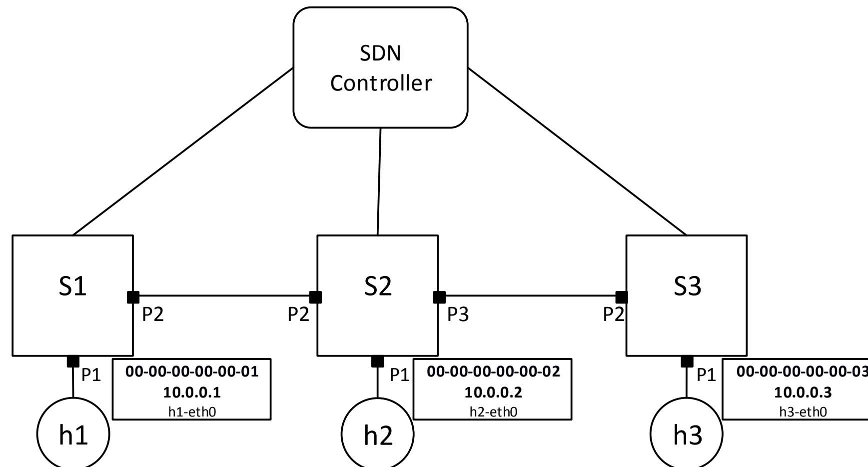


Figure 5.2: Basic Example Scenario

5.4.2 Spoofing with Regular ARP

For the regular ARP handling case, we initially used POX as the SDN controller platform and its *l2_learning* component to implement the packet forwarding functionality. With the *l2_learning* controller component, OpenFlow switches emulate the behaviour of traditional Ethernet learning switches. Consequently, ARP requests are broadcast to all hosts in the network, and ARP replies are unicast to the originator of the ARP request.

In our example, we assume that host *h1* has been compromised and wanted to poison host *h2*'s ARP cache by creating an entry that maps IP address *10.0.0.3* to MAC address

```
mininet> h2 arp -a
? (10.0.0.1) at 00:00:00:00:00:01 [ether] on h2-eth0
? (10.0.0.3) at 00:00:00:00:00:01 [ether] on h2-eth0
```

Figure 5.3: Poisoned ARP Cache (Mininet)

00:00:00:00:00:01. As a result, all packets from host *h2* addressed to host *h3* will be sent to the attacking host *h1* instead.

We implemented the attack by crafting an ARP request with TPA=*10.0.0.2*, SPA=*10.0.0.3*, and SHA=*00:00:00:00:00:01*, and injecting from host *h1* to switch *S1* via port *P1*.

Switch *S1* sends the packet as an OpenFlow *Packet-In* message to the controller, which then sends it back to switch *S1* as an OpenFlow *Packet-Out* message, with an instruction (action) to flood the packet. This is repeated on switch *S2*, where it is finally sent to host *h2* via port *P1*. Host *h2* updates its ARP cache and adds an entry mapping the IP address *10.0.0.3* to the MAC address *00:00:00:00:00:01*. We can verify the success of the attack by printing *h2*'s ARP cache, as shown in Figure 5.3.

As a result, all packets from host *h2* sent to host *h3* are redirected to host *h1*, which disrupts connectivity between host *h2* and host *h3*, representing a DoS attack. Using the ARP spoofing attack, an attacker can also launch a MITM attack, by additionally poisoning *h3*'s ARP cache, and adding an entry which maps the IP address of host *h2* to the MAC address of the attacker *h1*. As a result, all traffic between hosts *h2* and *h3* is sent via host *h1*, which can read and alter the traffic before relaying it to the intended destination node. We also performed this attack and confirmed its feasibility in our SDN context. We further replicated these experiments for IPv6 and NDP (NS/NA), with identical results.

To validate the feasibility of the attack on real networks, we also conducted the basic attack on the OFELIA test-bed [32]. In our experiment, we replicated the ARP spoofing attack scenario discussed above and shown in Figure 5.2. The difference is only in regards to the respective host addresses, i.e. the attacking host *h1* has IP address *10.216.12.56* and MAC address *02:03:00:00:00:60*, while *h2* has IP address *10.216.12.58* and MAC address *02:03:00:00:00:63*, and host *h3* has addresses *10.216.12.57* and *02:03:00:00:00:5d*.

```
root@Host2:~# arp -a
?(10.216.12.56) at 02:03:00:00:00:60 [ether] on eth0
?(10.216.12.57) at 02:03:00:00:00:60 [ether] on eth0
? (10.216.12.1) at 00:25:90:33:b4:bc [ether] on eth0
```

Figure 5.4: Poisoned ARP Cache (OFELIA)

We were able to successfully replicate the attack on the OFELIA test-bed. Figure 5.4 shows the spoofed ARP cache of host *h2*.

Our discussion of spoofing with regular ARP handling so far has been based on POX. We also investigated regular ARP handling of other controllers, i.e. Ryu, ONOS, Floodlight, and OpenDaylight. While Ryu implements regular ARP handling essentially in the same way as POX, ONOS, OpenDaylight and Floodlight use a slightly different approach of forwarding ARP requests in the network. In the case of POX and Ryu, the controller simply returns any ARP request to the switch, where the packet was received from, with an action to flood it via all its ports, except the ingress port. At every other switch where the ARP request is received, this iterative process is repeated, until the entire network is covered.

Both Floodlight and OpenDaylight, operate similarly, with only slight differences in the way ARP request packets are broadcast. In contrast, ONOS uses an approach where the controller, after receiving an ARP request, will send it out to all the switches in parallel, with a set of actions that send the packet out on all host ports. This avoids the iterative approach of the other controllers and is hence more efficient.

Irrespective of these differences in the forwarding of ARP requests in the network, we were able to successfully replicate the POX-based ARP spoofing attack with all the other mentioned controllers, which we confirmed via the hosts' ARP caches.[1]

We illustrate the success of the attack via a screenshot of the Web GUI of ONOS and Floodlight, as shown in Figures 5.5 and 5.6 respectively. Before the attack, we see that the MAC address of host *h1* (the attacker) is associated only with its own IP address of *10.0.0.1*. After the attack, we see that it is also associated with the IP addresses of hosts *h2* and *h3*, i.e. *10.0.0.2* and *10.0.0.3*. As a result, traffic destined to those hosts will be redirected to

[1]We used the default forwarding components for all controllers, e.g. *fwd* for ONOS and *forwarding* for Floodlight.

Hosts (3 total)

	HOST ID	MAC ADDRESS	VLAN ID	IP ADDRESSES	LOCATION
	00:00:00:00:00:01/None	00:00:00:00:00:01	None	10.0.0.1	of:0000000000000001/1
	00:00:00:00:00:02/None	00:00:00:00:00:02	None	10.0.0.2	of:0000000000000002/1
	00:00:00:00:00:03/None	00:00:00:00:00:03	None	10.0.0.3	of:0000000000000003/1

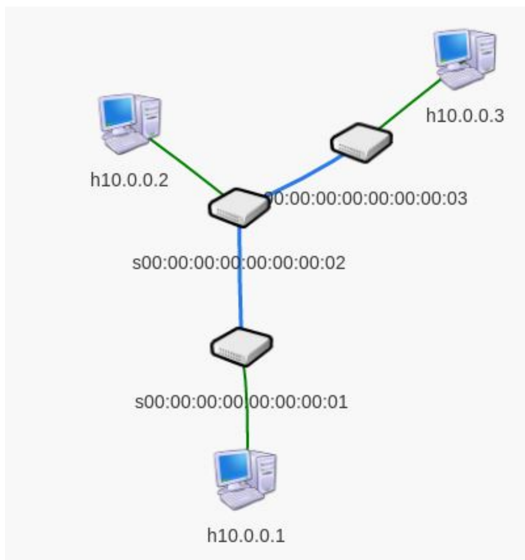
(a) Before the Attack

Hosts (3 total)

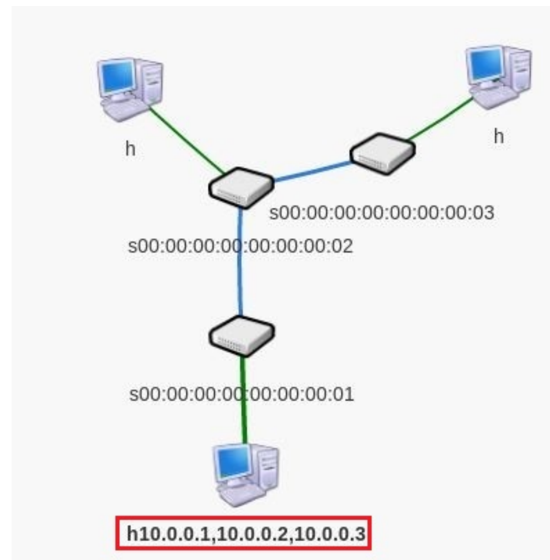
	HOST ID	MAC ADDRESS	VLAN ID	IP ADDRESSES	LOCATION
	00:00:00:00:00:01/None	00:00:00:00:00:01	None	10.0.0.3, 10.0.0.1, 10.0.0.2	of:0000000000000001/1
	00:00:00:00:00:02/None	00:00:00:00:00:02	None	10.0.0.2	of:0000000000000002/1
	00:00:00:00:00:03/None	00:00:00:00:00:03	None	10.0.0.3	of:0000000000000003/1

(b) After the Attack

Figure 5.5: ARP Spoofing Attack on ONOS



(a) Before the Attack



(b) After the Attack

Figure 5.6: ARP Spoofing Attack on Floodlight

host *h1*, and hence the attack was successful.

```
mininet@mininet-vm:~/pox$ ./pox.py proto.arp_responder
forwarding.l2_learning
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et
al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-03 1] connected
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
INFO:proto.arp_responder:00-00-00-00-00-01 learned 10.0.0.3
```

Figure 5.7: Poisoned ARP Responder Table

5.4.3 Spoofing with Proxy ARP

Proxy ARP is a commonly used approach to handling ARP in SDN since the centralised controller is ideally placed to handle ARP requests on behalf of hosts. That has the benefit of significantly reducing ARP traffic in the network. POX implements Proxy ARP functionality via the *arp_responder* component, which we used in this experiment.

The *arp_responder* component works as follows. All ARP requests are sent to the controller via an OpenFlow *Packet-In* message. The component uses its local ARP table to look up the matching MAC address for the address specified in the TPA field of the request and creates an ARP reply message with this information in the SHA field. The ARP reply message is then sent back to the switch, together with the instruction of sending it to the requesting host.

arp_responder builds its local ARP table by extracting IP-to-MAC address mappings from observed ARP requests, leaving it vulnerable to ARP spoofing attacks. We launched the same ARP poisoning attack as described in the previous section, but in this case, with the POX *arp_responder* component running. Figure 5.7 shows the log file of the POX *arp_responder* after the attack. The last line (in bold) shows the poisoned entry, mapping the IP address *10.0.0.3* of host *h3* to the MAC address *00:00:00:00:00:01* of the attacker *h1*. The controller will henceforth reply to any ARP request asking for the MAC address of host *h3* with the MAC address of host *h1*.

We further replicated the same ARP spoofing attack scenario on the Proxy ARP components of ONOS and Floodlight, which work in the same way as POX's *arp_responder* component. The results were identical, and we were able to spoof the controller's IP-to-MAC address

mapping database.

In the following sections, we discuss two countermeasures which aim to mitigate against the above shown ARP poisoning attacks in the specific context of SDN.

5.5 Countermeasure 1: SARP_DAI

We present SARP_DAI, our adoption of Dynamic ARP Inspection (DAI), to the context of SDN. The goal of SARP_DAI is not to provide ARP handling itself, but to transparently provide security for existing ARP handling mechanisms. DAI relies on a trusted database of IP-MAC address mappings. ARP packets with invalid information, i.e. SPA and SHA pairs that are inconsistent with the trusted database, are discarded. This checking of ARP packets in DAI is implemented in Ethernet switches. This is not practical in SDN, due to the limited 'intelligence' and feature set of OpenFlow switches. We, therefore, need to push this functionality to the SDN controller. In order to be able to check every single ARP packet, we install a high priority OpenFlow rule on each switch, which sends ARP packets to the controller.

In traditional networks, DAI maintains its trusted database of IP-MAC address mappings with methods such as DHCP-Snooping [144]. While DHCP-Snooping can be implemented in SDN, this is beyond the scope of this thesis. For our implementation, we simply assume that we have such a trusted database (arp_db) of IP and MAC address pairs.

We implemented SARP_DAI as a separate component in the POX controller platform. We configured the SARP_DAI POX component with the highest priority, which ensures that its Packet-In event handler is called before any other components. Algorithm 1 shows the basic processing of ARP packets by the SARP_DAI component.

If the received packet is an ARP packet, and the address specified in the SPA field is in our trusted database arp_db (line 2), we look up the corresponding value for SHA (line 3). If the SHA value in the packet does not match the corresponding value in the database, we drop the ARP packet (lines 4 and 5). If, however, the information in the ARP packet is consistent

Algorithm 1 SARP_DAI

```
1: for all received pkt do
2:   if pkt.type = ARP ∧ pkt.arp.spa ∈ arp_db then
3:     sha_trusted ← lookupSHA(arp_db, pkt.arp.spa)
4:     if pkt.arp.sha ≠ sha_trusted then
5:       drop ARP packet
6:     else
7:       continue
8:     end if
9:   end if
10: end for
```

with the value in arp_db, the packet handling routine returns, and the packet is passed to the next POX Packet-In event handler in the chain.

As mentioned, SARP_DAI relies on the availability of a trusted list of IP-to-MAC address mappings. If this is available, it is clear that SARP_DAI can successfully prevent any ARP spoofing attack, since it can easily identify and drop any ARP packet with spoofed and invalid information, i.e. the SPA and SHA fields. This requires that every single ARP packet is sent to the SDN controller, which incurs overhead and cost. In the following, we discuss and experimentally quantify this cost of SARP_DAI.

5.5.1 SARP_DAI Overhead

To evaluate the overhead and cost of SARP_DAI, we have performed a number of experiments using the experimental environment described in Section 5.4.1. We considered two overhead metrics in our evaluation, the additional CPU load generated at the controller by our SARP_DAI component, as well as the additional end-to-end Round Trip Time (RTT) measured via *ping*. Since we delete the ARP cache prior to each *ping*, the RTT gives us an indication of the increased delay caused by SARP_DAI. For our experiments, we defined two network topologies in Mininet, a linear topology of 64 switches and hosts, as shown in Figure 5.8, and a tree topology with nine switches and 64 hosts, with eight hosts connected to each of the eight access switches, as shown in Figure 5.9.

We evaluated the overhead for both Regular ARP handling as well as Proxy ARP. For the Regular ARP case, we used POX's *l2_learning* component to provide packet forwarding

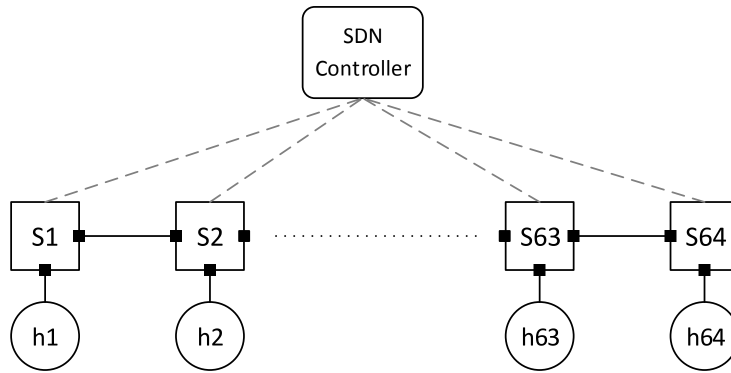


Figure 5.8: Linear Topology

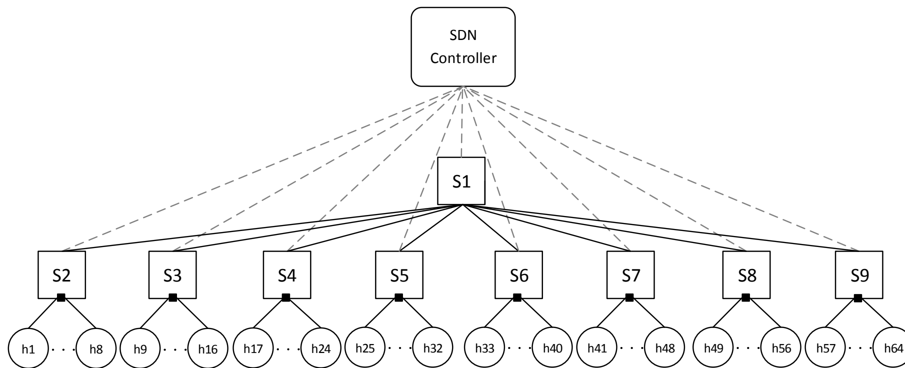


Figure 5.9: Tree Topology

functionality. For the Proxy ARP case, we used the *arp_responder* POX component. In our experiments, we generated 500 ARP requests per second at a constant rate. This corresponds to an ARP load of a reasonably large network. All ARP requests were initiated by host *h1* and sent to hosts *h2, h3, ..., h64* consecutively and continuously.

Figure 5.10 shows the average controller CPU load during our experiment. It is important to note that the absolute values are not that relevant here since they are determined largely by the level of the ARP load as well as the hardware. We are more interested in the relative increase in load due to our proposed security mechanism.

For the linear topology and Regular ARP handling, we see that the CPU load increases from around 22% to just over 41%. In the Proxy ARP case, the increase is from 4% to just under 10%. Not surprisingly, Proxy ARP creates a much smaller controller load compared to Regular ARP, since each ARP request is only handled once by the controller. In contrast, in Regular ARP along with the *l2_learning* component, each ARP packet is sent to the controller for checking at each switch along the path. The linear topology represents the worst case scenario for this since it has maximum length paths.

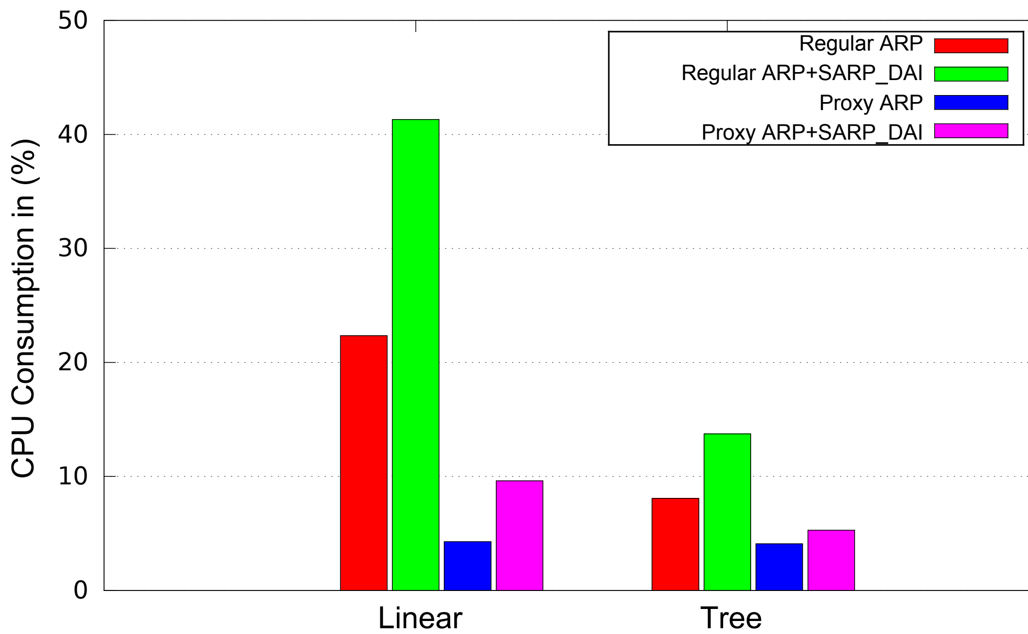


Figure 5.10: Controller CPU Load (SARP_DAI)

For the tree topology, we see a lower absolute CPU load, due to a shorter average path length. The increase in CPU load due to SARP_DAI for Regular ARP is from 8% to 14%. The corresponding increase for Proxy ARP is from 4% to 5%. In summary, we can say SARP_DAI imposes a significant computational load increase on the controller compared to traditional ARP handling. This is unavoidable, due to the centralised nature of SDN, where ARP packet checking can only be done at the controller.

In addition to the impact on controller CPU load, we also measured the additional end-to-end delay caused by SARP_DAI. We run *ping* from host *h1*, sending ICMP echo requests consecutively to all other hosts *h2, h3, ..., h64*. Since we delete the ARP cache every time before a new ICMP echo request is sent, we measure the time of the ICMP echo reply and request, as well as the time of the ARP exchange. Assuming that the transmission and processing time for the ICMP packets are the same no matter if SARP_DAI is enabled or not, we can use the difference in the RTT measurements as an indication of the additional delay incurred by SARP_DAI.

Figure 5.11 shows the RTT results for the linear topology for our four scenarios: Regular ARP, Regular ARP + SARP_DAI, Proxy ARP and Proxy ARP + SARP_DAI. As mentioned above, host *h1* is the originator of all ICMP echo requests and ARP requests. The x-axis in the figure represents the host ID of the target host, ranging from hosts *h2* to *h64*. As expected, we see a linear increase of the RTT with the index of the destination node. For

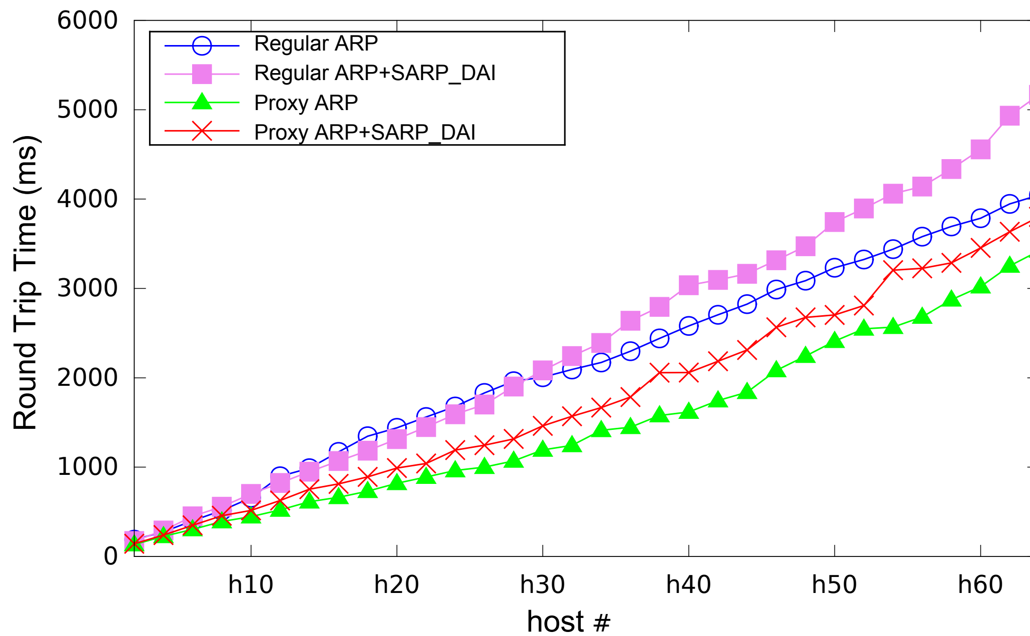


Figure 5.11: RTT in Linear Topology (SARP_DAI)

our linear topology, this directly corresponds to the path length. The absolute values of the RTT are very high, for all the four variants considered. This is largely due to the somewhat inefficient implementation of the *l2_learning* component, which forwards each broadcast packet to the controller, at every switch along the path. Consequently, the RTT for Proxy ARP is smaller since only the ICMP packet needs to be forwarded across the multiple hops, whereas the ARP request is directly answered by the controller via the first switch. As above, our focus is not on the absolute values, but rather on the relative increase due to SARP_DAI compared to the corresponding scenario without it. For Regular ARP, the increase is on average 12%, and for Proxy ARP, it is 16%.

Figure 5.12 shows the corresponding results for the tree topology. The key difference here is that we do not have a linear increase with the path length, but rather a bimodal behaviour, with a low delay for destination hosts *h2* to *h8*, and a higher, roughly constant delay for destination hosts *h9* to *h64*. This is due to the fact that hosts *h2* to *h8* are attached to the same access switch as host *h1*, the sender of all ARP requests, and hence only one switch needs to be traversed by all packets.

As expected in this case, Regular ARP has higher RTTs, since ARP packets are exchanged end-to-end between hosts, instead of being answered by the controller via the first switch in the path. Again, we focus on the relative increase in the RTT due to SARP_DAI instead of the absolute values. Average increase of the RTT caused by SARP_DAI for Regular ARP is

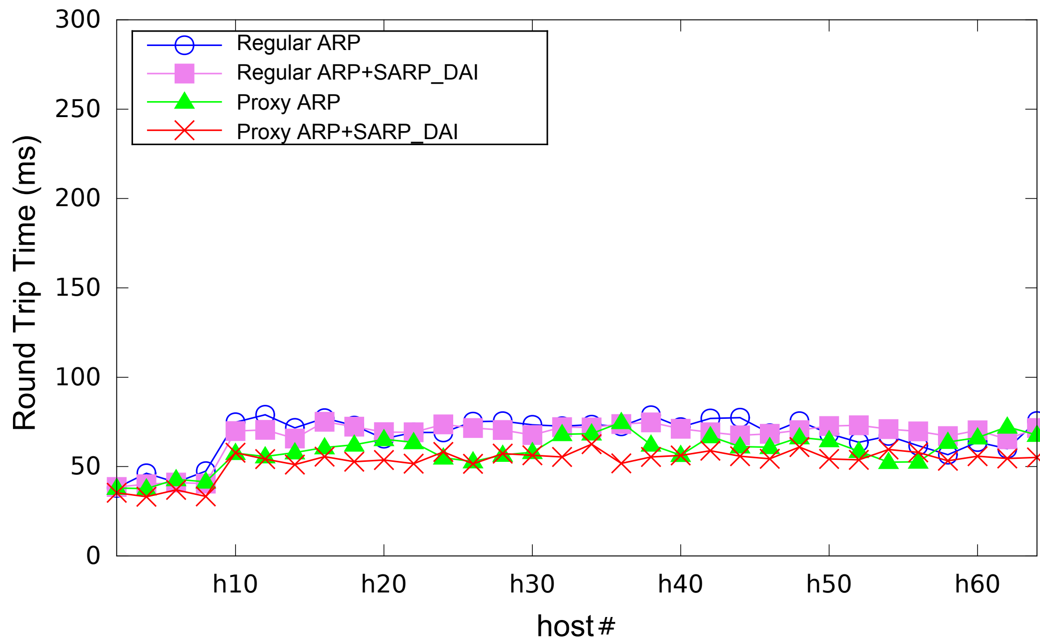


Figure 5.12: RTT in Tree Topology (SARP_DAI)

1%. For Proxy ARP it is 9%.

Similar to the CPU overhead, SARP_DAI also causes a significant increase in the delay of the ARP request/reply exchange, as measured via *ping* in our experiment. As mentioned above, this is due to the architecture of SDN and its centralised control plane.

However, it is easy to see that SARP_DAI is secure and can prevent the ARP spoofing attacks, as long as it has a trusted mapping of all IP and MAC address pairs, against which all ARP packets can be checked. Such as trusted database might not easily be available in all cases. In the following section, we explore a mitigation strategy that does not assume or rely on such a trusted database.

5.6 Countermeasure 2: SARP_NAT

In SARP_DAI, we assumed that the trusted mapping of IP-to-MAC addresses, which can be used to check the validity of every ARP packet is provided. However, such a trusted mapping might not be available in all scenarios. In this section, we propose a different method to mitigate against ARP spoofing attacks in SDN, which does not require any externally acquired

Algorithm 2 SARP_NAT

```
1: for all received pkt do
2:     if pkt.type = ARP.REQUEST then
3:         add entry in pend_req list
4:         pkt.arp.spa = spa_safe
5:         pkt.arp.sha = sha_safe
6:     end if
7:     if pkt.Type = ARP.REPLY then
8:         if pkt ∉ pend_req then
9:             Gratuitous ARP reply, drop
10:        else
11:            pkt.arp.tpa = lookupSPA(pend_req, pkt.arp.spa)
12:            pkt.arp.tha = lookupSHA(pend_req, pkt.arp.spa)
13:            deleteEntry(pend_req, TPA)
14:            addEntry(handled_req, pkt, t_handled)
15:            send ARP reply to pkt.arp.tha via Packet-Out
16:            if pkt ∈ handled_req then
17:                expSHA = lookupSHA(handled_req, pkt.arp.spa)
18:                if pkt.arp.sha ≠ expSHA then
19:                    Duplicate ARP reply attack detected, drop
20:                    Delete potentially poisoned host ARP cache
21:                else
22:                    Genuine duplicate ARP reply, drop
23:                end if
24:            else
25:                Gratuitous ARP reply, Drop Packet
26:            end if
27:        end if
28:    end if
29: end for
```

trusted ARP database. As before, the goal is to implement a controller component, which does not handle ARP itself, but provides security for ARP handling in SDN, in particular Regular ARP and Proxy ARP.

As mentioned, ARP spoofing attacks occur either via ARP request or reply messages. In both cases, the attacker spoofs the SPA and SHA fields in the ARP message to create an invalid address mapping. The key idea of our proposed mechanism is to prevent the potentially spoofed information in the SHA and SPA fields to come into contact with any hosts and thereby poisoning of their ARP cache. Similarly, in the Proxy ARP case, we want to avoid potentially poisoned information in the SPA and SHA fields to come into contact with an ARP handling controller component, i.e. the POX's *arp-responder* component that will implicitly trust this information. Our proposed solution is loosely inspired by Network Address Translation (NAT), and we therefore call it SARP_NAT. As in the case of SARP_DAI, in SARP_NAT ARP packets received by an SDN switch are sent to the controller. Algorithm 2 shows the packet processing of SARP_NAT at the controller and is explained below.

5.6.1 ARP Request based Attack

To explain the basic operation of SARP_NAT, we consider the scenario shown in Figure 5.2 and refer to the relevant lines in Algorithm 2. We assume the Regular ARP handling is used, and host *h1* attempts to launch the same ARP request based spoofing attack, described in Section 5.4.2, by injecting a poisoned ARP request with an invalid SPA field to switch *S1*. Due to a pre-installed rule at the switch, the ARP request is sent to the controller, where it is first handled by the SARP_NAT component. The SARP_NAT component stores each ARP request in a list of pending ARP requests ($pend_req$), consisting of the following 6-tuples for each entry: $(tpa, spa, sha, s, in_port, t_{rec})$ (line 3 in Algorithm 2). The elements tpa, spa and sha represent the corresponding fields in the ARP request, s and in_port represent the switch ID and ingress port via which the request was received, and t_{rec} is the time at which the request was received.

The SARP_NAT component then 'sanitises' the ARP request message by overwriting the potentially poisoned fields SPA and SHA, with safe dummy values, spa_{safe} and sha_{safe} (lines 4,5). These safe values are constant and can be any IP and MAC address pair that is not used on the local network. In our implementation, we used spa_{safe} = *11.11.11.11* and sha_{safe} = *00:11:22:33:44:55*. The Target Protocol Address (TPA) remains unchanged. The SARP_NAT component then passes the sanitised ARP packet to the next controller component or the event handler for processing. If we are using Regular ARP handling and a corresponding forwarding component, this will result in the ARP request packet being broadcast in the network, where it will eventually be received by host *h2*, the target node (TPA).

According to the pre-installed ARP rule, each ARP packet is sent to the controller for checking. However, this is not necessary for packets that have been sanitised by the SARP_NAT component and contain safe SPA and SHA values of spa_{safe} and sha_{safe}. We, therefore, modify the default ARP forwarding rule to specify that ARP packets with safe values are exempted, and are being forwarded as the normal forwarding rules.

If we assume Regular ARP handling, the ARP request is forwarded to the target host with IP address TPA, host *h2* in our example. Host *h2* then parses the ARP request packet and adds the specified SPA-SHA mapping in its ARP cache. This is the point where normally the ARP spoofing attack would have succeeded. However, in this case, host *h2* simply adds the

safe dummy values to its ARP cache, which has no impact.

When host *h2* creates an ARP reply message, it assumes that it is responding to the (non-existent) host with IP address spa_{safe} and MAC address sha_{safe}, and initialises the ARP reply accordingly, i.e. with TPA=spa_{safe} and THA= sha_{safe}. According to the standard ARP behaviour, the sender of the ARP reply (*h2*) initialises the fields SPA and SHA with its own IP and MAC address, in our example SPA=*10.0.0.2* and SHA=*00:00:00:00:00:02*. The ARP reply is then sent as a unicast frame via switch *S2*. Since the SPA and SHA fields are not the safe values spa_{safe} and sha_{safe}, the ARP reply is forwarded to the controller by switch *S2*, where it is processed by the SARP_NAT component. For an ARP reply, SARP_NAT first checks if it is a reply to a genuine request, i.e. if it is in $pend_req$ (line 8). If that is not the case, we have a gratuitous ARP reply, which we cannot trust and therefore drop. If, however, the ARR reply matches an entry in $pend_req$, the component performs the reverse address translation, by replacing the dummy values in the TPA and THA fields with the original values stored in $pend_req$, consisting of the set of $(tpa, spa, sha, s, in_port, t_{rec})$ 6-tuples (lines 11,12). In our example, the values are set as follows: TPA=*10.0.0.3* and THA=*00:00:00:00:00:01*. The corresponding entry is now deleted from $pend_req$ (line 13), and a new entry is created in a list of handled ARP requests $handled_req$ (line 14).[2] In addition to the information in $pend_req$, the list $handled_req$ also contains the resolved MAC address obtained from the ARP reply, as well as the time when the reply was received. We discuss how this is used below. Finally, the ARP reply is directly forwarded to host *h1*, via an OpenFlow *Packet-Out* message to $s = S1$, with instructions to send it out via port in_port, i.e. port *P1* in this case (line 15). The processing by SARP_NAT is completely transparent to hosts, and *h1* receives the same reply as it would have sent via the traditional ARP handling approach.

The above discussion considered the case of Regular ARP handling. The proposed SARP_NAT mechanism can also be used for Proxy ARP handling. This required minor modifications *arp_responder* component in our implementation. Normally, *arp_responder* implicitly trusts the information in the SPA and SHA fields in any ARP request it receives from switches, and the (potentially poisoned) information is used to update its local ARP table. We modify *arp_responder* to only trust and use IP-MAC address mappings in ARP reply messages which are sent in response to a genuine ARP request. This prevents spoofing attacks using ARP request messages against Proxy ARP handling. Attacks based on ARP

[2]The entry has a time out of t_{dup}, after which it is deleted from the list.

reply messages will be discussed below.

5.6.2 ARP Reply based Attack

In most ARP implementations, ARP spoofing can be done simply via gratuitous ARP replies, i.e. by sending an unsolicited reply to a host. To prevent such attacks, SARP_NAT will only accept ARP replies for which it has seen a corresponding request, i.e. for which there is an entry in the list of pending ARP requests $pend_req$. If this is not the case, it simply drops the ARP reply (line 9).

To avoid this check, an attacker could simply respond to a genuine ARP request with a spoofed reply. In this case, the SARP_NAT component receives two ARP replies with different values for the SHA field, one from the genuine target host, and one from the attacker. Unfortunately, it is impossible for the SARP_NAT component to determine which one is the genuine reply and which one is the spoofed one. SARP_NAT simply accepts the first ARP reply it sees for a pending request, processes and forwards it. In addition, it enters the information in the reply into its $handled_req$ list (line 14), in particular the corresponding SHA value and the time it was received.

If during a given time window t_{dup} another ARP reply for the same request is received, but with a different IP-to-MAC address mapping (lines 17-19), a *duplicate ARP reply* attack is detected. Since it is not possible to determine which of the ARP replies was spoofed, we make the conservative assumption that it was the first one forwarded to the host.

To mitigate the impact, the SARP_NAT component overwrites the potentially poisoned entry in the host's ARP cache by sending a new ARP reply with the same SPA but the safe value sha_{safe} (line 20). This stops any packets from being forwarded to the potentially wrong MAC address, thereby preventing a MITM attack. As a downside, it also prevents the host from communicating to the target host, until the next, valid ARP reply is received. The only way for this kind of attack to go undetected is if the attacker can prevent the genuine host from sending its valid ARP reply. This, however, is a significant challenge and greatly raises the bar for an ARP spoofing attack. SARP_NAT's protection against ARP reply based spoofing attacks works for both Regular ARP handling as well as Proxy ARP handling.

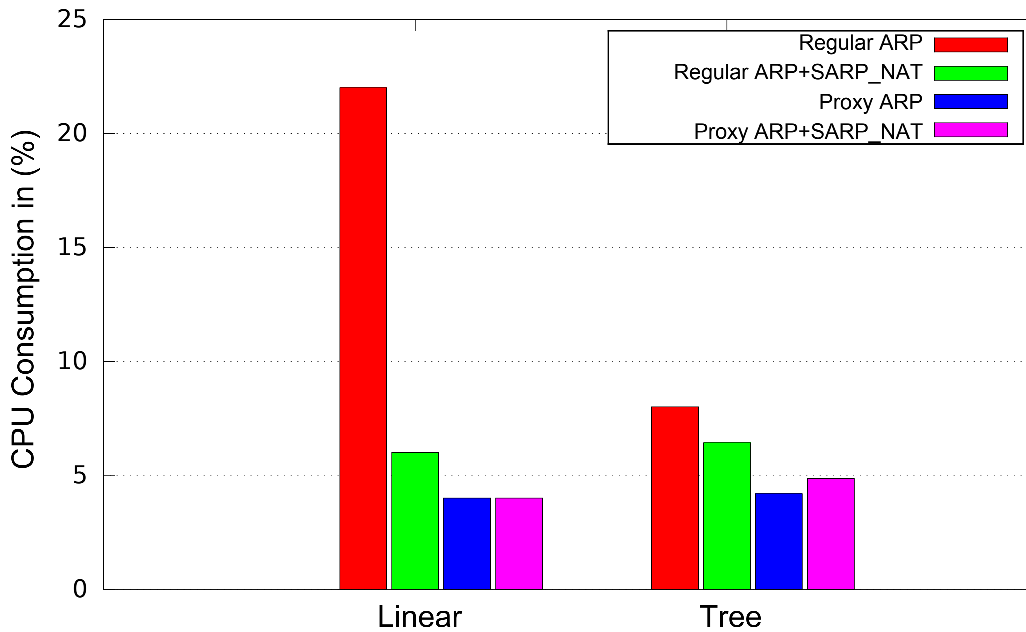


Figure 5.13: Controller CPU Load (SARP_NAT)

5.6.3 SARP_NAT Overhead

In order to evaluate the overhead imposed by SARP_NAT, we performed the same set of experiments we did for SARP_DAI. Figure 5.13 shows the average controller CPU load for the scenarios with and without SARP_NAT running. As in the case of SARP_DAI, we considered Regular ARP and Proxy ARP, for both our linear and tree topology.

We consider the results for the linear topology first. Surprisingly, we see that for Regular ARP, running SARP_NAT results in a significant reduction in CPU load. This is due to the fact that we install a rule which immediately forwards broadcast ARP request packets that have been 'sanitised' by the controller, instead of sending them to the controller at every switch, as is the behaviour POX's *l2_learning* component. In this case, adding security has the benefit of a significant controller overhead reduction, without sacrificing any generality of the solution. For Proxy ARP, SARP_NAT does not provide any reduction in CPU load, since the packet forwarding is the same for both cases. Here, we see that SARP_NAT minimally increases the CPU load by well below 1%.

For the tree topology and Regular ARP, we also see a reduction in CPU load due to SARP_NAT. However, the reduction is smaller compared to the linear topology since the average path length is shorter, resulting in less number of OpenFlow *Packet-In* messages

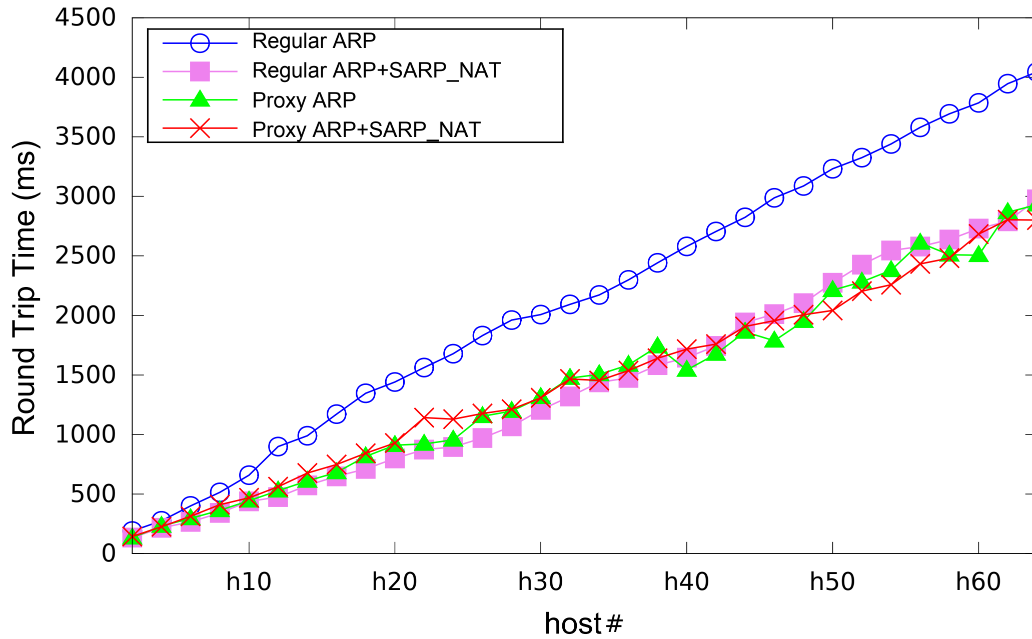


Figure 5.14: RTT in Linear Topology (SARP_NAT)

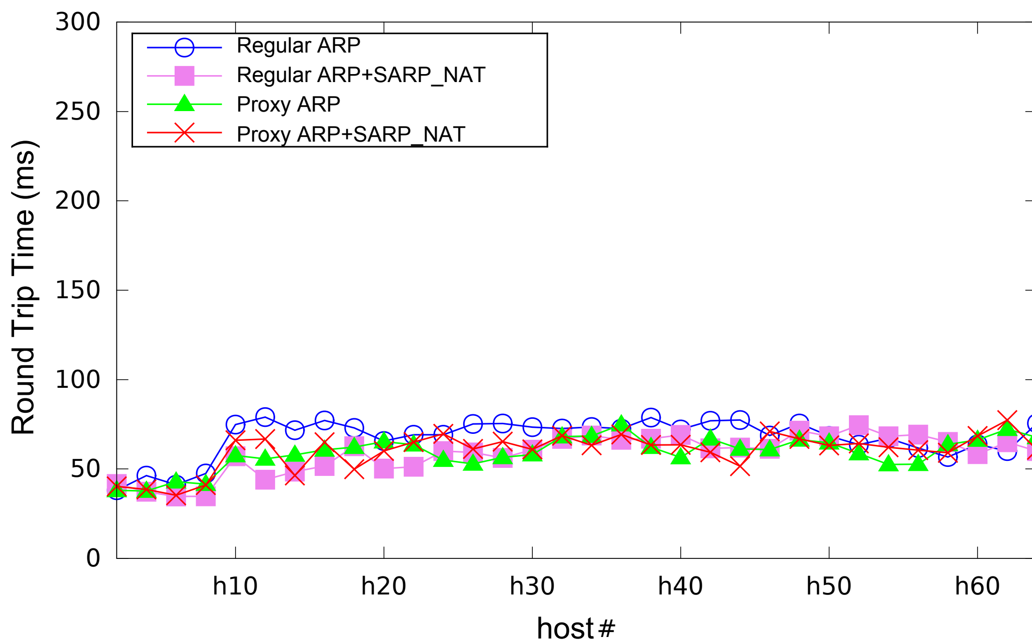


Figure 5.15: RTT in Tree Topology (SARP_NAT)

processed by the controller. For Proxy ARP, we see a small load increase of less than 1%.

We also measured the impact of SARP_NAT on the RTT, using the same scenario as for SARP_DAI. Figure 5.14 shows the results for the linear topology. We see that Regular ARP without SARP_NAT has the highest overall RTT values, due to the forwarding behaviour of the *l2_learning* component. The advantage of SARP_NAT is reflected here, with a reduction of 32% on average compared to the basic Regular ARP case. For Proxy ARP,

SARP_NAT imposes a minimal overhead of 2% on average. For the tree topology scenario in Figure 5.15, we see a smaller efficiency gain of SARP_NAT over basic Regular ARP of an average of 13%, as expected. For Proxy ARP, SARP_NAT processing at the controller results in a small increase in the RTT by 10% on average compared to the basic Proxy ARP case.

Overall, we can say that SARP_NAT achieves a significant performance gain over the basic Regular ARP case, and imposes a minimal overhead for Proxy ARP.

5.6.4 Comparison with SARP_DAI

In this subsection, we provide a brief qualitative and quantitative comparison of SARP_NAT with SARP_DAI.

The important point to stress here is that neither SARP_NAT nor SARP_DAI aim to provide ARP handling functionality. They are orthogonal to the provisioning of ARP handling, and simply provide a layer of security for whatever ARP handling approach is chosen.

As mentioned above, the key advantage of SARP_NAT compared to SARP_DAI is that it does not rely on a trusted database of IP-to-MAC address mappings to distinguish spoofed ARP packets from legitimate ones. This makes it a lot easier and practical to deploy in a production network.

Tables 5.2 and 5.3 provide a direct quantitative comparison of SARP_NAT with SARP_DAI in terms of controller CPU overhead as well as the RTT. The tables provide a different view of the experimental results shown and discussed in this and the previous section, with the aim of providing a direct comparison of SARP_NAT and SARP_DAI.

Table 5.2 shows that controller CPU overhead (in %) of SARP_NAT is consistently lower than for SARP_DAI, and the difference is particularly significant for Regular ARP and the linear topology scenario. This can be attributed to the reduction in the number of OpenFlow *Packet-In* messages processed by the controller in the case of SARP_NAT.

Table 5.2: Controller CPU Load Comparison

	SARP_DAI		SARP_NAT	
Topology	**Regular ARP**	**Proxy ARP**	**Regular ARP**	**Proxy ARP**
Linear	41.3%	9.6%	6.5%	4.2%
Tree	13.7%	5.2%	6.4%	4.8%

Table 5.3: Round Trip Time Comparison

	SARP_DAI		SARP_NAT	
Topology	**Regular ARP**	**Proxy ARP**	**Regular ARP**	**Proxy ARP**
Linear	+12%	+16%	-32%	+2%
Tree	+1%	+9%	-13%	+10%

Table 5.3 shows the corresponding table for the RTT measurements. The numbers (in %) show the relative increase or decrease in the RTT due to SARP_NAT or SARP_DAI, compared to the reference case without any ARP security mechanism in place.

With the exception of the Proxy ARP and the tree topology case, where SARP_DAI has a slightly lower RTT overhead (9% vs 10%), SARP_NAT outperforms SARP_DAI in all other scenarios. This difference is particularly significant, for Regular ARP, where SARP_NAT even manages to lower the RTT value, due to the more efficient forwarding of ARP messages, as discussed earlier in this chapter.

Overall, we can say that SARP_NAT provides better performance and a lower overhead compared to SARP_DAI. This is in addition to its benefit of not relying on a trusted IP-to-MAC database.

5.7 Related Works

The problem of ARP security has been well studied and discussed in the context of traditional networks, and we discussed the corresponding key works in Section 5.2.3. Here, we discuss works which consider the problem of ARP spoofing in the specific context of SDN, and are hence more closely related to our work. For our discussion, we classify the proposed

approaches into two separate categories. The first category, which we refer to as *DAI-like Approaches*, relies on a trusted IP-MAC address database to check the validity and integrity of ARP packets in the network. This is the same basic idea used in our proposed SARP-DAI module. The second category, *Proxy ARP-based Approaches*, proposes a central Proxy ARP handler at the controller. The assumption here is, as in the *DAI-like Approaches*, that a trusted IP-MAC address mapping is available, based on which ARP requests can be answered. This is in contrast to our proposed SARP_NAT, which is separate from the ARP handling mechanism, and can protect ARP packets without relying on a trusted IP-MAC database. In the following, we briefly discuss key recent papers for these two categories.

5.7.1 DAI-like Approaches

The authors of [146] propose a method for mitigating ARP spoofing attacks by validating ARP request and reply packets based on a number of consistency checks, e.g. the mechanism verifies if the MAC source and destination addresses in the Ethernet header are consistent with the corresponding addresses in the ARP payload. The most important rule is the one that checks if the IP-MAC address mapping in the ARP packet is consistent with the one in the trusted database.

SPHINX [128], as discussed in Chapter 3, is a proposal to detect a range of attacks against SDN, including ARP poisoning. As the previous approach, SPHINX relies on a trusted IP-MAC address mapping database, against which each ARP packet in the network is validated by the controller. The paper has a broad scope, and ARP spoofing represents only a small subsection. As a result, the approach is discussed in only limited technical detail, and no quantitative evaluation is provided.

In [147], a similar ARP poisoning mitigation method is proposed. While the authors provide some basic functional evaluation via experiments, they do not consider the overhead and impact on network performance of the proposed approach.

The same concept is presented in *FICUR* [148], an approach to observe ARP traffic and seek to protect the SDN controller against various ARP attacks. The main weakness of those studies is the failure to prevent the ARP spoofing attack deployed via an ARP reply

message, which the attacker could easily spoof and accomplish the ARP spoofing attack. It has taken no account of that the first packets received by the controller could be already spoofed, which in this case the database would eventually include wrong information.

All the above works propose the same basic idea to prevent ARP poisoning or spoofing, by adopting the concept of DAI [25] to the SDN context, which is essentially the same as our SARP_DAI approach.

In contrast to our work, none of the papers mentioned above provide an experimental and quantitative evaluation of the computational overhead on the controller of this approach (DAI), as well as its impact on network performance, i.e. end-to-end Round Trip Time (RTT).

5.7.2 Proxy ARP-based Approaches

In [149], the authors propose a centralised approach to ARP handling in SDN, with the main aim of improving efficiency and reducing overhead, which it successfully achieves. All ARP requests are sent to the SDN controller, which then creates ARP replies based on its IP-MAC address database. The paper specifically considers a data centre scenario, and assumes that the database is available from "the data centre management framework". While the paper does not specifically consider the security aspects of ARP handling, it could be extended to do so and form the basis of a secure Proxy ARP handler. Since all ARP requests are handled centrally and the trusted IP-MAC database is available, the controller is able to provide trusted and valid ARP replies. However, the approach would need to be extended in regards to the handling of gratuitous ARP replies, and a mechanism would need to be provided to guarantee that potentially spoofed ARP request messages are quarantined from other hosts. Furthermore, if the approach was to be generalised for other types of network scenarios, another way would need to be found to establish the trusted IP-MAC address mapping.

In [150], the author briefly investigates the security issues with the current implementation of ARP and attempts to reduce ARP broadcast traffic via centralised ARP handling. In contrast to [149], ARP requests are not handled by the controller, but by a dedicated server instead, which also handles DHCP requests. In this approach, OpenFlow rules are installed

on all switches dropping all ARP replies which are not originating from the ARP server. The proposed approach also relies on a trusted database of IP-MAC address mappings. The author suggests that this information can be learned by simply observing network traffic, and extract the IP-MAC address pairs from packets. The problem with this method is that the initially observed packets can already be spoofed, which then poisons the database. Another limitation of the proposal is its dependency on DHCP, and its reliance on an additional dedicated server, which is in addition to the standard SDN infrastructure.

More recently, the authors of [151] propose a centralised ARP handling approach via the SDN controller, with the aim of preventing ARP spoofing attacks. The approach in [151] forwards all ARP request to the controller. If the controller has the required IP-MAC mapping, it will create the corresponding ARP reply message. Otherwise, the ARP request is broadcast as in the regular ARP handling approach. The problem is that this approach still allows ARP poisoning via ARP request packets since only ARP reply messages are checked. Furthermore, the approach also relies on observing the ARP packets in the network to build up its IP-MAC database. The paper does not provide a solution to the problem of trust bootstrapping, since the initial ARP packets could already be spoofed.

In [152], a distributed ARP handling approach for SDN is proposed, where multiple *DR-ARP* responder entities are in charge of handling ARP requests. The motivation for a distributed approach is not made very clear, especially since a centralised approach, e.g. via the controller, seems a natural fit for SDN. The proposed method requires ARP requests to be tagged with meta-data, e.g. the switch ingress port. Furthermore, the approach requires a subset of all network traffic to be mirrored to the DR-ARP entities, in order for them to learn the IP-MAC address mapping. This raises questions about the practicality and scalability of the approach, in addition to the problem of trust bootstrapping, which is not addressed.

In summary, the above approaches propose to handle ARP requests via Proxy ARP, where ARP replies are created based on a database of IP-MAC address mappings. The security of these approaches relies on the availability of a trusted source of information to populate the database. This is a very challenging problem, and none of the discussed papers provides a solution that is generic and provides absolute security guarantees.

This is in contrast to our SARP_NAT approach, which does not handle ARP request itself,

but provides a layer of security to the ARP handling mechanism. The key distinction and benefit of SARP_NAT compared to all these related approaches is that it does not rely on the availability of a trusted IP-MAC address database. Finally, none of the related works have provided an experimental evaluation of the computational cost of their proposed mechanism, as is provided in this chapter for both SARP_DAI and SARP_NAT.

5.8 Conclusions

While SARP_DAI, an adoption of an existing solution used in traditional networks, can successfully prevent all such attacks, it imposes a significant overhead on the controller, and it relies on a trusted database of IP-to-MAC address mappings. Since this cannot always be assumed, we presented SARP_NAT, an active, SDN-specific mitigation approach that does not rely on any trusted a-priori information. SARP_NAT can defend against ARP request based spoofing attacks, and against gratuitous ARP reply attacks. While it is impossible to prevent attacks based on ARP replies sent to genuine ARP requests, SARP_NAT can detect such duplicate ARP reply attacks and mitigate the impact by overwriting the affected host's ARP cache. SARP_NAT's active address translation approach provides a novel solution to the problem of ARP spoofing in SDN.

Chapter 6

Efficient Address Resolution Protocol Handling

6.1 Introduction

Layer 3 to Layer 2 address mapping is a critical functionality in packet switched networks. In IPv4, this service is provided via the Address Resolution Protocol (ARP), and in IPv6 via Neighbour Discovery Protocol (NDP). While our discussions in this chapter focus on ARP, our proposed method can equally be applied to NDP in IPv6, in particular Neighbour Solicitation (NS) and Neighbour Announcement (NA).

ARP imposes a significant overhead on Ethernet networks. For example, [153] reports that ARP traffic represents 88% of all broadcast traffic. Efficient handling of ARP is therefore critical for the scalability of networks. In the context of SDN, a widely used approach to handle ARP and address the problem of extensive broadcast traffic is by using Proxy ARP, where the SDN controller handles ARP requests on behalf of hosts. While Proxy ARP in SDN manages to reduce the ARP induced broadcast traffic, it places a significant load on the controller, limiting overall network performance and scalability and making the entire network vulnerable to DoS attacks [154].

To address this problem, we explore the idea of offloading Proxy ARP functionality from the control plane (SDN controller) to the data plane (switches). We refer to our proposed solution

as Switch-based Proxy ARP (SProxy ARP).

For our work, we assume an SDN with an OpenFlow-based southbound interface [2]. OpenFlow provides a very limited interface to program switch functionality via its basic match-action paradigm. This chapter demonstrates how the limited OpenFlow interface can be utilised to successfully implement SProxy ARP, without requiring any further, non-standard modifications to switches.

Our experimental evaluations show the potential performance benefits of SProxy ARP, with a reduction of ARP response time of more than an order of magnitude, while providing a significant reduction of controller load.

The rest of the chapter is organised as follows: Section 6.2 discusses the state-of-the-art in ARP handling in SDN, and Section 6.3 describes SProxy ARP and its implementation. Section 7.4 briefly describes our experimental platforms and Section 6.5 presents our evaluation results. Section 6.6 discusses the memory-performance trade off in SProxy ARP. Section 8.4 discusses related works, and Section 8.8 concludes the chapter.

6.2 ARP Handling in SDN

As discussed in Chapter 5, there are two basic methods for handling ARP in OpenFlow-based SDNs. The first method, which we call *Regular ARP*, in which a host broadcasts an ARP request on the local subnet, and the host with the IP address specified in the TPA field responds with a unicast ARP reply containing its own MAC address as the answer. There is nothing SDN-specific in this approach, and it works as in traditional networks. The role of the SDN controller here is just to provide the packet forwarding functionality, e.g. via a learning switch component, for ARP request and ARP reply messages between hosts. This approach suffers from the same problem of high broadcast traffic load as is the case in traditional networks.

The second approach to handle ARP in SDN is via *Proxy ARP*. In this case, ARP requests are sent to the controller when they are received at a switch from a host. The controller

maintains a database of IP-MAC address mappings and is then able to directly create the corresponding ARP reply message. This ARP reply is sent back to the switch, encapsulated in an OpenFlow *Packet-Out* message, together with the instruction of sending it out to the ingress port via which the ARP request was received. This approach is well suited to the centralised nature of SDN. The key benefit is that it avoids excessive broadcast traffic in the network and the associated overhead. However, a key drawback of this approach is the increased controller load, since the controller needs to handle every single ARP request, in addition to handling other load.

Proxy ARP is widely used in SDN and is implemented by most SDN controller platforms. Our investigations showed that the following controllers support Proxy ARP: POX [33], Ryu [34], ONOS [35], Floodlight [36], OpenDaylight [37], and Beacon [121].

In the following section, we discuss *SProxy ARP*, our simple and practical proposal to mitigate the key limitations of Proxy ARP in SDN.

6.3 Switch-based Proxy ARP (SProxy ARP)

The key idea in SProxy ARP is to offload the ARP handling functionality, at least partially, from the control plane (SDN controller) to the data plane (switches).

In SProxy ARP, the switch directly replies to the ARP request with the corresponding ARP reply message, without involving the controller. The problem is that OpenFlow provides a very narrow interface to program switch functionality, limited to basic primitives that can be implemented, via its *match-action* paradigm. For example, OpenFlow does not provide a mechanism to create a new message in response to another message. Furthermore, its packet matching functionality is generally limited to packet headers and does not allow access to payload information. However, since OpenFlow version 1.3, there is an important exception to this rule, i.e. ARP payload fields such as SPA, SHA, TPA, THA, etc. are added as match fields. Since OpenFlow supports the operation of rewriting of packet fields that are defined as match fields, this mechanism allows us to answer ARP requests at the switch. We are able to rewrite the fields of the ARP request to convert it into the corresponding ARP

Algorithm 3 SProxy ARP Request Processing

1: **for all** received pkt **do**
2: **if** pkt.type = ARP $\wedge$ pkt.arp.operation=1 **then**
3: $THA \leftarrow lookupMAC(pkt.arp.tpa)$
4: $TPA \leftarrow pkt.arp.tpa$
5: $SHA \leftarrow pkt.arp.sha$
6: $SPA \leftarrow pkt.arp.spa$
7: $pkt.op \leftarrow 2$
8: $pkt.eth_src \leftarrow THA$
9: $pkt.eth_dst \leftarrow SHA$
10: $pkt.arp.tpa \leftarrow SPA$
11: $pkt.arp.tha \leftarrow SHA$
12: $pkt.arp.spa \leftarrow TPA$
13: $pkt.arp.sha \leftarrow THA$
14: **end if**
15: **end for**

reply message and send it back to the host, via the ingress port on which the ARP request was received.

The process is shown in Algorithm 3. If the packet is an ARP request (line 2), we look up the value for THA in the local IP-MAC database and store it as the temporary variable THA (line 3). We also read the values of TPA, SHA and SPA from the ARP payload and store them in temporary variables (lines 4-6). Now we convert the packet from an ARP request into an ARP reply by rewriting the *operation* field (line 7). Next, we set the source and destination MAC address of the Ethernet frame which carries the ARP payload to the value of THA and SHA respectively (lines 8-9). Finally, the TPA, THA, SPA, and SHA fields of the converted ARP reply messages are set accordingly (lines 10-13), as discussed in Chapter 5.

Unfortunately, we cannot directly implement Algorithm 3 in OpenFlow, since it does not support a database and lookup mechanism (line 3), nor does it support variables (lines 3-6). Instead, we need to install a dedicated OpenFlow rule for each TPA we want to handle at the switch, with the corresponding values for THA, SHA and SPA 'hard-coded' in the rules as literals. To illustrate this, Figure 6.1 shows an example of such an OpenFlow rule, defined for a TPA of *10.0.0.2*, with its corresponding match and action components. Lines 1 and 2 represent the match rules, and lines 3-11 are the corresponding actions.[1]

Line 1 in the figure matches on ARP request packets, corresponding to line 1 in Algorithm 3,

[1] The syntax used in the figure is as provided by the *dpctl* tool, we only add the line breaks and line numbers.

```
1:  arp, arp_op=1
2:  arp_tpa=10.0.0.2
3:  actions=
4:  set_field:2->arp_op
5:  set_field:00:00:00:00:00:02->eth_src
6:  set_field:ff:ff:ff:ff:ff:ff->eth_dst
7:  set_field:10.0.0.2->arp_spa
8:  set_field:00:00:00:00:00:02->arp_sha
9:  set_field:10.255.255.255->arp_tpa
10: set_field:ff:ff:ff:ff:ff:ff->arp_tha
11: IN_PORT
```

Figure 6.1: Example OpenFlow Rule for SProxy ARP

and Line 2 matches on a particular value of TPA (*10.0.0.2*), in this example. The rest of the rule specifies the actions for initialising the ARP reply for this particular IP address.[2] From line 3 onwards, we see the OpenFlow actions for creating, initialising and forwarding the corresponding ARP reply message. Line 4 sets the ARP type to 'reply'. Line 5 in Figure 6.1 sets the Ethernet source MAC address as the hard-coded value corresponding to the TPA of *10.0.0.2*, which is *00:00:00:00:00:02* in this example. Line 6 sets the destination MAC address of the Ethernet frame as the broadcast address. This differs from the corresponding line 9 in Algorithm 3, where we use the corresponding unicast address. Due to the limitation of OpenFlow, we cannot copy that value from the ARP request packet, and we, therefore, need to use the broadcast address as a constant, hard-coded value. Similarly, we need to use the corresponding broadcast address for the values of TPA and THA. While this is a deviation from the regular approach to initiate ARP replies, it provides the identical functionality. Finally, line 11 in Figure 6.1 tells the switch to send the 'ARP request-turned-ARP reply' packet out to port *IN_PORT*, i.e. the ingress port, where the request was received on.

For SProxy ARP, we assume that the controller maintains a database of IP-MAC address mappings, as implemented in traditional Proxy ARP, i.e. via DHCP-Snooping [155] or via learning the information from passively observing data packets. The controller decides for

[2]As mentioned previously, this approach requires the installation of a separate rule for each TPA to be handled at the switch.

which IP addresses (TPAs), a corresponding rule for ARP handling is to be installed on the switches. For now, we assume a rule is installed for each TPA, which means that all ARP requests can be handled by the switch. In Section 6.6, we discuss the consequences of relaxing that assumption. We implement SProxy ARP as a separate component in the Ryu controller platform.

6.4 Experimental Platform

We evaluated SProxy ARP both in an emulated network using Mininet [30], as well as a dedicated hardware test-bed. For our experiments, we configured the simple scenario shown in Figure 6.2, with an SDN controller running Ryu, an OpenFlow switch *S1* and two hosts *h1* and *h2*.

Since we are evaluating the performance of Proxy ARP, there is no need to consider more complex topologies. In fact, the basic process of handling ARP requests from a host involves only the corresponding access switch and the controller and is therefore independent of the rest of the network. We further used the PackETH tool [40] to generate an ARP request from host *h1* to host *h2* at various rates.

For our emulation experiments, we used Mininet [30] and Open vSwitch (OVS) [31]. Table 8.2 summarises the relevant software tools we used in the experiments of this chapter. All experiments were run on a Dell server (PowerEdge R320 with a 12-core Xeon E5-2400 CPU and 32GB of RAM), running Ubuntu Linux 16.04 with kernel version 3.16.0.

For our hardware test-bed experiments, we used two Dell R320 servers, with specifications as mentioned above, one as host *h1* and the second one as the OVS switch *S1*. We further used 2 Dell laptops with an Intel 2.6GHz dual-core CPU and 4GB of RAM, one as the SDN controller running Ryu and the other one as host *h2*. With these nodes, we configured the scenario shown in Figure 6.2.

Table 6.1: Software Tools used for Implementation and Experiments in Chapter 6

Software	Function	Version
Mininet [30]	Network Emulator	2.2.2
Open vSwitch [31]	Software SDN Switch	2.5.2
Ryu [34]	SDN Controller Platform	3.19
PackETH [40]	Packet Generator	1.8.1
Stress-ng [42]	Control Traffic Tool	0.02.26

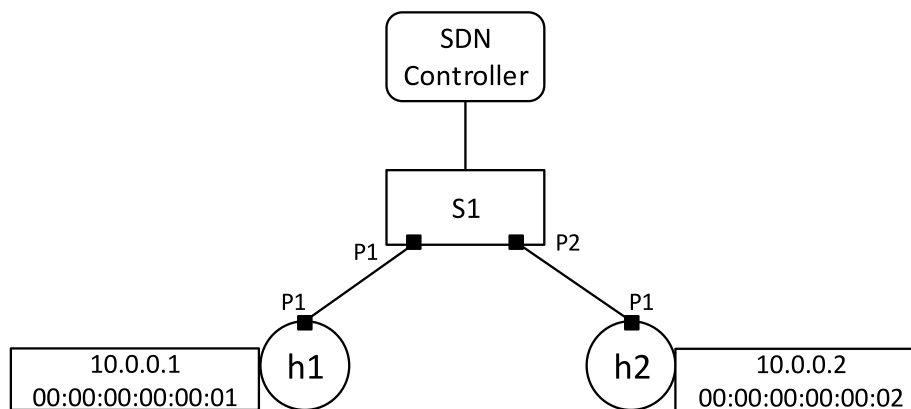


Figure 6.2: Basic Experiment Scenario

6.5 Evaluation

In this section, we first consider the improvement in the ARP response time that SProxy ARP achieves compared to the traditional Proxy ARP approach in SDN. We also look at the reduction in controller overhead of SProxy compared to Proxy ARP, with different ARP request sending rates.

To achieve this goal, we performed a number of experimental evaluations of SProxy ARP. For these experiments, we installed OpenFlow rules for all values of TPA that we used in our experiments, which made sure that all ARP requests can be answered by the switch. This represents a best-case scenario.

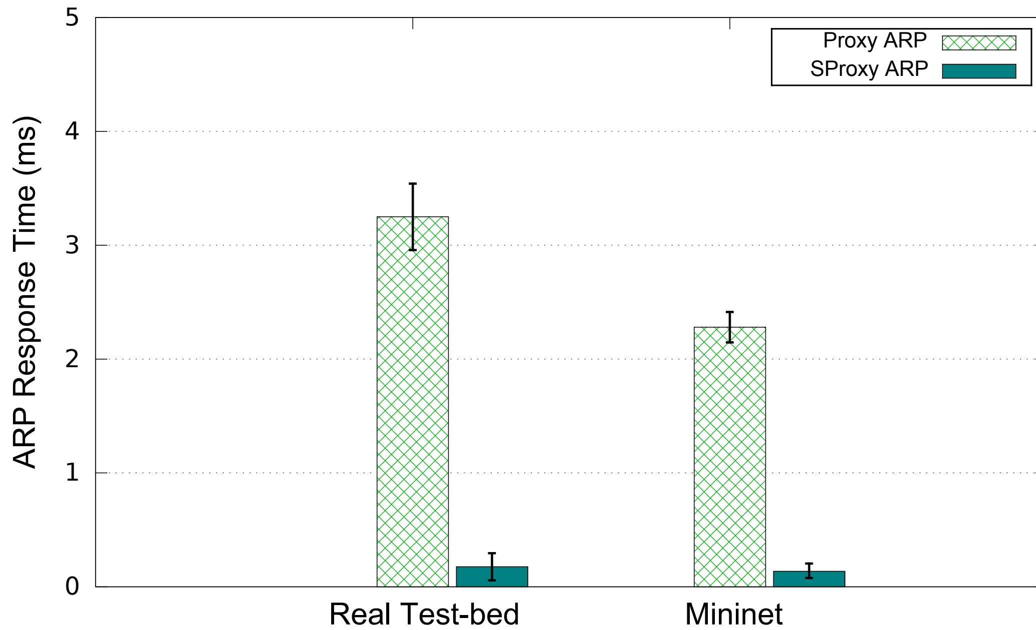


Figure 6.3: ARP Response Time

6.5.1 ARP Response Time

In this experiment, we measured the ARP response time δ at host *h1* by taking the difference $\delta = t_2 - t_1$ between the time t_1 when the ARP request is sent and the time t_2 when the corresponding ARP reply is received at the host.

Figure 6.3 shows the results for both Proxy ARP and SProxy ARP, from both our hardware test-bed as well as Mininet. The figure shows the average over ten individual measurements, with the corresponding 95% confidence interval. As expected, we see a significant reduction in the ARP response time achieved by SProxy ARP, from 3.25ms to 0.18ms on the real test-bed, and from 2.28ms to 0.14ms in Mininet. This is achieved by avoiding the extra round trip time to the controller, and the necessary parsing of OpenFlow *Packet-In* and *Packet-Out* messages. The lower value of ARP response time of Proxy ARP in Mininet compared to the hardware test-bed can be explained by the fact that the control channel over which OpenFlow *Packet-In* and *Packet-Out* messages are sent is faster since it is done via an emulated link rather than a physical link.

In the above experiment, the controller was idle and did not perform any other functionality. This is a somewhat unrealistic assumption, since in a real network, the controller would typically be busy, e.g. handling *table-miss* events and installing new forwarding rules on

switches. We therefore want to investigate how the ARP response time is affected by various levels of background load on the controller. We used the Stress-ng tool [42] to create various levels of control traffic and hence CPU load.

Figure 6.4 shows the ARP response time for Proxy ARP as well as SProxy ARP, measured in both Mininet and the hardware test-bed. The x-axis shows the increasing controller CPU background load generated by the Stress-ng tool. The figure shows the average over ten measurements, with the corresponding 95% confidence interval.

We see that an increased controller background load results in an increased ARP response time for Proxy ARP, as expected. For a controller load of 100%, the ARP response time is double the amount as for the idle controller case. The increase is approximately linear for both the Mininet and hardware test-bed case, reaching more than double the ARP response time for a maximum controller load compared to the case of an idle controller.

Also as expected, the ARP response time for SProxy ARP is constant and independent of the controller load since ARP requests are handled by the switch without any controller involvement. In fact, the SProxy ARP figures are almost identical for both Mininet and the hardware test-bed, and we, therefore, do not differentiate between the two.

We observe that in the case of 100% controller load, SProxy ARP reduces the ARP response time by a factor of more than 36 (hardware test-bed), compared to traditional Proxy ARP.

6.5.2 Controller Overhead

Here, we consider the computational overhead placed on the SDN controller by the traditional Proxy ARP handling approach. The controller CPU load is caused by the parsing of ARP request packets encapsulated in OpenFlow *Packet-In* messages, looking up the MAC address corresponding to the TPA, and creating and sending the ARP reply message encapsulated in an OpenFlow *Packet-Out* message to the switch. That gives us an indication of the potential controller load reduction of SProxy ARP, which in the ideal case places virtually no load on the controller.

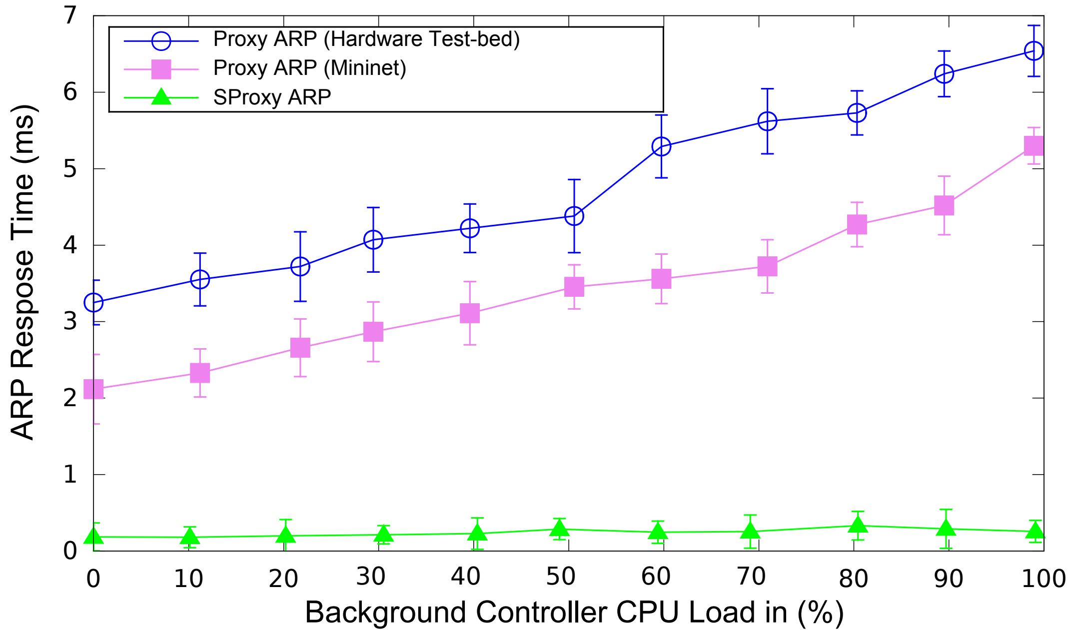


Figure 6.4: ARP Response Time vs. Background Controller CPU Load

Figure 6.5 shows the controller CPU load caused by ARP handling via Proxy ARP, for different ARP request sending rates, ranging from 200/s up to 1600/s. As before, the figure shows the average over 10 experiment runs, with 95% confidence intervals. As expected, we see a linear increase in the controller CPU load with an increased ARP request sending rate for Proxy ARP in both scenario, Mininet and the hardware test-bed. While the absolute CPU load values depend on the specific controller hardware, we see that Proxy ARP handling can create a significant CPU load. This has a negative impact on network scalability and the ability of the SDN controller to react timely to other important network events, e.g. *table-miss* events.[3] In contrast to Proxy ARP, we see that SProxy ARP creates no controller load in this scenario, as expected.

In addition to the CPU load imposed on the controller by Proxy ARP, we also considered its overhead in terms of ongoing control traffic between switches and the controller.[4] Figure 6.6 shows the amount of control traffic as a function of the ARP request sending rate, for the same scenario as used in Figure 6.5. As expected, we again see a linear increase in the control traffic overhead for Proxy ARP and a constant value of 0 for SProxy ARP.

[3]While an ARP request sending rate of 1600/s is arguably very high, our own measurements of networks at the University of Mjamaah and the University of Queensland have shown ARP rates of several hundred per second.

[4]We ignore traffic required for the initial installation of rules, since this is a one-off cost, compared to continuous traffic overhead of Proxy ARP.

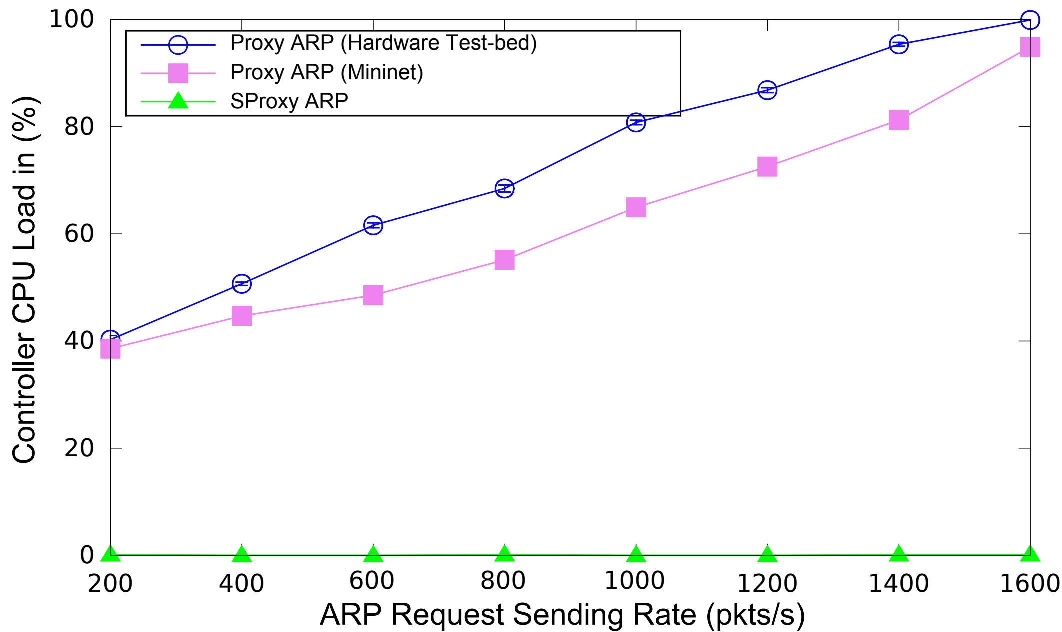


Figure 6.5: CPU Consumption with Sending Rate

In summary, we can conclude that SProxy ARP achieves a number of key improvements over traditional Proxy ARP handling in SDN. It achieves a better than the order of magnitude reduction in the ARP response time, while also significantly reducing the overhead on the controller, both in terms of CPU load as well as in regards to the control traffic. Reducing controller load, which makes the network roust against DoS attacks, is a critical issue for network scalability in SDN.

However, so far we have assumed that all ARP requests are handled by the switch, and consequently a rule for each requested IP-MAC address mapping is installed on the relevant switch. This is a somewhat idealistic assumption. While memory in software switches such as OVS might not be such a limiting factor, TCAM memory in hardware OpenFlow switches is certainly a limited and expensive resource. The following section explores the corresponding memory-performance trade-off in SProxy ARP.

6.6 Switch Memory-Performance Trade-off

In this section, we discuss the trade-off between the number of rules installed and required switch memory, versus the possible performance and efficiency gain. To investigate the

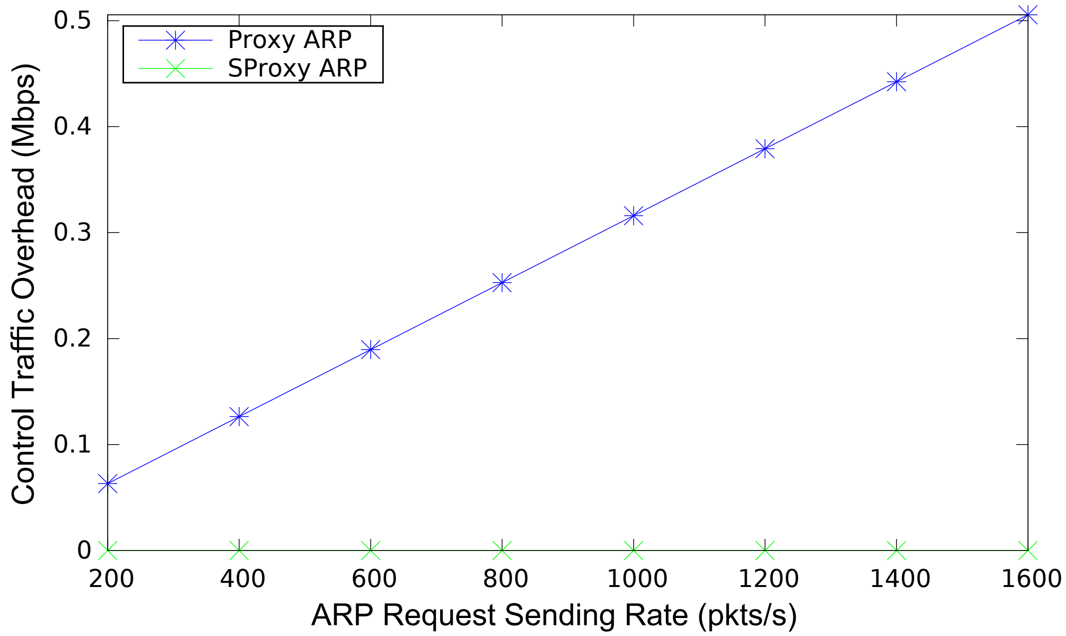


Figure 6.6: Total Packet size with Sending Rate

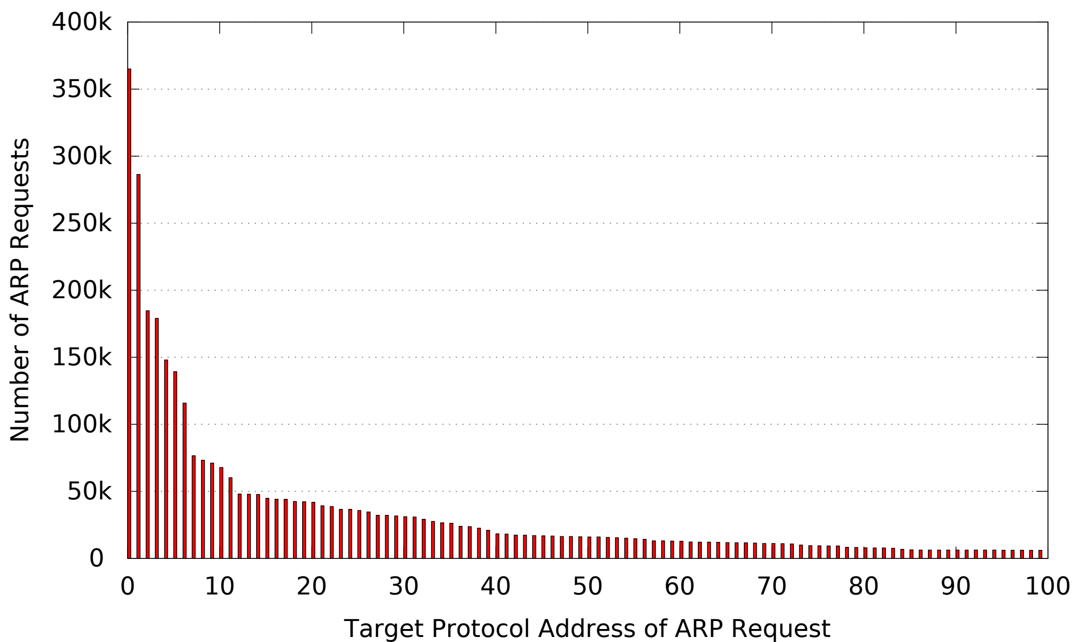


Figure 6.7: ARP Request Distribution based on TPA

memory-performance trade-off of SProxy ARP, we collected a 33-hour trace of ARP traffic from a core switch at the School of ITEE at the University of Queensland.

Figure 6.7 shows the distribution of ARP requests based on the TPA. We see a power-law-like distribution, a small number of IP addresses (TPAs) with a high number of ARP requests, and a long tail of addresses with a small number of requests each.

Table 6.2: Memory Performance Trade-off

Number of Rules(n)	Fraction of ARP Requests Covered
10	22.7%
20	31.4%
30	37.8%
40	42.5%
50	45.5%
60	48.1%
70	50.2%
80	51.9%
90	53.2%
100	54.2%
200	62.4%
300	68.2%
400	72.8%
500	76.5%
600	80.1%
700	83.2%
800	86.0%
900	88.8%
1000	91.4%

This seems to indicate that by installing a relatively small number of switch rules for SProxy ARP handling, a significant proportion of ARP traffic can be handled at the switch, and hence a significant performance gain can be achieved at a limited memory cost.

Table 6.2 shows the percentage of ARP requests that can be handled at the switch by installing rules for the n most requested IP addresses, for different values of n. We see that by installing rules for the $n = 50$ most common IP addresses, more than 46% of all ARP request can be offloaded to the switch, and therefore more than 46% of the associated performance gains and controller load reduction can be achieved. For $n = 100$, we can cover more than 50% of all ARP requests. As can be seen in Figure 6.7, the marginal gain of installing additional rules is very low for n approaching 100 and below. We have collected further ARP traces from other networks, and the qualitative property of the distribution with a small number of high demand IP addresses and a long tail of rarely requested addresses has been consistent.

The optimal value of n and the corresponding memory-performance trade-off for SProxy ARP depend on the specifics of the network scenario, e.g. if we are dealing with software or hardware switches, with more constrained TCAM memory. A detailed evaluation of this is

beyond the scope of this thesis and represents future work.

6.7 Related Works

There have been a number of works that address the issue of controller load and scalability. Some of these address the problem by distributing the control plane functionality across multiple physical nodes, such as [78] and [156]. Kandoo [157] proposes a hierarchical controller architecture with two layers of controllers, one responsible for local control applications, dealing with frequent, low-level events. The other control layer is responsible for managing global aspects of the network.

More directly related to our work are [158] and [159]. Both papers consider the offloading of control plane functionality to the data plane (switches), and both papers discuss ARP handling as an example. However, the approach taken in these papers is fundamentally different to SProxy ARP. Both [158] and [159] require a significant modification to switches, and the offloading is achieved by additional software agents placed on the SDN switches. In contrast, our approach (SProxy ARP), is fully OpenFlow standard compliant and can be easily implemented in any controller and switch that supports OpenFlow version 1.3 and higher.

6.8 Conclusions

In this chapter, we presented SProxy ARP, a new approach to handling ARP traffic in SDN. Through offloading the ARP functionality from the SDN controller to the switches, we have demonstrated that SProxy ARP can achieve greater than the order of magnitude reduction in the ARP response time. In addition, our approach significantly reduces the controller overhead, thereby increasing network scalability and robustness, which are paramount to enhance security in SDN.

We have demonstrated how offloading the control plane functionality to the data plane is

achieved without requiring any modifications to the switch. SProxy ARP is completely OpenFlow standard compliant, which makes it a highly practical and easy to deploy in a production network. While the discussions in this chapter have focussed on ARP and hence IPv4, the proposed mechanism can be directly applied to IPv6 to provide the corresponding layer 3 to layer 2 address mapping via NDP (NS/NA), since the relevant protocol match fields are supported in OpenFlow. The implementation of SProxy ARP for IPv6, as well as for other SDN controller platforms, represents ongoing and future work.

Chapter 7

Evaluation of Denial of Service Attacks

7.1 Introduction

The SDN architecture provides a different Denial of Service (DoS) attack surface compared to traditional networks. In particular, the centralised controller represents an attractive target for DoS attacks. By bringing down the controller, an attacker can bring down, or at least significantly disrupt, the entire network.

DoS attacks are a significant problem in traditional networks, with an estimated annual cost of $113 billion globally [26]. With rapidly increasing deployment, it can be expected that SDNs will become the target of significant DoS attacks. It is therefore critical to investigate and understand the threats and the potential impact of these attacks. Towards this goal, this chapter presents an experimental evaluation of the impact of DoS attacks on SDN. We consider two types of DoS attacks, attacks against the control plane (SDN controller), as well as attacks against the data plane (switches). In both attacks, the attacker aims to exhaust the resources of the target. In the case of the control plane attack, this results in the inability of the controller to handle new flows and to install new forwarding rules reactively. In the attack against the data plane, we consider two different aims of an attacker, exhaustion of memory to store forwarding rules, as well as exhaustion of computing resources required to perform the packet forwarding. Since the memory exhaustion attack has been well studied in the literature [68, 160, 161, 162, 163, 164], we focus on the attack on the CPU resources of software SDN switches. This is relevant since software SDN switches running on commodity

x86 hardware are increasingly widely deployed.

In our experiments, we quantify the impact of these attacks on the ability of the SDN controller to handle regular network traffic, under varying attack rates. We perform our experiments for the following key SDN controller platforms, Ryu [34], ONOS [35] and Floodlight [36]. To the best of our knowledge, such as comparison has not been presented before.

Another contribution of this chapter is a discussion and investigation of the *amplification effect* of DoS attacks against the control plane, where the impact of the attack increases with the network size, i.e. the number of switches.

The rest of the chapter is organised as follows. Section 7.2 provides some basic background on SDN packet forwarding. Section 7.4 presents and discusses the results of our experiments. Section 7.5 gives an overview of key related works, and Section 8.8 concludes the chapter.

7.2 SDN Packet Forwarding

The OpenFlow protocol allows the controller to install forwarding rules on the switches via *flow-mod* messages. These rules follow a simple *match-action* paradigm, where the match part can consist of layer 2-4 packet header fields (supporting wild-cards), plus other parameters, such as the ingress port. Rules can be installed in a proactive or reactive approach. In the proactive approach, forwarding rules are pre-installed, prior to the arrival of any packets. In the reactive approach, which is more common, forwarding rules are installed on-demand. When a packet arrives at a switch, and there is no matching rule (a *table-miss* event), the packet is sent to the controller via an OpenFlow *Packet-In* message. The controller then sends the packet back to the switch in an OpenFlow *Packet-Out* message, with instructions on how to forward the packet. The controller also installs a set of corresponding forwarding rules on switches along the path, to handle any subsequent packets belonging to the new flow [2, 50]. A *table-miss* event and the corresponding OpenFlow messages are relatively expensive operations for a controller. As we will see later, this can be exploited by an attacker.

The exact behaviour of the controller in regards to handling flows is determined by the forwarding application running in the controller. Common SDN reactive forwarding applications supported by most controller platforms include a L2 learning switch and L3 shortest path routing. For our experiments, we use the default reactive forwarding application of the controllers we are considering, as discussed later.

7.3 DoS Attacks against SDN

As mentioned before, we consider two types of attacks in this chapter. The first attack aims to exhaust the resources of the SDN controller, while the second attack aims to exhaust the computing resources of a software switch, i.e. the data plane. In the following, we describe these attacks in more detail, based on a simple example scenario.

7.3.1 Attack on the Control Plane

We consider the example network topology shown in Figure 7.1, with three hosts attached to a single OpenFlow switch, which is in turn connected to a controller. The IP and MAC addresses of the hosts are as indicated.

Here, host *h1* is the attacker, whose aim is to create a maximum workload for the controller. This is achieved by creating a *table-miss* event for every single packet, resulting in the packet being sent to the controller in an OpenFlow *Packet-In* message, and taking up the controller computing resources in the process. In order to achieve this, the attacker spoofs the source IP and MAC addresses of the packets, by choosing the addresses uniformly randomly for each individual packet.[1] The destination IP and MAC addresses are also chosen randomly.

In our example, we assume that the controller runs a learning switch forwarding application, which is the default in most SDN controller platforms. Below is a step-by-step account of what happens during the attack.

[1] Choosing different source addresses for every packet makes simple countermeasures implemented by common controller platforms ineffective.

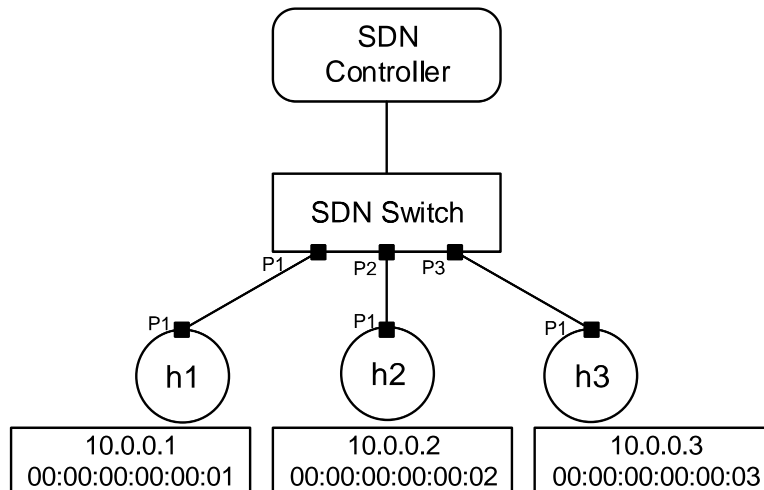


Figure 7.1: Basic Attack Scenario

1. The attacker *h1* crafts an IP packet (UDP or TCP), with randomly chosen source and destination IP and MAC addresses, and sends the packet to the switch, where it is received via port *P1*. Due to the randomly chosen addresses, there is no matching rule installed in the switch, which causes the packet to be sent to the controller.

2. The controller sends the packet back to the switch via an OpenFlow *Packet-Out* message, with an action which instructs the switch to flood the packet, i.e. send it out on all ports except the ingress port.[2]

3. The switch now sends the packet out on ports *P2* and *P3*, reaching hosts *h2* and *h3*, neither of which is the destination.[3] As a result, no response to the packet is sent, which also means no flow rule is installed in the switch.

By increasing the attack packet sending rate, the attacker can consume an increasing amount of controller resources, up to the point where it becomes unable to handle legitimate flows.

[2]SDN learning switches only install a forwarding rule once it has observed packets in both directions of the flow.

[3]With negligible probability.

7.3.2 Attack on the Data Plane

A widely discussed attack on the SDN data plane is via exhaustion of the TCAM memory in hardware SDN switches [165, 166]. In contrast, we consider the exhausting of computing resources in a software SDN switch and the resulting impact on its ability to forward legitimate packets.

In order to consider this kind of attack, we need to slightly change our scenario. Here, we only want to consider the impact of the attack on the switch's ability to forward packets, independently of the controller. We, therefore, assume that the relevant forwarding rules to handle legitimate traffic are pre-installed on the switch. This means these packets can be forwarded by the switch, independently of the controller, and we can isolate the attack's impact on the switch.

Attack packets are created by host *h1* in the same way as in the control plane attack scenario. As a result, the switch sends each attack packet to the controller, encapsulated in an OpenFlow *Packet-In* message. Sending this OpenFlow *Packet-In* message and receiving and processing the corresponding OpenFlow *Packet-Out* message from the controller also take significant computing resources at the switch. In a software switch, the available computing resources are shared between the packet forwarding and the processing of OpenFlow control messages. Therefore, the attack can result in the disruption of the forwarding capability of the switch. In the following section, we discuss our experiments in which we quantify the impact of these attacks, considering three different SDN controllers.

7.4 Experimental Evaluation

7.4.1 Testbed

For our experiments, we used Mininet [30] and OpenvSwitch (OVS) [31]. The controllers evaluated include three modern SDN controllers, i.e. Ryu [34], ONOS [35], and Floodlight [36]. We used the default forwarding applications in these controllers, which are *sim-*

Table 7.1: Software Tools used for Implementation Experiments in Chapter 7

Software	Function	Version
Mininet [30]	Network Emulator	2.2.2
Open vSwitch [31]	Software SDN Switch	2.5.2
OFELIA [32]	Hardware SDN Test-bed	=======
Ryu [34]	SDN Controller Platform	3.22
ONOS [35]	SDN Controller Platform	1.10.0
Floodlight [36]	SDN Controller Platform	1.0
Scapy Library [38]	Packet Manipulation Tool	2.2.0
Tcpreplay [41]	Traffic Replay Tool	4.2.6

ple_switch_13 in Ryu, *fwd* in ONOS and *forwarding* in Floodlight.

Initially, we used the Scapy library [38] to craft and send the attack packets. However, the maximum packet sending rate in this approach was limited to around 500 pkts/s. We, therefore, used an alternative approach where we created a pcap file with attack traffic prior to the experiment, and then used Tcpreplay [41] to inject the packets into the network at the desired rate. With this approach, we were able to achieve a packet sending rate of well above 70,000 pkts/s. Table 8.2 summarises the relevant software tools we used in the experiments of this chapter. All our experiments were carried out on a Dell server (PowerEdge R320 with a 12-core Xeon E5-2400 CPU and 32GB of RAM), running Ubuntu Linux 17.04 with kernel version 3.16.0. Each process (ovs-switch, controller, packet injection) was allocated to a dedicated CPU core to avoid any interference.

7.4.2 Control Plane Attack

In our first experiment, we want to measure the impact of a DoS attack on the controller's ability to handle legitimate traffic. For this, we consider the scenario shown in Figure 7.1. We run ping between hosts *h2* and *h3*, at a rate of 10 ICMP Echo requests per second and measure the packet delivery ratio (PDR). We slightly modified the controller behaviour to avoid the installation of forwarding rules for the ping traffic. This forces the ping packets to be sent to the controller via OpenFlow *Packet-In* messages and requires the controller's

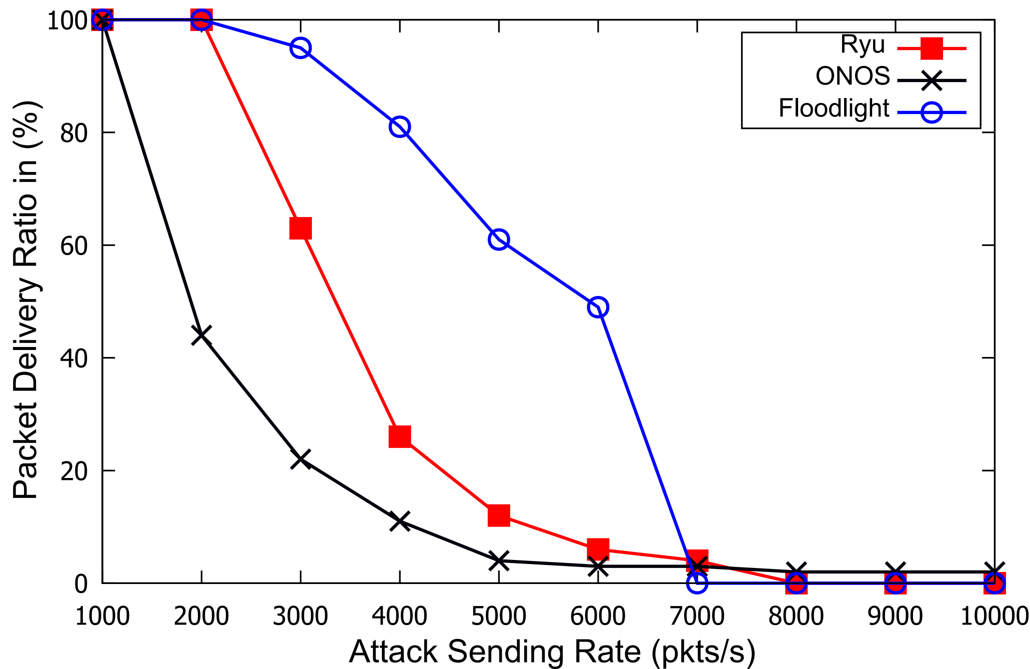


Figure 7.2: Control Plane Attack, PDR

involvement in order to forward the packets. We can, therefore, measure the impact of the DoS attack on the controller via the PDR of the ping packets, for a range of attack sending rates.

Figure 7.2 shows the PDR of network traffic between hosts *h2* and *h3* with different attack sending rate, ranging from 1000 pkts/s up to 10,000 pkts/s for the three considered controllers. We see that from the attack sending rate of around 7000 pkts/s, all the controllers are essentially overwhelmed with handling OpenFlow *Packet-In* and *Packet-Out* messages caused by attack traffic. As a result, they are unable to handle any legitimate traffic, and the PDR drops close to 0%. However, we see a significant difference between the three controllers.

The PDR for ONOS drops relatively sharply, starting from the attack sending rate of only 2000 pkts/s. Ryu performs slightly better, with the PDR drop starting from 3000 pkts/s. Floodlight appears to be the most resilient controller, with the ability to handle a much higher attack sending rate before it completely loses the ability to handle traffic.

We also measured the CPU load of the SDN controllers for the same range of the attack sending rates. Figure 7.3 shows the result. For all the controllers, we see a near-linear increase in the CPU load as a function of the attack packet rate. All controllers reach close

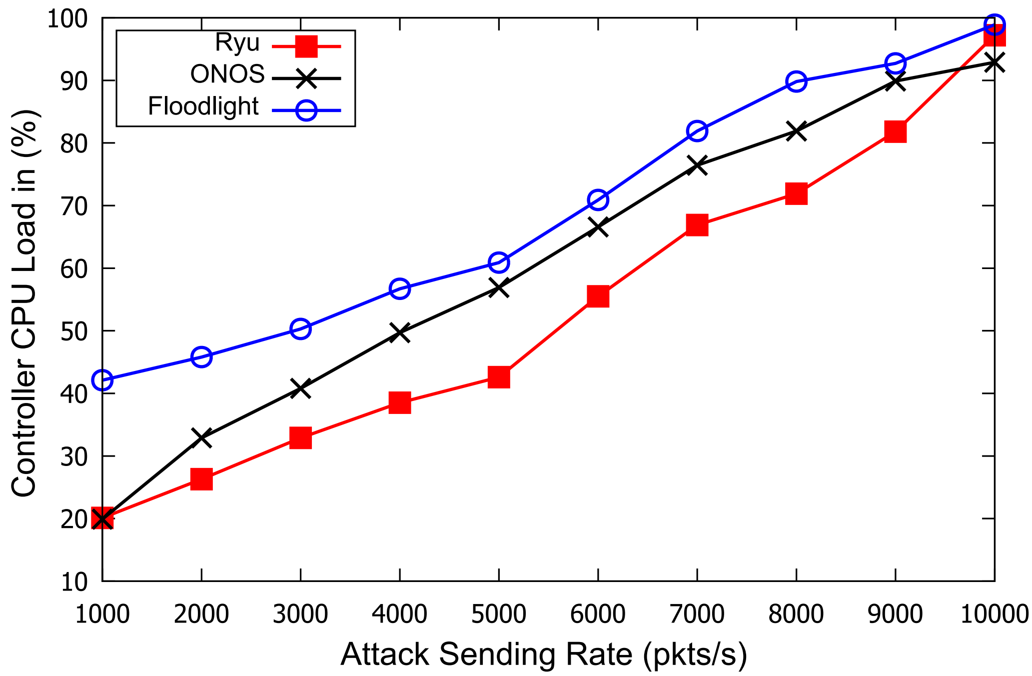


Figure 7.3: Control Plane Attack, Controller CPU Load

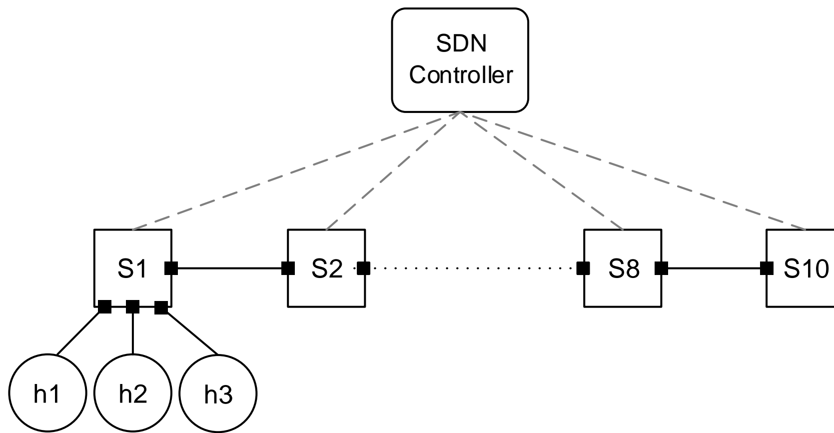


Figure 7.4: Linear Topology

to 100% for the attack sending rate of 10,000 pkts/s.

While there is a clear correlation between the CPU load and the PDR drop, we see that the CPU load by itself does not completely explain the drop in the PDR. In particular for ONOS, we see that the PDR drops well before the CPU load reaches 100%. We believe this is due to differences in the implementation of OpenFlow message handling among the controllers.

So far, we have only considered the simple network topology with a single switch, as shown in Figure 7.1. We now consider a larger topology in order to explore the effect of the network size on the impact of the DoS attacks. For the next experiment, we consider a linear topology

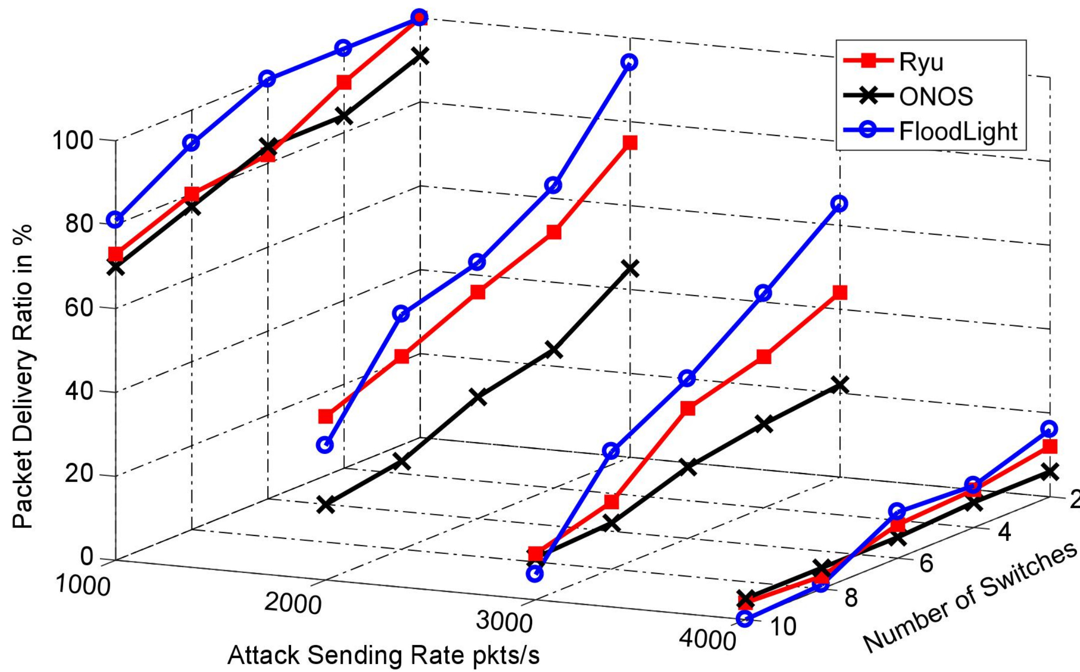


Figure 7.5: Attack Amplification Effect on PDR

with 2, 4, 6, 8 and 10 switches. The example with ten switches is shown in Figure 7.4. As before, the attacker *h1* injects UDP packets with random IP and MAC addresses at various attack sending rates, and we measure the impact of the attack via the PDR of legitimate traffic sent between hosts *h2* and *h3*. We are interested in how the network size, i.e. number of switches, affects the impact of the DoS attack.

Figure 7.5 shows the PDR results for the attack sending rates of 1000, 2000, 3000 and 4000 pkts/s on linear topologies of size 10, 8, 6, 4 and 2 switches. We reversed the order on the axis showing the number of switches to increase legibility.

We see that for all controller platforms, an increasing number of switches results in a more significant drop in the PDR, which corresponds to a substantial impact of the attack. We refer to this as the *attack amplification effect*.

As expected, in the case of reactive forwarding applications, each attack packet causes a pair of OpenFlow *Packet-In* and *Packet-Out* messages from each switch to be handled by the controller. For example, in the case of a learning switch forwarding application, if host *h1* sends an attack packet, it will cause a *table-miss* event at switch *S1*, with the packet being sent to the controller as an OpenFlow *Packet-In* message. Since the controller does not have sufficient information to install any flow rules at this stage, it will send it back to the switch

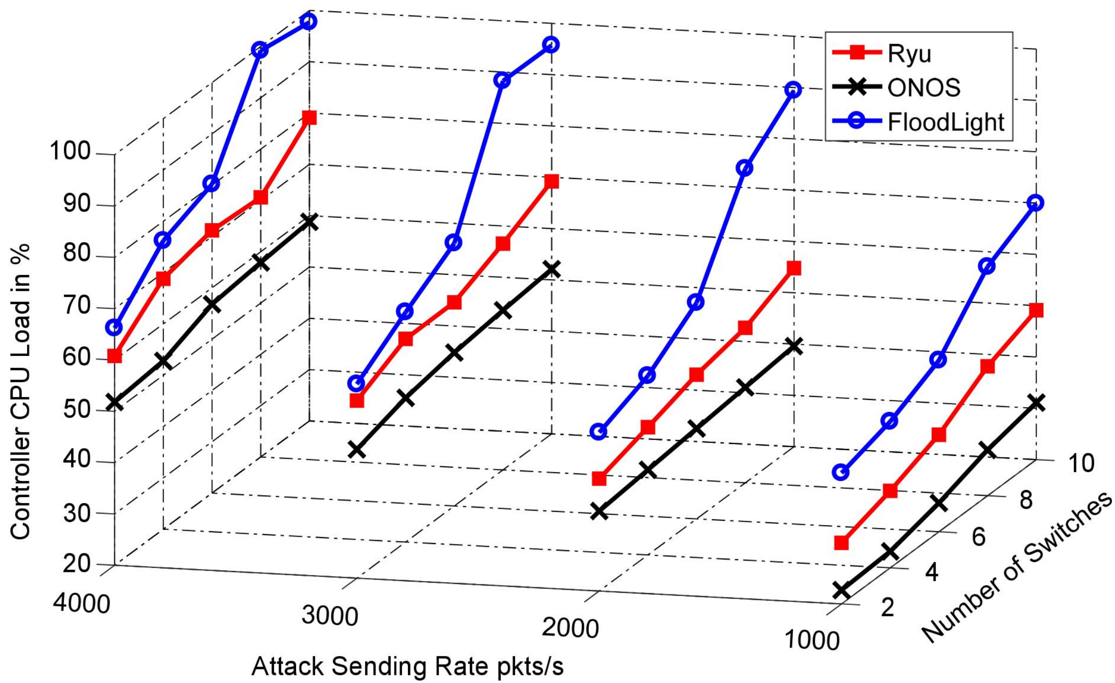


Figure 7.6: Attack Amplification Effect on Controller CPU Load

in an OpenFlow *Packet-Out* message, with instructions to flood it. As a result, the packet is sent to switch *S2*, where it again causes a *table-miss* event, with another OpenFlow *Packet-In* and *Packet-Out* message, and so forth. As we can see in the graph, the relationship between the network size and the PDR is roughly linear. We also observe differences in the attack impact between the three SDN controller platforms, which is consistent with our previous experiment for the basic topology. We see that ONOS suffers from the most notable impact of the attack, followed by Ryu and Floodlight.

As for the previous experiment, we also measured the controller CPU load, as shown in Figure 7.6.[4] As expected, we observe an increased controller CPU load for larger networks, for the same attack sending rate. In summary, we can say that the larger the network, the easier it is for an attacker to launch a successful attack on the controller, due to the attack amplification effect. This is in stark contrast to traditional networks, where the control plane is distributed.

[4]Note reverse order on the axis showing the number of switches.

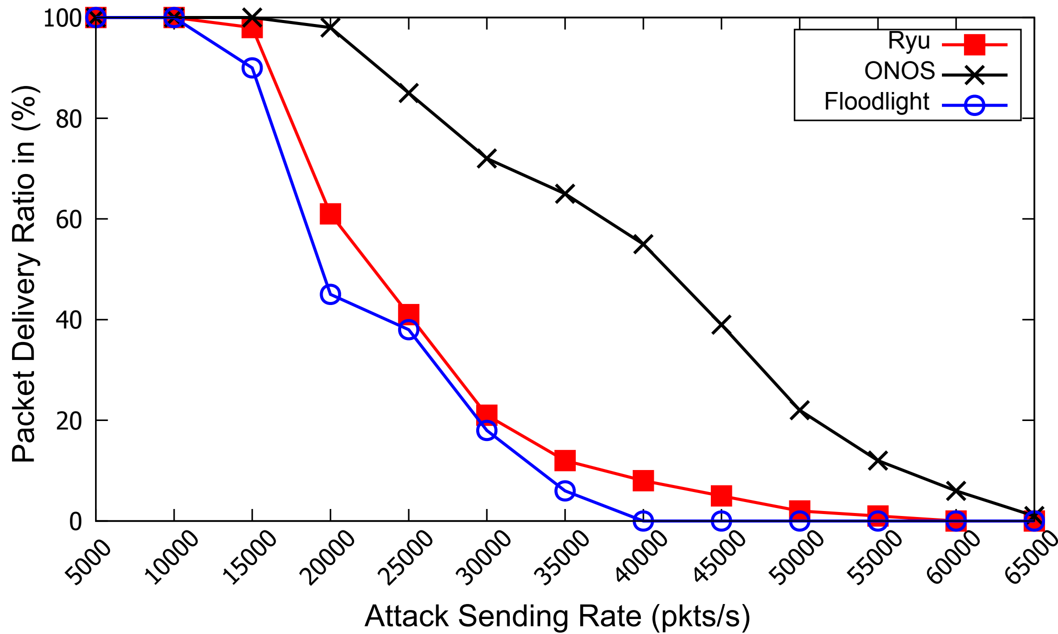


Figure 7.7: Data Plane Attack, PDR

7.4.3 Data Plane Attack

As mentioned before, the aim is to explore the impact of DoS attacks on the data plane. We particularly consider an attack that tries to exhaust the CPU resources of software SDN switches, where the computing resources are shared between both the handling of OpenFlow control messages as well as the forwarding of data packets. We are investigating what extent an attack, which overwhelms the switch with control message processing, can disrupt the switch's packet forwarding capability.

For this experiment, we used the basic scenario in Figure 7.1, with the same assumption that the attack packets are injected by host *h1*. Again, we measured the PDR of ping packets exchanged between hosts *h2* and *h3*. Here, in contrast to the attack on the control plane, we allow permanent forwarding rules for packets between hosts *h2* and *h3* to be installed proactively on the switch. This means that the controller is not involved in the forwarding of these packets, and any drop in the PDR can be directly attributed to the switch, i.e. the impact of the DoS attack on the switch.

Figure 7.7 shows the PDR values for the three SDN controller platforms, for different attack packet sending rates. We again performed the experiment for the three different controllers, ONOS, Ryu and Floodlight. These might seem strange, since we are attacking the switch,

and are not concerned about the controller in this experiment. However, we noticed that each controller reacts differently to the OpenFlow *Packet-In* messages sent by the switch. In particular, their rate of sending the corresponding OpenFlow *Packet-Out* messages back to the switch can vary significantly, which in turn results in varying loads on the switch.

We observe that for the attack to result in a close to 0% PDR for all controllers, an attack rate of more than 65,000 pkts/s is required. This is significantly higher than in the control plane attack scenario. A possible explanation for this is that OVS, which is written in C and implemented in the Linux kernel, is more efficient in handling OpenFlow control messages, compared to the controllers.

We also see that for the ONOS controller, a higher attack rate is needed to achieve the same PDR drop, compared to the other two controllers, Ryu and Floodlight. This does not necessarily show that the ONOS controller platform is better in that regard. The ONOS controller simply causes a lower load on the switch, due to its reduced rate of sending OpenFlow *Packet-Out* messages, and hence reduces the impact of the DoS attack on the switch. This is the result of a rate control mechanism that ONOS implements. In contrast, both Ryu and Floodlight have a higher rate of OpenFlow *Packet-Out* messages, and hence cause a higher load on the switch, resulting in a considerable PDR drop for lower attack rates.

We also measured the switch CPU load during the experiment, and the result is shown in Figure 7.8. As expected, we see an increase in the switch CPU load with an increasing attack sending rate.[5] Consistent with the results in Figure 7.7, we see that the load is smallest for the ONOS controller, due to its reduced rate of OpenFlow *Packet_Out* messages. As for the PDR results, the Ryu and Floodlight controllers behave similarly. We observe a clear correlation between the switch CPU load and the PDR drop of ping packets. It is interesting to note that a significant disruption of the switch's ability to forward packets, as measured via the PDR, starts to happen well before the switch CPU is fully saturated.

[5] To be more precise, this refers to a CPU core. As mentioned, before, OVS is allocated a dedicated CPU core, so that we can isolate the CPU load.

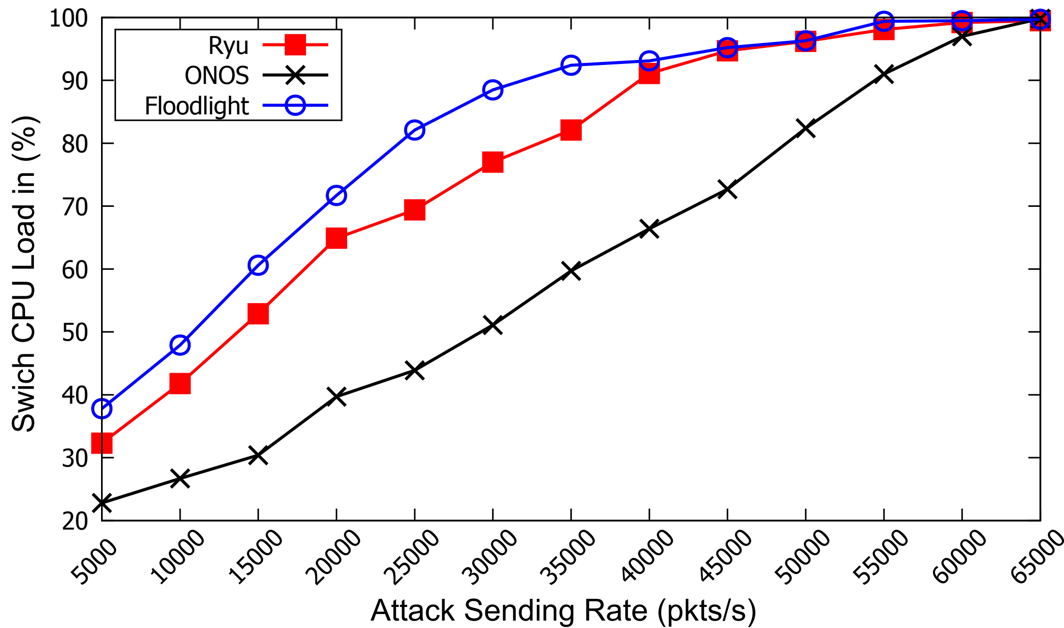


Figure 7.8: Data Plane Attack, CPU Load

7.5 Related Works

There are a number of papers that have considered the problem of DoS attacks in the context of SDN. This section provides a brief overview of key related works.

The authors of [105] propose a lightweight method to detect distributed DoS attacks against the control plane, based on an entropy-based anomaly detection approach. The entropy is calculated from destination IP addresses of incoming packets, and a simple threshold is used to detect an attack, i.e. an attack is detected if the entropy falls below the defined threshold. The proposed approach is not able to detect the attack proposed in this chapter, due to the fact that network traffic generated in our approach has maximum entropy since IP addresses are uniformly generated for each packet. Despite the title, which indicates DoS attacks against SDN controllers are considered, the main focus of the paper is on attacks against end-hosts in the context of SDN.

The paper [165] discusses DoS attacks in SDN, considering the exhaustion of control channel bandwidth as well as switch memory, which are both different from the DoS attacks considered in our work. The paper proposes mitigation strategies based on rate limiting as well as choosing optimal time-out values of flow rules.

FloodGuard [167] is a defence mechanism designed to protect the SDN controller from DoS attacks, in particular it aims to prevent resource exhaustion attacks against the switch-controller control channel. This is in contrast to our work, which specifically considers the attack on the computing resources of the SDN controller and switches. The paper further proposes a mitigation approach based on rate limiting control messages.

Similarly, [168] also consider DoS attacks against the SDN control channel. The authors introduce LineSwitch, which is a mitigation approach based probabilistic proxying and black-listing. To implement this, a special proxy module needs to be installed on edge switches. A potential hurdle to adoption is the fact that the approach requires a modification to the OpenFlow standard in order to be implemented. The paper does not provide a quantitative evaluation of the attack impact for different SDN controller platforms.

In [98], the authors present AVANT-GUARD, an OpenFlow switch extension with the aim to mitigate against DoS attacks in SDN. The basic idea is similar to LineSwitch. A key component in AVANT-GUARD is the connection migration module, which adds intelligence to the data plane to differentiate source nodes who likely complete the TCP connections from the ones who perform a TCP SYN flooding attack. The mechanism is implemented via a TCP proxy functionality deployed on the OpenFlow switches. The paper does not address the kind of simple volumetric DoS attacks that are discussed and demonstrated, and neither does it provide an experimental comparison of the attack impact for different SDN controller platforms.

In summary, there are a lot of related works on DoS attacks in SDN, proposing a number of attack detection and mitigation mechanisms. However, none of the works have presented a detailed experimental evaluation on the quantitative impact of DoS attacks against the computing resources of the SDN control plane and data plane. To the best of our knowledge, no paper has presented a detailed comparison of the impact of DoS attacks for the three SDN controller platforms considered in this chapter.

7.6 Conclusions

DoS attacks are a critical problem in traditional networks and are likely to become an even bigger one in SDN, due to the centralisation of network control functionality at the controller. It is therefore important to understand the types of DoS attacks that are possible against the SDN infrastructure and to have an understanding of the quantitative impact of the attacks. In this chapter, we presented detailed experimental evaluations of DoS attacks against SDN. We considered attacks against the control plane (SDN controller) as well as attacks against the data plane (switches). For the latter case, we considered an attack scenario that has not been well studied so far, i.e. an attack that aims to exhaust the computing resources of an OpenFlow-based software switch.

A key novel contribution of our work is the comparison of the impact of the DoS attacks for three different SDN controller platforms, i.e. Ryu, ONOS and Floodlight. While we found significantly different results for the different controllers, the overall conclusion is that an attacker controlling a single host can completely disrupt the forwarding capability of a network with relatively limited resources. We also discussed the attack amplification effect, which results in a roughly linear increase in the attack impact with an increase in the network size. This is an important consideration, especially for large-scale SDNs. It is also a major difference to traditional networks with a distributed control plane. We believe that our findings provide further insights into the problem of DoS attacks against SDNs, and can hopefully be used to inform the development of DoS mitigation mechanisms and countermeasures.

Chapter 8

Security of Virtualisation

8.1 Introduction

Network virtualisation (NV) is a key functionality enabled by SDN. NV allows multiple entities (tenants) to have their own individual virtual network, based on a shared physical network infrastructure. In the context of SDN, this means that multiple SDN controllers can run concurrently, each controlling its own dedicated and isolated virtual network. This approach essentially hides the underlying network complexity and the characteristics of the forwarding elements by subdividing or slicing the physical network into multiple virtual networks. It provides an abstraction layer that allows multiple logical networks to run simultaneously on the same physical infrastructure [169, 170, 171].

The main drivers behind the growth of NV include cost-effectiveness, network deployment speed and flexibility. Thus, users are free to efficiently and dynamically aggregate network resources and request different network services from the same underlying physical infrastructure without interfering with each other and worrying about the characteristics of underlying hardware infrastructure.

A key requirement of NV in regards to security is the maintenance of isolation of the different virtual networks. An attacker on one virtual network should not be able to bypass the virtualisation layer (hypervisor), and interact with and possibly disrupt nodes or controllers on other virtual networks. This is similar to the corresponding requirement in compute virtualisation

[172], which has been widely studied.

This chapter aims to provide an initial exploration of network virtualisation security in SDN that has received very limited or no attention. In particular, we consider FlowVisor [27] and OpenVirteX (OVX) [28], the two most relevant SDN hypervisor platforms commonly deployed in network test-beds, e.g. OFELIA [32], GENI [118], FITS [173], etc. OVX is an integrated component of the widely used ONOS SDN controller platform. We consider these network hypervisors together with key SDN controller platforms such as Ryu [34], ONOS [35] and Floodlight [36].

Our exploration identifies a number of significant security vulnerabilities in the current implementation of SDN hypervisors, which are either due to design flaws or implementation bugs. A further contribution is the practical demonstration of the feasibility of the attacks, and an evaluation of their potential impact.

The chapter is organised as follows. Section 8.2 provides a brief overview on the current state-of-the-art of network virtualisation mechanisms in SDN. Section 8.3 presents a classification of security threats towards network virtualisation in SDN, and Section 8.4 discusses related works. Section 8.5 briefly presents our experimental platform. Section 8.6 and Section 8.7 describe the vulnerability of the current SDN hypervisors, i.e. FlowVisor and Open-Virtex respectively, and demonstrate a number of potential attacks. Section 8.8 concludes the chapter.

8.2 SDN Hypervisor Platforms

8.2.1 FlowVisor

The first OpenFlow-based hypervisor platform that provides virtualisation for SDN is FlowVisor [27], which uses a typical multi-tenancy technique that enables multiple SDN controllers to share the hardware resources of a particular physical infrastructure. FlowVisor allows virtualisation of bandwidth, topology, traffic, device CPU and forwarding tables, with no modification applied to the control plane and the data plane. It acts as a proxy between forwarding

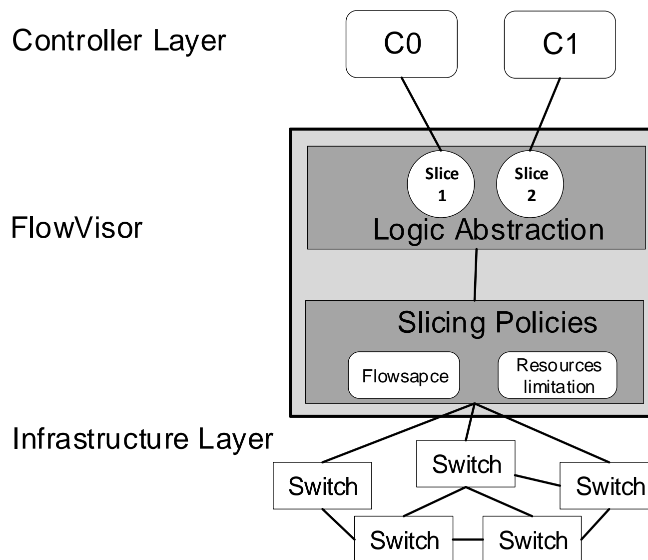


Figure 8.1: FlowVisor Architecture

elements and controllers and is, therefore, able to inspect and rewrite all OpenFlow messages sent between the control plane and the data plane. In this way, it can enforce isolation of the different network slices and can make sure that packets stay within their configured virtual networks. The key virtualisation mechanism in FlowVisor works via the slicing of the flow space, which is made up of the OpenFlow packet header bits. Therefore, slicing can be done based on IP addresses, MAC addresses, VLAN tags, etc.

Figure 8.1 illustrates the general FlowVisor architecture. The bottom layer, i.e. *Slicing Policies*, allocates a fraction of link bandwidth, network topology and number of forwarding entries per slice, and ensures no slice monopolises the entire hardware resources. The layer also determines where to forward a packet based on a set of flow rules, i.e. flow space. At the top of the FlowVisor architecture is the logic abstraction layer, which presents a logical copy of routers and switches, i.e. a virtually sliced networks to each of the different tenant controllers, configured individually based on information provided by the *Slicing Policies* layer.

Upon receiving an OpenFlow message from an SDN controller, FlowVisor parses the packet header and makes a policy check to decide which slice is the message belongs to. In this process, FlowVisor also verifies that the flow definition of the message is within the allocated flow space of the sender tenant. Packets are generally rewritten and forwarded to adhere the slice policy. For example, if slicing is done using VLAN tags, a packet sent from a controller

will be modified, and the corresponding VLAN tag will be set. One of the limitations of OpenFlow is that it only supports network slices with disjoint flow spaces.

8.2.2 OpenVirteX

OpenVirteX (OVX) [28] is a network hypervisor that takes virtualisation in SDN a step further. Similar to FlowVisor, it acts as a transparent proxy between OpenFlow switches and SDN controllers. The key improvement of OVX over FlowVisor is that it supports full flow space virtualisation, which means it supports multiple slices with overlapping flow spaces, e.g. IP and MAC address range, VLAN ID, etc.

OVX assigns each virtual network a global unique identifier, i.e. an ID (*tenant ID*) for each tenant. Instead of allocating packets based on flow space matching, the key limitation of FlowVisor, OVX places a new functionality at SDN edge switches, that explicitly rewrites the header fields on incoming packets into its own, unique format. This rewriting is reversed when packets are sent to the respective destination slices, i.e. hosts or controller.

The OVX architecture consists of two logical layers, *OVX virtual networks* and *OVX physical networks*, as shown in Figure 8.2. The OVX virtual networks layer provides tenants with a completely isolated virtual network, consisting of virtual switches and links. As a result, each tenant can specify its own, unique virtual network topology. The OVX physical networks layer provides a network topology, equivalent to the physical infrastructure and maintains the mapping of OpenFlow messages between OVX and the data plane. The information that allows this bridging is maintained in the OVXMap, which also records the (*tenant IDs*) to track the state of the OVX physical networks (e.g. network topology) and correspondingly update the OVX virtual networks.

Upon receiving a packet via an OpenFlow message from an SDN controller, OVX parsers the packet and maps it to the corresponding tenant ID, re-writes the header fields and validates that isolation is maintained between the different virtual networks. When OVX receives a message from an SDN switch, it uses the tenant ID to determine the corresponding virtual network, and therefore how the packet is to be handled [28].

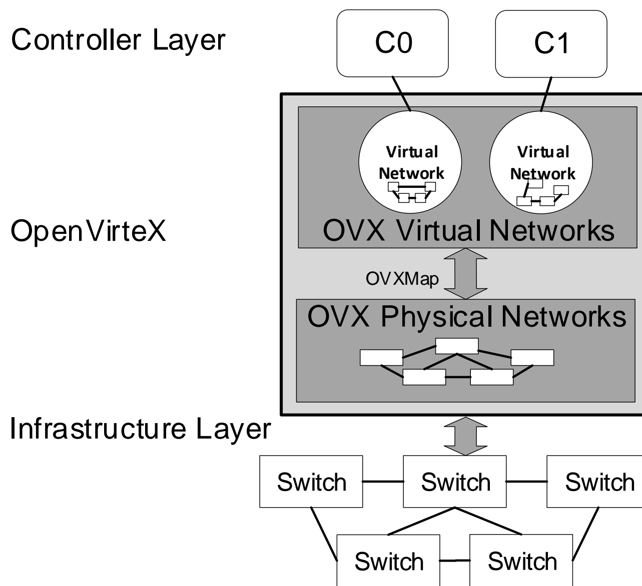


Figure 8.2: OpenVirteX Architecture

The key difference between OVX and FlowVisor lies in the flexibility of the topology customisation and addressing scheme, which has implications for traffic isolation. By leveraging OVX, each tenant is provided with a fully virtualised network and is free to implement the desirable topology regardless of the physical topology, and choose any addressing scheme for their hosts, regardless if there is an overlap with other virtual networks. However, in FlowVisor, tenants are only allowed to specify a topology that is isomorphic to the actual physical network topology. The entire flow space is essentially sliced into non-overlapping flow spaces.

In this chapter, we focus on the security of FlowVisor and OVX, since they are the most relevant and widely used SDN hypervisors. We also briefly discuss other SDN hypervisors and related proposals in the following section.

8.2.3 Other SDN Hypervisor Platforms

VeRTIGO [174] extends the network slicing techniques of FlowVisor and introduces additional abstraction features to improve and overcome the FlowVisor's limitation. With VeRTIGO, each SDN controller basically operates on a disjoint subset of network and is provided with a logical representation of the physical topology. Therefore, the full virtual network that includes virtual links and nodes is presented to the SDN controller.

Advanced FlowVisor (ADVisor) [175] is also an extension of FlowVisor that mainly relies on the tag-based virtualisation to distinguish between tenants. OpenFlow switches are programmed to add the VLAN ID for traffic entering the network.

Similarly, the FlowN architecture [176] is proposed to improve the tag-based virtualisation. FlowN is a container-based virtualisation that communicates directly with the tenant controller over a special OpenFlow API, rather than the OpenFlow protocol. Instead of simultaneously running multiple SDN controllers, FlowN allows only one SDN controller with multiple containers to share the kernel space with independent namespaces. Based on the VLAN ID, FlowN virtualises the physical network infrastructure and allows tenants to specify the addressing scheme and the network topology. FlowN basically tags traffic entering the network with the VLAN ID and removes the VLAN ID when traffic leaves the network. The key benefit of FlowN is that the mapping of OpenFlow messages traverse the control channel between the physical and virtual networks is no longer required, resulting in the reduction of the memory and computing overhead.

AutoSlice [177] is an SDN virtualisation layer that distributes the functionality of the hypervisor through the segmentation of the physical infrastructure into multiple SDN domains. Each SDN domain includes a controller proxy to essentially manage OpenFlow messages between the SDN controller and switches. To optimise network resource utilisation, AutoSlice dynamically assigns virtual sources to each separate SDN domain.

AutoVFlow [178] is an extension of AutoSlice that grants the tenants of wide-area networks a full control of their own virtual SDNs. In contrast to AutoSlice, where the flow space is shared, and controller tenants are restricted to the permissible header values of data packets, AutoVFlow provides each controller proxy of each SDN domain with the ability to freely utilise the entire flow space, similar to OpenVirteX, i.e. allowing overlapping flow spaces.

8.3 SDN Virtualisation Vulnerabilities

Most security vulnerabilities in software systems arise from either improper design or implementation bugs, i.e. software flaws, due to the fact that the system designer and the software programmer are humans and can make mistakes [179]. As a result, attackers can readily exploit the software bugs and produce unpredictable inputs to passively modify the system behaviour as desired. Despite the fact that Heartbleed [180], the most impactful security bug in OpenSSL's history, was simple and uncomplicated to remediate, the impact was extremely severe. In general, software flaws are notoriously difficult to discover and are often the root cause of major system disruptions and outages. Therefore, writing bug-free and reliable software remains a critical challenge [181]. Due to the 'softwareisation' of networks in SDN, the problems of software design and implementation flaws become an increasingly critical security threat. SDN controllers are complex software systems, with a significant potential for flaws and bugs.

In virtualised SDN networks, the network hypervisor appears as a controller to the data plane and as forwarding elements to the control plane, which in this case represents a single point of failure. By successfully attacking the SDN network hypervisor, an attacker can disrupt or potentially bring down the entire network, including all virtual networks controlled by different tenants. This provides a key motivation for the work presented in this chapter.

Table 8.1 provides a brief summary with representative examples of threat categories applied to the SDN virtualisation layer. It is apparent from the table that the attacker can masquerade and falsify packets information of other tenants running on a completely separate network simply after breaking the isolation mechanism. As a result, we believe that applying an effective and efficient testing mechanism to any SDN controller platforms is crucial and should be conducted through the software development life cycle. For our security analysis of SDN hypervisors, we used a combination of code analysis and fuzz testing [29].

Table 8.1: Classification of Network Virtualisation Threats

Threat categories	**Definitions**	**Examples in SDN Virtualisation**
Unauthorised Disclosure	An unauthorised user gains access to protected information	A malicious tenant intercepts network traffic that belongs to other tenants
Deception	An authorised user receives that data is being altered without the user's knowledge	A malicious tenant breaks the isolation mechanism and modifies network packets of other tenants
Disruption	Interrupt the communication and cause a system failure	A malicious tenant launches Denial of Service (DoS) attacks against other tenants

8.4 Related Works

In this section, we particularly focus on describing the most relevant work on the area of network virtualisation security in the SDN context. We only found a very limited number of works, which specifically consider the security of current SDN hypervisor platforms.

The paper [182] briefly mentions potential vulnerabilities in FlowVisor that can violate the isolation mechanism through the VLAN ID and rewriting fields. The proposed solution is an independent extension of FlowVisor, which basically limits the number of actions supported in the OpenFlow protocol. The paper does not provide clear technical details on how the VLAN ID and rewriting fields can break the isolation mechanism, as provided in this chapter. The paper also does not discuss or evaluate the impact of those vulnerabilities, nor does it investigate other vulnerabilities in FlowVisor.

The authors of [183] mention a potential attack against FlowVisor, where it is assumed that an administrator configures virtual networks with overlapping flow spaces. The 2-page paper lacks details and does not provide an evaluation of the potential impact of the attack.

Existing work on the security analysis of FlowVisor is very limited. Furthermore, to the best of our knowledge, there is no previous work on vulnerability assessment of OpenVirteX.

Table 8.2: Software Tools used for Implementation and Experiments in Chapter 8

Software	Function	Version
Mininet [30]	Network Emulator	2.2.2
Open vSwitch [31]	Software SDN Switch	2.6.1
Ryu [34]	SDN Controller Platform	3.22
ONOS [35]	SDN Controller Platform	1.11.1
Floodlight [36]	SDN Controller Platform	1.0
FlowVisor [27]	SDN Hypervisor	1.2.0
OpenVirteX [28]	SDN Hypervisor	branch 0.0-MAINT
Netcat [43]	Network Sniffing Tool	5.59*BETA1*

8.5 Experimental Platform

For all our experiments discussed later in this chapter, we used Mininet [30] and Open vSwitch (OVS) [31]. The standard network topology used in our experiments consists of three OpenFlow switches with two hosts attached to each switch, as shown in Figure 8.3. Each host is assigned to a different virtual network.

We used FlowVisor and OVX as our SDN hypervisors to provide isolation that allows tenants, *Tenant 1* and *Tenant 2* to run two virtual networks in parallel, i.e. *Virtual Network 1* and *Virtual Network 2* over the same physical network infrastructure. Each virtual network runs its own tenant controller.

The SDN controller platforms we considered to run on top of *Virtual Network 1*, i.e. *Tenant 1* are Ryu [34], ONOS [35], and Floodlight [36], with the default forwarding applications, *simple_switch* in Ryu, *fwd* in ONOS and *forwarding* in Floodlight. We further used Netcat [43], a network tool for collecting network traffic. Table 8.2 summarises the relevant software tools we used in the experiments of this chapter.

All our experiments were conducted on a Dell server (PowerEdge R320 with a 12-core Xeon E5-2400 CPU and 32GB of RAM), running Ubuntu Linux 17.10 with kernel version 3.16.0. To minimise interference, we allocated each key process (ovs-switch, controllers, FlowVisor, OVX) to a dedicated CPU core. The following sections present our results and a demonstra-

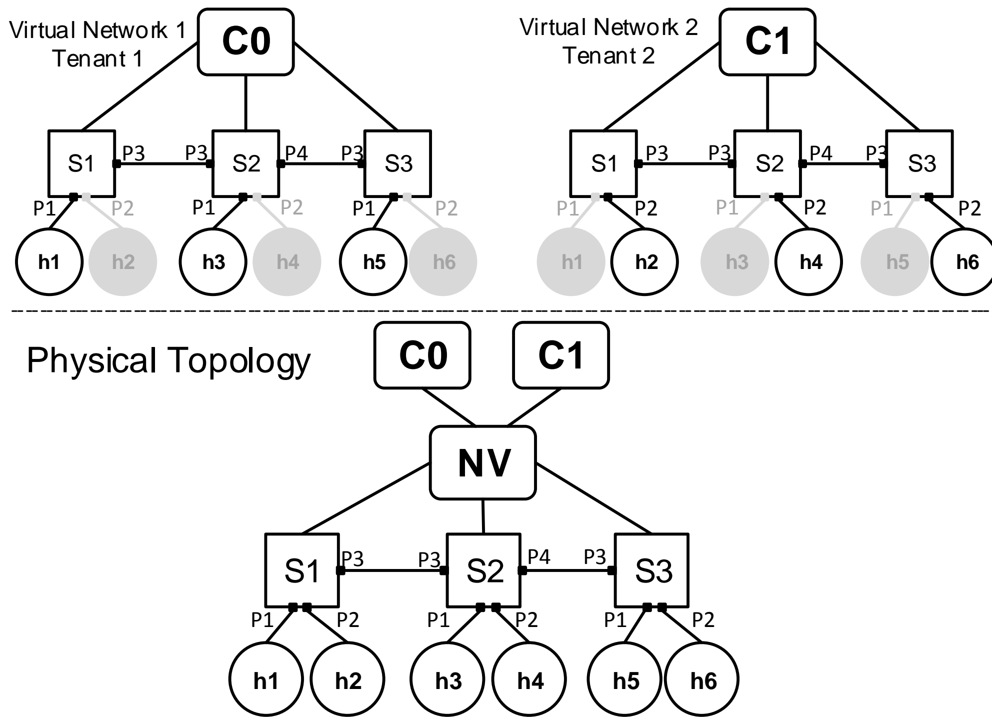


Figure 8.3: Network Infrastructure

tion of the feasibility of attacks using the identified vulnerabilities in FlowVisor and OVX.

8.6 Security of FlowVisor

By analysing the source code of FlowVisor and using fuzz testing, we exposed previously unknown security vulnerabilities that make the system susceptible to various attacks. In the following, we discuss these vulnerabilities and demonstrate how they can be exploited.

8.6.1 Topology Discovery

FlowVisor discovers the topology of the physical network infrastructure via utilising the OpenFlow Discovery Protocol (OFDP) and builds its own link database based on the information provided in the corresponding LLDP packets. FlowVisor relies on this database to provide tenant controllers with the underlying topology. It mainly precludes any SDN controller running the topology discovery component from directly retrieving the actual topology of

Preamble	Dst MAC	Src MAC	Ether-type: 0x88CC	Chassis ID TLV	Port ID TLV	Time to live TLV	Open Netw TLV	Opt. TLVs	End of LLDPDU TLV	Trailer	Frame check seq.

Figure 8.4: FlowVisor LLDP Frame Structure

the physical network itself. For example, when a tenant controller runs topology discovery, which sends its own LLDP packets, FlowVisor intercepts them, and creates LLDP packets in response, in order to 'emulate' the virtual topology that this controller is supposed to see.

The format of the LLDP frame used in FlowVisor is similar to the original LLDP format, described earlier except that FlowVisor adds a trailer to its own LLDP packets as shown in Figure 8.4. The trailer consists of two new fields, the first one carries the *"FlowVisor"* name and the second one carries the *"Slice"* name, to distinguish between LLDP packets generated by FlowVisor and LLDP packets generated by SDN controllers running their own topology discovery. The basic security problem with the current implementation of OFDP is that it is fundamentally insecure due to the lack of any authentication and integrity protection mechanism, as demonstrated previously.

The trailer that FlowVisor adds to the LLDP packets does not add any security mechanism, and hence any LLDP packet with a trailer is accepted by FlowVisor for processing. The information provided in the packet is used to update the FlowVisor's link database. As a consequence, it is relatively easy for an attacker to inject a fabricated LLDP packet, resulting in a poisoning of the topology information of the FlowVisor database.

The method of the attack is essentially the same as the one discussed previously in Chapter 4, but the key difference here is that we are attacking the SDN hypervisor, instead of the SDN controller. As a result, all virtual networks and tenants are impacted by the attack and are presented with invalid network topology information.

To demonstrate the feasibility of the attack, we used the virtual network test-bed and the network topology described in Section 8.5. For this example scenario, FlowVisor is configured to slice the network based on the VLAN ID. In this case, we have two virtual networks, *Virtual Network 1* with VLAN ID 100, and *Virtal Network 2* with VLAN ID 200, and each virtual

```
ubuntu@sdnhubvm:~[18:45]$ fvctl list-links
Password:
[
  {
    "dstDPID": "00:00:00:00:00:00:00:01",
    "dstPort": "2",
    "srcDPID": "00:00:00:00:00:00:00:03",
    "srcPort": "1"
  },
  {
    "dstDPID": "00:00:00:00:00:00:00:03",
    "dstPort": "3",
    "srcDPID": "00:00:00:00:00:00:00:02",
    "srcPort": "4"
  },
  {
    "dstDPID": "00:00:00:00:00:00:00:02",
    "dstPort": "4",
    "srcDPID": "00:00:00:00:00:00:00:03",
    "srcPort": "3"
  },
  {
    "dstDPID": "00:00:00:00:00:00:00:02",
    "dstPort": "3",
    "srcDPID": "00:00:00:00:00:00:00:01",
    "srcPort": "3"
  },
  {
    "dstDPID": "00:00:00:00:00:00:00:01",
    "dstPort": "3",
    "srcDPID": "00:00:00:00:00:00:00:02",
    "srcPort": "3"
  }
```

Figure 8.5: FlowVisor Database Attack

network runs its own ONOS SDN controller instance.

For this experiment, we assume that the attack is generated via host *h2*, which injects an LLDP packet with the structure shown in Figure 8.4, aiming to fabricate a fake link from switch *S1* on port *P2*, and switch *S3* on port *P1*.

As mentioned, the attack steps are quite similar to the attack discussed in Chapter 4, except that we need to include the relevant FlowVisor LLDP trailer, i.e the *"magic flowvisor1"* and *"fvadmin"* fields.

When the packet arrives at switch *S1*, which adds its own *Chassis ID* and *Port ID*, according to its pre-defined rule, it forwards the packet to the controller, encapsulated in an OpenFlow *Packet-In* message. FlowVisor intercepts the packet and updates its links database based on the information provided in the payload of the received LLDP packet.

Figure 8.5 shows the FlowVisor link database after launching the attack.[1] Each group represents detailed information, e.g. source and destination of ports and switches of a unidirectional link. For example, the first line (in bold) indicates that there is a unidirectional link between switch *S3* on port *P1* and switch *S1* on port *P2*. Hence, the attack has been successful, since such a link does not exist in the topology. This is the network information that FlowVisor will provide to the tenants when they query the underlying topology, resulting in a poisoning of all controllers' topology information. The potential impact of this has been discussed in detail in Chapter 4.

8.6.2 Breaking Isolation

Another security problem with the current design of FlowVisor is that there is no detailed security check implemented to investigate the content of packet sent from a controller to a switch in the form of an OpenFlow packet. In particular, FlowVisor fails to properly check the OpenFlow actions (action list) associated with the packet and eventually installed on switches.

This is a problem, if the action list contains a *'set-filed'* action, which provides direct access to the header fields of a packet and overwrites specified fields with arbitrary values. Therefore, it is relatively easy for an attacker to redirect his own traffic to another virtual network. This is particularly simple if the network is sliced based on the VLAN IDs, which the attacker can alter by adding a *set_VLAN ID* action to a match rule.

To experimentally demonstrate this vulnerability, we used the same network scenario as mentioned above. We assume that the attack comes from *Virtal Network 2* (*Tenant 2*), and the aim is to break FlowVisor's isolation and inject network traffic into *Virtual Network 1* (*Tenant 1*).

The attack can be broken down into the following steps:

1. Controller *C1* initially installs an OpenFlow rule with a high priority on all switches, switch *S1*, switch *S2* and switch *S3*, to match on the VLAN ID of 200, which explicitly

[1] This is obtained by a FlowVisor command, i.e. *fvctl list-links*.

matches all packets sent by any hosts, belonging to its own network, i.e. *Virtal Network 2*. The corresponding action list associated with this rule has two actions: (1) set the VLAN ID to 100, and (2) forward the packet to the controller. In this case, all packets traversing through *Virtual Network 2* are redirected to the controller *C0* of *Virtual Network 1*.

2. The attacker now needs to generate traffic either from a hosts (*h2*, *h4* or *h6*) or from the controller *C1*. For our experiment, we assume that the attack is generated via the controller, by injecting an ARP packet, encapsulated in an OpenFlow *Packet-Out* message at different sending rates. The problem with this scenario is that the OpenFlow rule installed previously to modify the VLAN ID is not executed, and hence the packet is treated according to the action list associated with this packet. The simple solution to this is to set the action of the output port to *OFPP_TABLE*, which allows the switch to handle the packet as if it was received via any of the switch's regular ports, and processes the packet according to the rules in its forwarding table. In this case, we can ensure that involved switches rewrite the VLAN ID from 200 to 100 and then send the packet to the controller.

3. FlowVisor receives the packet, and based on the VLAN ID, it forwards the packet to the switches *S1*, *S2* and *S3*. Each switch receives a copy of the packet and performs the actions as specified in the forwarding table, which includes modifying the VLAN ID from 200 to 100 and sending the packet back to the controller, encapsulated in an OpenFlow *Packet-In* message.

4. FlowVisor receives three OpenFlow *Packet-In* messages and checks the VLAN ID to know where to forward the packets to, either *Virtual Network 1* or *Virtual Network 2*. Since the VLAN ID is 100 and FlowVisor is unable to detect the VLAN ID modification, the packets are forwarded to *Virtual Network 1*, and hence the attack is successful.

Being able to direct traffic to a controller of a foreign virtual network can be used for DoS attacks. To quantify the severity of such an attack, we consider the CPU load on the target controller *C0*, caused by the processing the DoS packets, i.e. the process of parsing received OpenFlow *Packet-In* messages, generated by the malicious controller *C1*, as well as transmitting the corresponding OpenFlow *Packet-Out* messages.

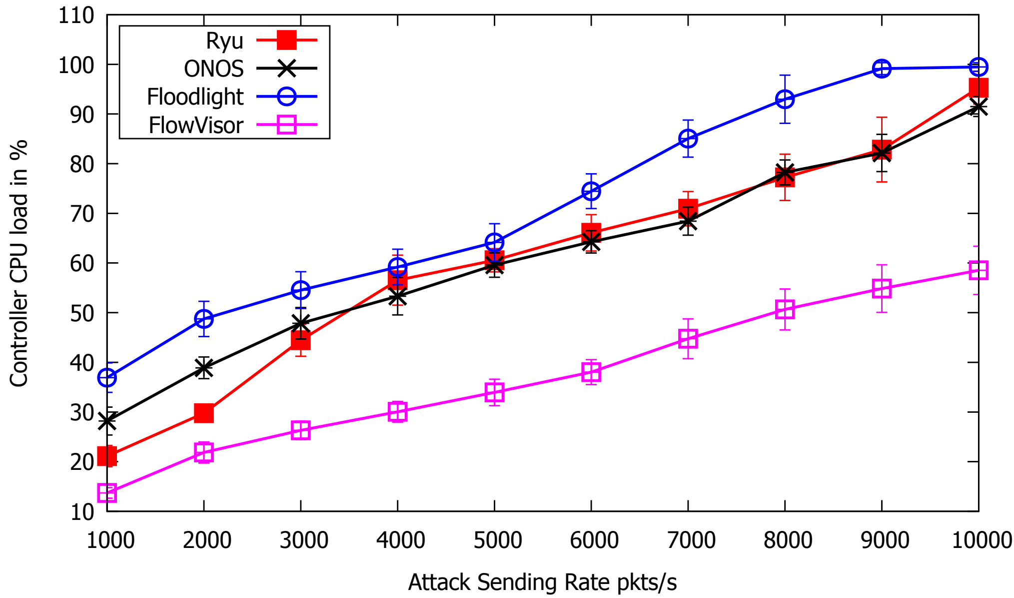


Figure 8.6: Controller CPU Load

Figure 8.6 shows the controller CPU load under the attack, depending on the attack sending rate, ranging from 1000 to 10,000 pkts/s. We repeated the experiment with three SDN controller platforms, i.e. Ryu, ONOS and Floodlight, and the corresponding CPU load values are shown in the graph. In addition, the graph also shows the CPU load for FlowVisor. All experiments were repeated 15 times, and the graph shows the mean value as well as the 95% confidence intervals.

As can be seen in the figure, with an increase in the attack sending rates, the controller CPU load increases roughly linearly. In this scenario, the attacker is able to saturate the controller CPU with the rate of 10,000 pkts/s, which can be done with relatively minimal effort. The attacker can achieve the same impact with a much lower attack rate, by adding multiple forward actions to the action list of the OpenFlow *Packet-Out* messages. For example, if each OpenFlow *Packet-Out* message contains 100 forward actions, the impact is multiplied by a factor of 100. This amplification potentially allows attackers with relatively minimal computing resources and bandwidth to saturate high-powered controllers. As mentioned, the attack can equally be launched from any of the hosts.

8.6.3 Ping of Death

Using fuzz testing, we also found that a single malformed message can crash FlowVisor. We refer to this attack as 'Ping of Death'. The result of this is quite severe since the entire network is disabled.

For the attack, we create an LLDP packet, with the format shown in Figure 8.4, but simply without a trailer. For our experimental evaluation of the attack, we used our previous scenario.

The attack can be generated either from the tenant controller or any of the hosts. For this experiment, we assume that the attack comes from *Virtual Network 2* (*Tenant 2*), and is generated via our controller component running as the tenant controller of *Virtual Network 2*.

The impact of this attack is only on FlowVisor, and there is no need to consider different SDN controller platforms for *Virtual Network 1*. In this scenario, we used ONOS as the tenant controller of *Virtual Network 1*. The controller *C1* injects the attack LLDP packet, encapsulated in an OpenFlow *Packet-Out* message and forwards it to the switch. FlowVisor intercepts the packet and parses the packet header.

As a result, FlowVisor immediately throws an exception and shuts down, as shown in the FlowVisor log messages in Figure 8.7. Since FlowVisor represents a single point of failure, the entire network is taken down as a result, with potentially severe consequences.

We found this vulnerability using fuzz testing with a relatively small effort. It is likely that more extensive investigation would reveal further bugs and vulnerabilities. Our example highlights the importance of secure coding practices for critical network infrastructure services such as hypervisors in SDN.

```
CRIT:2017-04-17T22:25:19.549:none:: MAIN THREAD DIED!!!
CRIT:2017-04-17T22:25:19.552:none:: restarting after main thread died
WARN:2017-04-17T22:25:19.552:TopoDiscovery:: shutting down
WARN:2017-04-17T22:25:19.552:topoDpid=00:00:00:00:00:00:00:01:: shutting down
WARN:2017-04-17T22:25:19.553:topoDpid=00:00:00:00:00:00:00:03:: shutting down
WARN:2017-04-17T22:25:19.553:topoDpid=00:00:00:00:00:00:00:02:: shutting down
WARN:2017-04-17T22:25:19.553:TopoDiscovery:: shutting down
WARN:2017-04-17T22:25:19.553:classifier dpid=00:00:00:00:00:00:00:01::tearing down
WARN:2017-04-17T22:25:19.554:classifier dpid=00:00:00:00:00:00:00:03::tearing down
WARN:2017-04-17T22:25:19.556:classifier-dpid=00:00:00:00:00:00:00:02::tearing down
```

Figure 8.7: FlowVisor Crash, Ping of Death

8.7 Security of OpenVirteX (OVX)

We conducted the same basic approach to analyse the security of OVX as we did for FlowVisor. In the following, we discuss the results.

8.7.1 Topology Discovery

Like FlowVisor, OVX utilises OFDP to discover the underlying topology to build its own topology database. By intercepting LLDP packets sent by tenant controllers' topology discovery service, and by creating the corresponding responses, OVX let the controllers see any arbitrary virtual network topology.

The format of the LLDP packets used in OVX is similar to the format used in FlowVisor, and in the original LLDP format, with only two additional TLVs, as shown in Figure 8.8.

The first TLV field, *OpenNetw1 TLV*, carries the *OpenVriteX* name, while the second one, *OpenNetw2 TLV*, carries the *Switch ID*. Upon receiving an LLDP packet from a switch, OVX processes the packet and extracts link information, i.e. Switch ID (the value specified in the *OpenNetw2* TLV), and Port ID from the payload of the packet. The link information is stored in the OVX database.

As mentioned previously, the current implementation of OFDP is insecure, and adding extra TLVs fields cannot protect topology discovery from the link spoofing attack. The attacker is still able to craft an LLDP packet that includes two additional TLV fields to deceive OVX to

Preamble	Dst MAC	Src MAC	Ether-type: 0x88CC	Chassis ID TLV	Port ID TLV	Time to live TLV	Open Netw1 TLV	Open Netw2 TLV	Opt. TLVs	End of LLDPDU TLV	Frame check seq.

Figure 8.8: OpenVirteX LLDP Frame Structure

poison the topology information in the OVX database.

We demonstrated the vulnerability of OVX against the topology poisoning or link fabrication attack via a simple experiment, via our standard experiment scenario used for FlowVisor. For this, we created two virtual networks, *Virtual Network 1* and *Virtual Network 2*. Similar to the FlowVisor case, the impact of this attack does not depend on the controller, and thus we used ONOS in both virtual networks.

For this experiment, we used the assumption that the attack is launched via host *h2*, which belongs *Virtual Network 2* controlled by malicious controller *C1*. The attacker aims to fabricate a fake link from switch *S1* on port *P2* to switch *S3* on port *P2*. The only difference in the attack steps to the FlowVisor case is that the *OpenNetw1* field included "*OpenVirteX*" and the *OpenNetw2* field included the sender switch ID, which in this case is the ID of switch *S3*. This makes the packet looks like it was sent out on port *P2* of switch *S3*. The packet is forward to the controller by switch *S1*, encapsulated in an OpenFlow *Packet-Out* message that includes the *Chassis ID* and the *Port ID* of switch *S1*. Upon receiving the packet, OVX extracts the link information and incorrectly updates its links database.

Figure 8.9 shows the OVX topology database after launching the attack.[2] The highlighted line (in bold), which appears after the attack is performed, indicates the success of the attack, and there is a physical link from *S3, P1* to *S1, P2*, which is incorrect. As mentioned in the case of FlowVisor, poisoning the hypervisor topology database will also poison the topology view of all tenant controllers, potentially resulting in the disruption of network operation and packet forwarding.

[2]This is obtained through the OVX command line interface, i.e. the *getPhysicalTopology* command.

```
ubuntu@sdnhubvm:~/OpenVirteX/utils[21:15] (0.0-MAINT)$ python
ovxctl.py getPhysicalTopology
Password:
{"switches": ["00:00:00:00:00:00:00:03", "00:00:00:00:00:00:00:01",
"00:00:00:00:00:00:00:02"], "links": [{"linkId": 3.0, "dst":
{"port": "3", "dpid": "00:00:00:00:00:00:00:01"}, "src": {"port":
"3", "dpid": "00:00:00:00:00:00:00:02"}}, {"linkId": 2.0, "dst":
{"port": "4", "dpid": "00:00:00:00:00:00:00:02"}, "src": {"port":
"3", "dpid": "00:00:00:00:00:00:00:03"}}, {"linkId": 1.0, "dst":
{"port": "3", "dpid": "00:00:00:00:00:00:00:03"}, "src": {"port":
"4", "dpid": "00:00:00:00:00:00:00:02"}}, {"linkId": 4.0, "dst":
{"port": "2", "dpid": "00:00:00:00:00:00:00:01"}, "src": {"port":
"1", "dpid": "00:00:00:00:00:00:00:03"}}, {"linkId": 0.0, "dst":
{"port": "3", "dpid": "00:00:00:00:00:00:00:02"}, "src": {"port":
"3", "dpid": "00:00:00:00:00:00:00:01"}}]}
```

Figure 8.9: OpenVirteX Database Attack

8.7.2 Breaking Isolation

As discussed in the previous section, OVX's topology discovery mechanism is vulnerable to the poisoning attacks, where an attacker can fabricate fake links. Here, we show how this can be exploited to break the isolation mechanism between two virtual networks. Using this attack, we can create a fake link between *S1, P2* and *S3, P1*, using our standard scenario. We use the fact that tenant controllers build their topology database based on the (poisoned) topology information provided by OVX, thereby including the fake link between *S1, P2* and *S3, P1*.

While this does not allow us to inject traffic from one network to another, it still allows the attacker *h2* to passively observe traffic from *Virtual Network 2*, traversing through the fake link. In order to achieve this, the attacking host *h2* simply needs to be disconnected from its network, and its interface needs to be set to promiscuous mode.

To show the ability to break the isolation and intercept network packets, we established a TCP connection between hosts *h1* and *h5*, which both belong to *Virtual Network 1*. In parallel, we ran *Netcat* [43] on the attacker *h2* that belongs to *Virtual Network 2* to collect network traffic transmitted between hosts hosts *h1* and *h5*. As a result, all packets from host *h1* destined to host *h5* pass through host *h2*, as shown in the first two lines of Figure 8.10. We also ran *ping* between the hosts, and the corresponding ICMP packets were also seen by host *h2*, as shown in the last 7 lines of the figure.

```
23:38:34.554139 IP 10.0.0.1.57246 > 10.0.0.5.1234: Flags [S],seq
1288162783, win 29200, options [mss 1460,sackOK,TS val1336219ecr
0,nop,wscale 9], length 0
23:38:35.858258 IP 10.0.0.1.57246 > 10.0.0.5.1234: Flags [S],seq
1288162783, win 29200, options [mss 1460,sackOK,TS val1336721ecr
0,nop,wscale 9], length 0
23:38:36.275558  IP 10.0.0.1  >  10.0.0.5: ICMP  echo request,id
29105, seq 1, length 64
23:38:37.284092  IP 10.0.0.1  >  10.0.0.5: ICMP  echo request,id
29105, seq 2, length 64
23:38:38.292113  IP 10.0.0.1  >  10.0.0.5: ICMP  echo request,id
29105, seq 3, length 64
23:38:39.300131  IP  10.0.0.1 >  10.0.0.5: ICMP  echo request,id
29105, seq 4, length 64
23:38:40.595273 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01 (oui
Ethernet), length 28
23:38:41.595261 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01 (oui
Ethernet), length 28
23:38:42.568291 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01 (oui
Ethernet), length 28
```

Figure 8.10: Network Traffic from Another Tenant

Providing isolation between virtual networks is a key security requirement for any network hypervisor. Our example shows that this is not provided in OVX, and an attacker can observe traffic from another network, thereby potentially revealing sensitive information.

8.7.3 Ping of Death

As FlowVisor, OVX is vulnerable to a 'Ping of Death' attack, in which a single malformed LLDP packet results in a fatal system error, i.e. a system crash that completely stopped OVX, and hence it brings the entire network down. As before, we found this vulnerability using fuzz testing.

OVX expects all LLDP packets include a *Switch ID* field in the *OpenNetw2* TLV with the ID of a valid switch. If the *Switch ID* field is set to a value other than one of the existing switches in the network, it will force OVX to restart, which causes all network configuration to be lost, as shown in Figure 8.11. The consequences of this are essentially the same as in a system crash.

In essence, these software bugs are coding errors, such as failure to properly parse received packets, which should be caught by thorough code review and testing. The result of these vulnerabilities can be extremely severe. OVX, for example, is part of the ONOS controller

```
22:35:25.787 [pool-5-thread-145] ERROR SwitchChannelHandler - Error while
processing message from switch DPID : 1, remoteAddr : /127.0.0.1:35596
state ACTIVE
java.lang.NullPointerException
22:35:25.811 [pool-5-thread-145] INFO  PhysicalNetwork - removing port 1
22:35:25.812 [pool-5-thread-145] INFO  PhysicalNetwork - removing port 2
22:35:25.812 [pool-5-thread-145] INFO  PhysicalNetwork - removing port 3
22:35:25.813 [pool-5-thread-145] INFO  PhysicalNetwork - removing port 4
22:35:25.814 [pool-5-thread-145] INFO  PhysicalNetwork - Removing
physical link between 00:00:00:00:00:00:00:01/3 and
00:00:00:00:00:00:00:02/3
22:35:25.815 [pool-5-thread-145] INFO  PhysicalSwitch - Switch
disconnected 1
22:35:25.816 [pool-5-thread-145] INFO  StatisticsManager - Stopping Stats
collection thread for 00:00:00:00:00:00:00:01
22:35:25.818 [Thread-20] INFO  OVXPort - Cleaning up flowmods for sw
00:00:00:00:00:00:00:01 port 1
22:35:25.820 [Thread-19] ERROR DBManager - Failed to remove from db:
Write operation to server /127.0.0.1:27017 failed on database OVX
22:35:25.825 [Thread-19] INFO  OVXPort - Cleaning up flowmods for sw
00:00:00:00:00:00:00:01 port 2
22:35:25.836 [Thread-19] INFO  OVXPort - Cleaning up flowmods for sw
00:00:00:00:00:00:00:01 port 3
22:35:26.421 [pool-5-thread-128] INFO  PhysicalSwitch - Switch connected
with dpid 1, name 00:00:00:00:00:00:00:01 and type Open vSwitch
22:35:26.426 [pool-5-thread-129] INFO  PhysicalNetwork - Adding physical
link between 00:00:00:00:00:00:00:01/3 and 00:00:00:00:00:00:00:02/3
22:35:27.209 [pool-5-thread-131] INFO  PhysicalNetwork - Adding physical
link between 00:00:00:00:00:00:00:02/3 and 00:00:00:00:00:00:00:01/3
```

Figure 8.11: OpenVirteX Crash, Ping of Death

platform, which is a 'carrier-grade' SDN controller and is widely used in large-scale networks. Being able to disable such a network via the sending of a single message is definitively a problem. This work hopefully provides the motivation for a more thorough code analysis and testing of key SDN infrastructure components such as network hypervisors.

8.8 Conclusions

Network virtualisation is an essential service in SDN, and it provides a number of key benefits. Given their critical position in the SDN architecture, representing a single point of failure, the security analysis of SDN hypervisors is critical for the security and reliability of SDN in general. In this chapter, we provided the results of our security analysis of FlowVisor and OVX, the two most widely used SDN hypervisor platforms.

Our analysis found a number of new vulnerabilities in both FlowVisor and OVX, which allow an attacker to significantly disrupt or disable networks, as demonstrated in our experiments.

Given the increasingly critical role SDN hypervisors in large-scale networks, our findings provide an important motivation for a more careful testing and analysis of hypervisor code, prior to deployment in production systems.

Chapter 9

Conclusion

SDN is an important new networking paradigm with very large potential impact and is gaining rapid adoption. Network security is absolutely critical, given the increased number and sophistication of cyber attacks. While there has been a lot of research exploring SDN to implement a range of security features and functionality, there has been a limited amount of work exploring the security of the SDN platform itself. SDN provides a very different approach to managing network, with its logically centralised control plane. As a result, security in SDN is potentially very different from security in traditional networks, with potentially new attack vectors.

This thesis tries to address a gap in the SDN literature, by providing a security analysis of the current SDN architecture and platforms. The aim was to thoroughly analyse the security of critical SDN building blocks, services and components, such as Topology Discovery, ARP handling, and network hypervisors. As a result, we have identified new, critical SDN security vulnerabilities and attacks. For some of the vulnerabilities, we were able to demonstrate the attacks and discuss and quantify the impact while for others, we were able to propose efficient countermeasures and mitigation strategies.

For the critical SDN service of topology discovery (OFDP), which is a core component of all current SDN controller platforms, we demonstrated the feasibility of topology poisoning attacks, where an attacker can create fake links in a controller's topology database. We proposed a mitigation approach, which provides authentication and integrity protection to LLDP packets, by adding a hash-based message authentication code (HMAC). We demonstrated

that the method is secure against replay attacks, and is computationally efficient.

We have further discussed how current ARP handling approaches in SDN are vulnerable to ARP spoofing attacks. As an initial solution, we have adopted Dynamic ARP Inspection (DAI) mechanism, used in traditional networks to detect and mitigate ARP spoofing attacks through relying on a trusted database of IP-to-MAC address mappings, to the SDN architecture. By leveraging the key features of SDN, such as centralised nature of the control plane functionality, we have also developed a novel SDN-specific method, which does not require a trusted database of IP-to-MAC address mappings. Instead, it sanitises potentially spoofed fields of ARP requests and replies. This method prevents ARP spoofing attacks from poisoning the end-host ARP caches and ARP handling approaches. It significantly achieves better network performance compared to the current state-of-the-art ARP handling components without security and imposes a minimal control CPU load.

While exploring ARP handling in SDN, we have discovered a new, significantly more efficient method of ARP handling in SDN. By offloading ARP handling functionality from the control plane to the data plane, we achieved an order of magnitude reduction in the ARP response time, as well as a significant reduction in the computational overhead at the controller. This has the important side effect of making the SDN controller more resilient against resource exhaustion attacks. Furthermore, we have investigated a range of Denial of Service (DoS) attacks against the SDN architecture, both the control plane and the data plane. We have demonstrated the attacks, and have analysed and quantified their impact on different modern SDN controller platforms. Our experiments have shown that an attacker can successfully disrupt the operation of an OpenFlow-enabled network with DoS attacks, using relatively modest resources.

Finally, the thesis presents an evaluation of the security of the network virtualisation layer in SDN, i.e. network hypervisors. We specifically considered FlowVisor and OpenVirteX, the two most important SDN hypervisors in terms of practical relevance. Using simple techniques such as code analysis and fuzz testing, we have identified a number of new, critical vulnerabilities that allow an attacker to disrupt an entire network, as well as break the isolation between virtual networks. We hope this work provides the required motivation for future work in this space. If SDN is to be widely adopted, the security of its infrastructure is absolutely critical.

While this thesis made significant steps towards securing and protecting the SDN architecture from various attack types, more investigation needs to be done. Critical areas of future work include providing a higher degree of resilience to DoS attacks, as well as securing critical SDN components such as network hypervisors.

Bibliography

[1] SDN: Transforming Networking to Accelerate Business Agility. Available from: http://www.opennetsummit.org/archives/mar14/site/why-sdn.html [20 May 2018].

[2] OpenFlow Switch Specification . Available from: https://www.opennetworking.org/software-defined-standards/specifications/ [20 May 2018].

[3] R. Hand, M. Ton, and E. Keller, "Active security," in *Proc. of the Twelfth Workshop on Hot Topics in Networks*. ACM, 2013, p. 17.

[4] A. Zaalouk, R. Khondoker, R. Marx, and K. Bayarou, "Orchsec: An orchestrator-based architecture for enhancing network-security using network monitoring and sdn control functions," in *Proc. of the Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9.

[5] K. Wang, Y. Qi, B. Yang, Y. Xue, and J. Li, "LiveSec: Towards effective security management in large-scale production networks," in *Proc. of the 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2012, pp. 451–460.

[6] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proc. of the second SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 55–60.

[7] J. Moy, "OSPF version 2," 1997.

[8] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," Tech. Rep., 2005.

[9] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[10] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, “Software defined networking: State of the art and research challenges,” *Computer Networks*, vol. 72, pp. 74–98, 2014.

[11] N. Feamster, J. Rexford, and E. Zegura, “The road to SDN,” *Queue*, vol. 11, no. 12, p. 20, 2013.

[12] N. McKeown, “Software-defined networking,” *INFOCOM keynote talk*, 2009.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, “B4: Experience with a globally-deployed software defined WAN,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[15] Cisco Open SDN Controller. Available from: https://www.cisco.com/c/en/us/products/cloud-systems-management/open-sdn-controller/index.html [20 May 2018].

[16] Huawei Agile Modular Switch Stays Ahead of Competition in SDN by Receiving OpenFlow v1.3 Certification. Available from: http://e.huawei.com/en-IN/news/global/2015/201511241041 [20 May 2018].

[17] OpenFlow Support on Juniper Networks Devices. Available from: https://www.juniper.net/documentation/enUS/release-independent/junos/topics/reference/general/junos-sdn-openflow-supported-platforms.html [20 May 2018].

[18] Run Both Worlds with Hybrid SDN. Available from: https://www.hpe.com/us/en/networking/infrastructure.html [20 May 2018].

[19] O. N. Fundation, “Software-defined networking: The new norm for networks,” *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[20] P. Congdon, “Link Layer Discovery Protocol,” RFC 2922, July, Tech. Rep., 2002.

[21] D. Plummer, “Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48. bit Ethernet address for transmission on Ethernet hardware,” 1982.

[22] C. L. Abad and R. I. Bonilla, “An analysis on the schemes for detecting and preventing ARP cache poisoning attacks,” in *Proc. of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*. IEEE, 2007, pp. 60–60.

[23] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, “Inferring internet denial-of-service activity,” *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 115–139, 2006.

[24] A. Ornaghi and M. Valleri, “Man in the middle attacks Demos,” *Blackhat [Online Document]*, vol. 19, 2003.

[25] Dynamic ARP Inspection. Available from: http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/dynarp.html [20 May 2018].

[26] Cyber Security, Terrorism, and Beyond: Addressing Evolving Threats to the Homeland. https://www.fbi.gov/news/testimony/cyber-security-terrorism-and-beyond-addressing-evolving-threats-to-the-homeland [20 May 2018].

[27] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, “Flowvisor: A network virtualization layer,” *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[28] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, W. Snow, and G. Parulkar, “OpenVirteX: A Network Hypervisor,” *Open Networking Summit*, 2014.

[29] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[30] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proc. of the 9th SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[31] Open vSwitch. Available from: http://www.openvswitch.org [20 May 2018].

[32] M. Suñé, L. Bergesio, H. Woesner, T. Rothe, A. Köpsel, D. Colle, B. Puype, D. Simeonidou, R. Nejabati, M. Channegowda *et al.*, “Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed,” *Computer Networks*, vol. 61, pp. 132–150, 2014.

[33] S. Kaur, J. Singh, and N. S. Ghumman, “Network programmability using POX controller,” in *Proc. of the International Conference on Communication, Computing & Systems (ICCCS)*, no. s 134. IEEE, 2014, p. 138.

[34] Ryu SDN Controller. Available from: https://osrg.github.io/ryu [20 May 2018].

[35] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow *et al.*, “ONOS: towards an open, distributed SDN OS,” in *Proc. of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[36] “Floodlight SDN Controller,” Available from http://www.projectfloodlight.org/floodlight [20 May 2018].

[37] J. Medved, R. Varga, A. Tkacik, and K. Gray, “Opendaylight: Towards a model-driven sdn controller architecture,” in *Proc. of the 15th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2014, pp. 1–6.

[38] Scapy Library. Available from: http://www.secdev.org/projects/scapy/doc/usage.html [20 May 2018].

[39] B. Hatch, J. Lee, and G. Kurtz, *Hacking Linux exposed: Linux security secrets & solutions*. Osborne/McGraw-Hill New York, 2001.

[40] PACKETH. Available from: http://packeth.sourceforge.net/packeth/Home.html [20 May 2018].

[41] A. Turner, M. Bing, and F. Klassen. Tcpreplay - Pcap editing and replaying utilities. Available from: http://tcpreplay.appneta.com [20 May 2018].

[42] Stress-ng Tool. Available from: http://manpages.ubuntu.com/manpages/wily/man1/stress-ng.1.html [20 May 2018].

[43] J. Kanclirz Jr, *Netcat power tools*. Syngress, 2008.

[44] “VirtualBox,” Available from: https://www.virtualbox.org [20 May 2018].

[45] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proc. of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[46] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[47] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[48] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[49] S. Shenker, M. Casado, T. Koponen, N. McKeown *et al.*, "The future of networking, and the past of protocols," *Open Networking Summit*, vol. 20, pp. 1–30, 2011.

[50] Open Networking Foundation. Available from: https://www.opennetworking.org [20 May 2018].

[51] C. Douligeris and D. N. Serpanos, *Network security: current status and future directions*. John Wiley & Sons, 2007.

[52] W. Stallings, *Cryptography and network security: principles and practices*. Pearson Education India, 2006.

[53] R. W. Shirey, "Internet security glossary, version 2," 2007.

[54] W. Stallings, *Network security essentials: applications and standards*. Pearson Education India, 2007.

[55] V. L. Voydock and S. T. Kent, "Security mechanisms in high-level network protocols," *ACM Computing Surveys (CSUR)*, vol. 15, no. 2, pp. 135–171, 1983.

[56] D. Senie and P. Ferguson, "Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing," *Network*, 1998.

[57] M. Kaeo, *Designing network security*. Cisco Press, 2003.

[58] E. Rescorla, *SSL and TLS: designing and building secure systems*. Addison-Wesley Reading, 2001, vol. 1.

[59] N. Doraswamy and D. Harkins, *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

[60] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, “Implementing a distributed firewall,” in *Proc. of the 7th conference on Computer and communications security*. ACM, 2000, pp. 190–199.

[61] C. H. Rowland, “Intrusion detection system,” Jun. 11 2002, uS Patent 6,405,318.

[62] D. E. Denning, “An intrusion-detection model,” *IEEE Transactions on software engineering*, no. 2, pp. 222–232, 1987.

[63] M. Roesch *et al.*, “Snort: Lightweight intrusion detection for networks.” in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[64] G. M. Jackson, “Intrusion prevention system,” Nov. 25 2008, uS Patent 7,458,094.

[65] L. Schehlmann, S. Abt, and H. Baier, “Blessing or curse? Revisiting security aspects of Software-Defined Networking,” in *Proc. of the 10th International Conference on Network and Service Management (CNSM)*. IEEE, 2014, pp. 382–387.

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

[67] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, “SIMPLE-fying middlebox policy enforcement using SDN,” *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 27–38, 2013.

[68] S. Shin and G. Gu, “Attacking software-defined networks: A first feasibility study,” in *Proc. of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 165–166.

[69] A. Akhunzada, E. Ahmed, A. Gani, M. K. Khan, M. Imran, and S. Guizani, “Securing software defined networks: taxonomy, requirements, and open issues,” *IEEE Communications Magazine*, vol. 53, no. 4, pp. 36–44, 2015.

[70] Q. Yan, F. R. Yu, Q. Gong, and J. Li, “Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 602–622, 2016.

[71] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A Protection Architecture for Enterprise Networks," in *Proc. of USENIX Security Symposium*, vol. 49, 2006, p. 50.

[72] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proc. of SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 1–12.

[73] M. Cheminod, L. Durante, L. Seno, F. Valenza, A. Valenzano, and C. Zunino, "Leveraging SDN to improve security in industrial networks," in *Proc. of the 13th International Workshop on Factory Communication Systems (WFCS)*. IEEE, 2017, pp. 1–7.

[74] F. Hao, T. Lakshman, S. Mukherjee, and H. Song, "Secure Cloud Computing with a Virtualized Network Infrastructure," in *Proc. of HotCloud*, 2010.

[75] M. Wang, B. Li, and Z. Li, "sFlow: Towards resource-efficient and agile service federation in service overlay networks," in *Proc. of 24th International Conference on Distributed Computing Systems*. IEEE, 2004, pp. 628–635.

[76] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[77] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[78] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proc. of OSDI*, vol. 10, 2010, pp. 1–6.

[79] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *Proc. of the internet network management conference on Research on enterprise networking*, 2010, pp. 3–3.

[80] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *Proc. of Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–4.

[81] M. P. Fernandez, “Comparing openflow controller paradigms scalability: Reactive and proactive,” in *Proc. of the 27th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2013, pp. 1009–1016.

[82] S. Scott-Hayward, S. Natarajan, and S. Sezer, “A survey of security in software defined networks,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 623–654, 2016.

[83] S. Scott-Hayward, G. O’Callaghan, and S. Sezer, “SDN security: A survey,” in *Proc. of SDN For Future Networks and Services (SDN4FNS)*. IEEE, 2013, pp. 1–7.

[84] R. Braga, E. Mota, and A. Passito, “Lightweight DDoS flooding attack detection using NOX/OpenFlow,” in *Proc. of the 35th Conference on Local Computer Networks (LCN)*. IEEE, 2010, pp. 408–415.

[85] T. Kohonen, “The self-organizing map,” *Neurocomputing*, vol. 21, no. 1, pp. 1–6, 1998.

[86] C. YuHunag, T. MinChi, C. YaoTing, C. YuChieh, and C. YanRen, “A novel design for future on-demand service and security,” in *Proc. of the 12th International Conference on Communication Technology (ICCT)*. IEEE, 2010, pp. 385–388.

[87] D. Farinacci, D. Lewis, D. Meyer, and V. Fuller, “The locator/ID separation protocol (LISP),” 2013.

[88] Y. Choi, “Implementation of content-oriented networking architecture (CONA): a focus on DDoS countermeasure,” in *Proc of 1st European NetFPGA Developers Workshop*, 2010.

[89] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, “A replication component for resilient OpenFlow-based networking,” in *Proc. of Network Operations and Management Symposium (NOMS)*. IEEE, 2012, pp. 933–939.

[90] K. Cabaj, J. Wytrebowicz, S. Kuklinski, P. Radziszewski, and K. T. Dinh, “SDN Architecture Impact on Network Security,” in *FedCSIS Position Papers*, 2014, pp. 143–148.

[91] S. Dotcenko, A. Vladyko, and I. Letenko, “A fuzzy logic-based information security management for software-defined networks,” in *Proc. of the 16th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2014, pp. 167–171.

[92] S. Schechter, J. Jung, and A. Berger, "Fast detection of scanning worm infections," in *Proc. of Recent Advances in Intrusion Detection*. Springer, 2004, pp. 59–81.

[93] M. M. Williamson, "Throttling viruses: Restricting propagation to defeat malicious mobile code," in *Proc. of the 18th Annual Computer Security Applications Conference*. IEEE, 2002, pp. 61–68.

[94] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui, "Towards an access control scheme for accessing flows in SDN," in *Proc. of 1st Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–6.

[95] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[96] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

[97] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[98] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proc. of the SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 413–424.

[99] I. Alsmadi and D. Xu, "Security of software defined networks: A survey," *computers & security*, vol. 53, pp. 79–108, 2015.

[100] S. Shin and G. Gu, "CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *Proc. of 20th International Conference on Network Protocols (ICNP)*. IEEE, 2012, pp. 1–6.

[101] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: building robust firewalls for software-defined networks," in *Proc. of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 97–102.

[102] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proc. of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[103] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *Proc. of NDSS*, 2013.

[104] K. Benton, L. J. Camp, and C. Small, "Openflow vulnerability assessment," in *Proc. of the second workshop on Hot topics in software defined networking*. ACM, 2013, pp. 151–152.

[105] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *Proc. of the International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2015, pp. 77–81.

[106] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic controller provisioning in software defined networks," in *Proc. of 9th International Conference on Network and Service Management (CNSM)*. IEEE, 2013, pp. 18–25.

[107] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proc. of the SIGSAC conference on computer and communications security*. ACM, 2014, pp. 78–89.

[108] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proc. of the 3rd workshop on Assurable and usable security configuration*. ACM, 2010, pp. 37–44.

[109] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takács, and P. Sköldström, "Scalable fault management for OpenFlow," in *Proc. of the international conference on Communications (ICC)*. IEEE, 2012, pp. 6606–6610.

[110] D. Kotani and Y. Okabe, "A packet-in message filtering mechanism for protection of control plane in openflow networks," in *Proc. of the 10th symposium on Architectures for networking and communications systems*. ACM, 2014, pp. 29–40.

[111] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, “Resonance: dynamic access control for enterprise networks,” in *Proc. of the 1st workshop on Research on enterprise networking*. ACM, 2009, pp. 11–18.

[112] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang, “Towards a secure controller platform for openflow applications,” in *Proc. of the second SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 171–172.

[113] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, “Veriflow: Verifying network-wide invariants in real time,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[114] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “Model checking invariant security properties in OpenFlow,” in *Proc. of International Conference on Communications (ICC)*. IEEE, 2013, pp. 1974–1979.

[115] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, “An assertion language for debugging SDN applications,” in *Proc. of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 91–96.

[116] M. Canini, D. Kostic, J. Rexford, and D. Venzano, “Automating the testing of OpenFlow applications,” in *Proc. of the 1st International Workshop on Rigorous Protocol Engineering (WRiPE)*, no. EPFL-CONF-167777, 2011.

[117] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, “Efficient topology discovery in software defined networks,” in *Proc. of the 8th International Conference on Signal Processing and Communication Systems (ICSPCS)*. IEEE, 2014, pp. 1–8.

[118] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, “GENI: A federated testbed for innovative network experiments,” *Computer Networks*, vol. 61, pp. 5–23, 2014.

[119] O. N. Fundation, “Software-defined networking: The new norm for networks,” *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[120] “IEEE Standard for Local and Metropolitan Area Networks– Station and Media Access Control Connectivity Discovery,” *IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005)*, pp. 1–204, Sept 2009.

[121] D. Erickson, "The beacon openflow controller," in *Proc. of the second SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 13–18.

[122] S. Skiena, "Dijkstraś algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.

[123] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[124] Using Bellman-Ford to Find a Shortest Path. Available from: http://csie.nqu.edu.tw/smallko/sdn/sdn.htm [20 May 2018].

[125] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *Proc. of the Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[126] H. Krawczyk, R. Canetti, and M. Bellare, "HMAC: Keyed-hashing for message authentication," *IETF RFC 2104*, 1997.

[127] S. Turner and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms," *IETF RFC 6151*, 2011.

[128] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. of the Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[129] S. E. Deering, "Internet protocol, version 6 (IPv6) specification," 1998.

[130] T. Narten, W. A. Simpson, E. Nordmark, and H. Soliman, "Neighbor discovery for IP version 6 (IPv6)," 2007.

[131] F. A. Barbhuiya, S. Biswas, and S. Nandi, "Detection of neighbor solicitation and advertisement spoofing in IPv6 neighbor discovery protocol," in *Proc. of the 4th international conference on Security of information and networks*. ACM, 2011, pp. 111–118.

[132] K. R. Fall and W. R. Stevens, *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[133] K. Kwon, S. Ahn, and J. W. Chung, “Network security management using ARP spoofing,” in *Proc. of the International Conference on Computational Science and Its Applications*. Springer, 2004, pp. 142–149.

[134] D. Bruschi, A. Ornaghi, and E. Rosti, “S-ARP: a secure address resolution protocol,” in *Proc. of the 19th Annual Computer Security Applications Conference*. IEEE, 2003, pp. 66–74.

[135] J. Arkko, J. Kempf, B. Zill, and P. Nikander, “Secure neighbor discovery (SEND),” Tech. Rep., 2005.

[136] P. Nikander, J. Kempf, and E. Nordmark, “IPv6 neighbor discovery (ND) trust models and threats,” Tech. Rep., 2004.

[137] M. G. Gouda and C.-T. Huang, “A secure address resolution protocol,” *Computer Networks*, vol. 41, no. 1, pp. 57–71, 2003.

[138] W. Lootah, W. Enck, and P. McDaniel, “Tarp: Ticket-based address resolution protocol,” *Computer Networks*, vol. 51, no. 15, pp. 4322–4337, 2007.

[139] B. Issac and L. A. Mohammed, “Secure unicast address resolution protocol (S-UARP) by extending DHCP,” in *Proc. of the 13th International Conference on Networks, Jointly held with the 7th Malaysia International Conference on Communication*, vol. 1. IEEE, 2005, pp. 6–pp.

[140] V. Ramachandran and S. Nandi, “Detecting ARP spoofing: An active technique,” in *Proc. of the International Conference on Information Systems Security*. Springer, 2005, pp. 239–250.

[141] V. Goyal and R. Tripathy, “An efficient solution to the ARP cache poisoning problem,” in *Proc. of Australasian Conference on Information Security and Privacy*. Springer, 2005, pp. 40–51.

[142] S. Y. Nam, D. Kim, J. Kim *et al.*, “Enhanced ARP: preventing ARP poisoning-based man-in-the-middle attacks,” *IEEE communications letters*, vol. 14, no. 2, pp. 187–189, 2010.

[143] A. P. Ortega, X. E. Marcos, L. D. Chiang, and C. L. Abad, “Preventing ARP cache poisoning attacks: A proof of concept using OpenWrt,” in *Proc. of the Network Operations and Management Symposium, Latin American*. IEEE, 2009, pp. 1–9.

[144] P. Saunderson and J. P. Smith, "Network including snooping," Sep. 29 2009, US Patent 7,596,614.

[145] Y. Taniguchi, H. Tsutsumi, N. Iguchi, and K. Watanabe, "Design and Evaluation of a Proxy-Based Monitoring System for OpenFlow Networks," *The Scientific World Journal*, 2016.

[146] A. M. AbdelSalam, A. B. El-Sisi, and V. Reddy. Mitigating ARP Spoofing Attacks in Software-Defined Networks. Available from: http://www.researchgate.net/publication/299369116_Mitigating_ARP_Spoofing_Attacks_in_Software-Defined_Networks [20 May 2018].

[147] M. Z. Masoud, Y. Jaradat, and I. Jannoud, "On preventing ARP poisoning attack utilizing Software Defined Network (SDN) paradigm," in *Proc. of Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*. IEEE, 2015, pp. 1–5.

[148] A. Nehra, M. Tripathi, and M. Gaur, "FICUR: Employing SDN programmability to secure ARP," in *Proc. of the 7th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2017, pp. 1–8.

[149] H. Cho, S. Kang, and Y. Lee, "Centralized ARP proxy server over SDN controller to cut down ARP broadcast in large-scale data center networks," in *Proc. of the International Conference on Information Networking (ICOIN)*. IEEE, 2015, pp. 301–306.

[150] A. Coxhead, "Improving the Security and Efficiency of Network Clients Using OpenFlow, BCMS(Hons) thesis, The University of Waikato, Hamilton, New Zealand, 2013," 2013.

[151] M. Matties, "Distributed responder ARP: Using SDN to re-engineer ARP from within the network," in *Proc. of the International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2017, pp. 678–683.

[152] R. OLIVEIRA, A. A. Shinoda, C. M. Schweitzer, R. L. Iope, and L. R. Prete, "L3-ARPSec–A Secure Openflow Network Controller Module to control and protect the Address Resolution Protocol," *XXXIII Simpósio Brasileiro De Telecomunicações*, pp. 158–162, 2015.

[153] K. Elmeleegy and A. L. Cox, "Etherproxy: Scaling Ethernet by Suppressing Broadcast Traffic," in *Proc. of INFOCOM*, 2010, pp. 1584–1592.

[154] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the Datacenter," in *Proc. HotNets*, 2009.

[155] B. Issac, "Secure ARP and secure DHCP protocols to mitigate security attacks," *arXiv preprint arXiv:1410.4398, 2014*, 2014.

[156] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an Elastic Distributed SDN Controller," in *Proc. of SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 7–12.

[157] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a Framework for Efficient and Scalable Offloading of Control Applications," in *Proc. of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24.

[158] J. Yang, Z. Zhou, T. Benson, X. Yang, X. Wu, and C. Hu, "FOCUS: Function Offloading from a Controller to Utilize Switch Power," Technical Report CS-TR-2016.001, Duke University, February 2016.

[159] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired Switches," ACM SIGCOMM SOSR, 2016.

[160] R. Mohammadi, R. Javidan, and M. Conti, "SLICOTS: An SDN-Based Lightweight Countermeasure for TCP SYN Flooding Attacks," *IEEE Transactions on Network and Service Management*, 2017.

[161] Q. Yan and F. R. Yu, "Distributed denial of service attacks in software-defined networking with cloud computing," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 52–59, 2015.

[162] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff, "Detecting Heavy Flows in the SDN Match and Action Model," *arXiv preprint arXiv:1702.08037*, 2017.

[163] S. Gao, Z. Peng, B. Xiao, A. Hu, and K. Ren, "FloodDefender: protecting data and control plane resources under SDN-aimed DoS attacks," in *Proc. of International Conference on Computer Communications (INFOCOM)*. IEEE, 2017.

[164] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, “A Survey on the Security of Stateful SDN Data Planes,” *IEEE Communications Surveys & Tutorials*, 2017.

[165] R. Kandoi and M. Antikainen, “Denial-of-service attacks in OpenFlow SDN networks,” in *Proc. of the IFIP International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 1322–1326.

[166] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, and V. Nigam, “Slow TCAM Exhaustion DDoS Attack,” in *Proc. of the IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 17–31.

[167] H. Wang, L. Xu, and G. Gu, “Floodguard: A dos attack prevention extension in software-defined networks,” in *Proc. of the 45th Annual IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015, pp. 239–250.

[168] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, “Lineswitch: tackling control plane saturation attacks in software-defined networking,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1206–1219, 2017.

[169] T. Anderson, L. Peterson, S. Shenker, and J. Turner, “Overcoming the Internet Impasse Through Virtualization,” *Computer*, vol. 38, no. 4, pp. 34–41, 2005.

[170] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, “Virtual network embedding: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[171] Q. Hu, Y. Wang, and X. Cao, “Resolve the virtual network embedding problem: A column generation approach,” in *Proc. of INFOCOM*. IEEE, 2013, pp. 410–414.

[172] J. Sahoo, S. Mohapatra, and R. Lath, “Virtualization: A survey on concepts, taxonomy and associated security issues,” in *Proc. of Second International Conference on Computer and Network Technology (ICCNT)*. IEEE, 2010, pp. 222–226.

[173] P. H. V. Guimaraes, L. H. G. Ferraz, J. V. Torres, D. M. Mattos, I. D. Alvarenga, C. S. Rodrigues, O. C. M. Duarte *et al.*, “Experimenting content-centric networks in the future internet testbed environment,” in *Proc. of the International Conference on Communications Workshops (ICC)*. IEEE, 2013, pp. 1383–1387.

[174] R. D. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "Vertigo: Network virtualization and beyond," in *Proc. of the European Workshop on Software Defined Networking*. IEEE, 2012, pp. 24–29.

[175] E. Salvadori, R. D. Corin, A. Broglio, and M. Gerola, "Generalizing virtual network topologies in OpenFlow-based networks," in *Proc. of Global Telecommunications Conference (GLOBECOM)*. IEEE, 2011, pp. 1–6.

[176] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, 2013.

[177] Z. Bozakov and P. Papadimitriou, "Autoslice: automated and scalable slicing for software-defined networks," in *Proc. of the conference on CoNEXT student workshop*. ACM, 2012, pp. 3–4.

[178] H. Yamanaka, E. Kawai, and S. Shimojo, "AutoVFlow: Virtualization of large-scale wide-area OpenFlow networks," *Computer Communications*, vol. 102, pp. 28–46, 2017.

[179] M. Caesar and J. Rexford, "Building bug-tolerant routers with virtualization," in *Proc. of the workshop on Programmable routers for extensible services of tomorrow*. ACM, 2008, pp. 51–56.

[180] B. Grubb. Heartbleed Disclosure Timeline. Available from http://www.smh.com.au/it-pro/security-it/heartbleed-disclosure-timeline-who-knew-what-and-when-20140414-zqurk [20 May 2018].

[181] J. Wack, M. Tracy, and M. Souppaya, "Guideline on network security testing," *Nist special publication*, vol. 800, no. 42, pp. 13–14, 2003.

[182] V. T. Costa and L. H. M. Costa, "Vulnerabilities and solutions for isolation in FlowVisor-based virtual network environments," *Journal of Internet Services and Applications*, vol. 6, no. 1, p. 18, 2015.

[183] Y. Qian, W. You, and K. Qian, "FlowVisor vulnerability analysis," in *Proc. of IFIP Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 867–868.