Intrusion Detection and Prevention and Proxy Hybrid using Software-Defined Networking: ARP Poisoning Defence

Name:	Merlin Roe
Student Number:	206059086
Supervisor:	Vasileios Vasilakis
Number of Pages:	61
	Including Abstract

A coursework submitted in fulfillment of the requirements
for the degree of MSc Cyber Security

Dept. of Computer Science

September 2020

Abstract

Software-Defined Networks (*SDN*) are centrally and programmatically controllable networks, by decoupling the control plane from the data plane on vertically designed routing devices. Providing network-wide configuration and monitoring by the central controllers and their associated network applications.

Intrusion Detection and Prevention Systems (*IDPS*) are devices or software that are specifically designed to monitor networks for malicious activity, taking preventative measures to prevent these actions and prevent further activity from potential threats.

This report investigates and develops an innovative IDPS tailored to function alongside the SDN Ryu controller specifically to detect and prevent a variety of ARP Poisoning attacks. By inspecting ARP packets within the network, inserting specifically crafted Link Layer flow rules on an Open vSwitch (*OvS*) to actively forward or drop ARP requests and replies. The developed IDPS utilises the network's DHCP server to maintain an autonomous host list, supporting static hosts via ARP packets. The developed IDPS is a functioning hybrid between IDPS and ARP proxy, by performing ARP replies for malicious devices that are blocked by the IDPS; ensuring traffic in false-positive scenarios.

Testing is carried out on a physical testbed, using an off-the-shelf TP-Link router with an OvS and an OpenFlow using several Raspberry Pi's as hosts, resulting in real environment results. Three separate experiments were conducted on the IDPS using the physical testbed. The first experiment tested the detection and mitigation with 400 total packets sent with a variety of ARP poisoning attacks, with prevention rate being *100%* and detected on average within *58.1ms*. The developed IDPS had minimal effect of regular ARP packets, only increasing by *1.2ms* on average with minimal impact on the Controller resources.

Future iterations of the IDPS expanding into IPv6 protocol by supporting the Neighbor Discovery Protocol (*NDP*), supporting a growing range of IPv4 and IPv6 aware networks. Further utilising the DHCP offers by directly incorporating DHCP lease times into inserted flow rules *hard_timeout* flag, ensuring flow rules exist for as long as IP and MAC pairings are valid.

Contents

List of Figures

List of Tables

List of Code Snippets

Abbreviations

ARP	Address Resolution Protocol
CSV	Comma-separated Values
CVE	Common Vulnerabilities and Exposures
DHCP	Dynamic Host Configuration Protocol
DoS	Denial of Service
DPI	Deep Packet Inspection
IDS	Intrusion Detection System
IDPS	Intrusion Detection and Prevention System
IP	Internet Protocol
MAC	Media Address Control
MiM	Man-in-the-middle
NAT	Network Address Translation
NIC	Network Interface Card
OvS	Open vSwitch
QoS	NQuality of Service
RL	Rate Limiting
RTT	Round Trip Time
SDN	Software Defined Network
TCP	Transport Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine

Ethics Statement

I, the author, Merlin Roe, declare that this report titled, Intrusion Detection and Prevention and Proxy Hybrid using Software-Defined Networking: ARP Poisoning Defence and any work presented within it are original. I confirm that:

- This work was done in candidature for a masters degree at the University of York.
- All source materials used have been acknowledged, whether they are directly quoted or not.
- The work has been completed in accordance with the University of Yorks Academic Integrity Misconduct Policies, Guidelines andProcedures, which can be found at the following URL:
  `https://www.york.ac.uk/staff/supporting-students/academic/taught/misconduct/`
- Any assistance has been in accordance with the Universitys Guidance on Proof reading and Editing and duly acknowledged. The latter document canbe retrieved from the following URL:
  `https://www.york.ac.uk/about/departments/support-and-admin/sas/student-related-policies/learning/`
- No part of this report has previously been submitted for a degree or any other qualification at this University or any other institution.
- In the case where source code has been developed from an existing script or template, this has been duly acknowledged.

1 Introduction

The introduction chapter explains the importance of the project and offers a summary of the overall system implementation and design. This includes the context, motivation, objectives and the contributions this report provides.

1.1 Context

Due to the eruption of mobile devices, internet applications and cloud services, there have been significant developments within computer networking. Many focus on the concepts of centralised programmable aware networks, where an entire cloud service could be managed from a single central server.

Software-Defined Networks (*SDN*) is an outcome of these developments as a new network architecture, which allows greater control over the network by simplifying network management and reducing complexity [1]. The SDN paradigm introduces a network controller as a new component within networks, this controller possesses omnipotent power at managing the network. A network professional designs and configures the programmatically aware controller by writing software specifically designed to control multiple packet forwarding devices using SDN defined protocols, such as OpenFlow [2]. The SDN paradigm has been exceeding adoption in many industries, for instance, Google has deployed OpenFlow within its data centre backbone network to aid the utilisation on links carrying large loads [3]. One study concluding significant SDN growth of 21.75% per year within data centres and the within the Internet of Things (*IoT*) expansion [4] [5].

Given the surge of adoption for SDN (and OpenFlow) within industries, the importance of security within SDN has grown. Research has been conducted into existing and new issues with regards to the SDN paradigm, introducing new security solutions made possible due to SDN and OpenFlow based environments. Particularly issues related to the address resolution protocol *(ARP)* poisoning attacks, which can be tackled in new ways relying upon the SDN paradigm.

ARP poisoning or ARP spoofing is an attack defined as the ability for an adversary to corrupt a hosts IP to MAC address table for local networks. Whereby an adversary pretending to be another device within a network, such as manipulating hosts into using itself as the gateway of a network. This allows an adversary to become the static route for all network traffic or a specific host, the attack is popular due to its simplicity to implement using the unauthorised ARP protocol [6]. ARP spoofing is a serious attack which opens up the adversary to perform multiple types of attacks such as Man-in-the-middle (*MiM*), eavesdropping, MAC flooding and Denial of Service (*DoS*).

1.2 Motivation

The vulnerabilities of ARP have been known since implementation, having been logged by Common Vulnerabilities and Exposures *(CVE)* as a possible DoS issue in 1999 [7]; being scored a *10* for being one of the highest threats. ARP mitigation has continued to be a concern as the protocol still exists within modern networks. A well-known piece of malware called *ServStart* relying on ARP poisoning has been used as recently as 2017 to insert malicious adverts, a server on a local subnet was compromised and the adversaries installed ARP spoofing malware on the machine. The ARP spoofing malware poisoned the local network so outgoing traffic was forwarded through it, the malware then inserted malicious JavaScript code into every HTML page served by hosts on the network [8].

Research detecting and mitigating ARP poisoning have started utilising SDNs central paradigm to maintain a paired list of MAC addresses and IP addresses of hosts on the local network, which aid in detecting when hosts are being impersonated. The methods used for gathering and maintaining this list is vastly different, from researchers listening on and recording the ARP request and reply packets [9], using the controller's own ARP Table [10], to relying on the Dynamic Host Configuration Protocol (*DHCP*) [11] [12].

The approach of a paired list is ideal as it becomes reference when detecting spoofed ARP packets, however, the research papers are split on how to accomplish this. The ones utilising DHCP fail to determine DHCP offers are from a valid DHCP server, allowing an adversary to push themselves onto the known list by acting as a valid DHCP server. These also fail to take into account previously configured hosts, and only one supporting static IP addresses [9].

The approach many of these papers take when testing their ARP poisoning mitigation's is to virtualise the environment [10] [12], this creates unrealistic results when testing for the CPU load of the controller using the new algorithm, and affects the Round Trip Time (*RTT*) of packets due to lack of a physical network switch.

1.3 Objectives

The objectives which this report will cover and attempt to accomplish, are based upon the previously discussed motivation and context around SDN networks. These are as follows:

1. Design and implement an IDPS system to detect and mitigate against ARP poisoning attacks using the SDN paradigm.
2. Impose minimal effect on network speed and SDN controller resources.
3. Test the IDS using a hardware testbed against a range of ARP poisoning attacks.

1.4 Project Contribution

This report designs and develops an SDN based Intrusion Detection and Prevention System (*IDPS*), specifically for mitigating a variety of Address Resolution Protocol (*ARP*) poisoning attacks; introducing a new hybrid approach IDPS by incorporating an ARP proxy server to manage actively blocked MAC addresses. The IDPS utilises the popular open-source Ryu SDN Controller and vendor adopted OpenFlow 1.3 protocol for communication to a physical SDN Open vSwitch (*OvS*) switch.

Innovative new software was developed as an IDPS operating as an extension to the Ryu controller, aiding in the decision of forwarding or dropping ARP packets. The proposed system builds upon other papers such as [9]- [11] which do not actively insert flow rules within the SDN paradigm, instead of handling every ARP packet regardless of having been previously verified. Therefore this system actively inserts flow rules to not only mitigate malicious packets, inserting flow rules which forward verified packets, greatly reducing the CPU load and RTT of ARP packets. While expanding on the research in [11], [12], that utilises the DHCP protocol to verify and generate the network MAC and IP pairing, while adding support for static IP addresses. The new hybrid IDPS, ARP proxy paradigm approach prevents the possibility for an adversary to mask their MAC address as a known host, causing a false positive IDPS blocking of a victim host. The hybrid approach ensures consistent replies from valid hosts by the system using it's known host list to act as an ARP proxy, generating ARP replies. The proposed system achieves this without any enhancements or modifications to the existing ARP protocol, not requiring specialised software on individual existing or new devices, with limited strain on the SDN controller.

The proposed IDPS is tested on a criteria of eight different signatures of poisoned ARP packets, consisting of ARP requests and replies. The tests sent a total of *400* poisoned ARP packets within the SDN network, were all poisoned ARP packets amongst standard ARP were detected and dropped with the adversary MAC address being actively blocked with a flow rule. The testing highlights the average RTT for standard ARP is *45ms*, IDPS flow rule ARP is *46ms* and being verified and forwarded by the proposed IDPS at *96ms*. An insignificant affect on the ARP RTT within the network, demonstrating the small interference the IDPS poses on the traffic.

2 Background

The background chapter contains the necessary information for the reader regarding potential unfamiliar concepts, tools and protocols. Initially discussing the differences between conventional networks and the programmable SDN paradigms, followed by detailed explanations of software and tools used to manage and control an SDN; such as OpenFlow and OvSwitch. This section also contains information about the specific tools and protocols used such as, the Ryu SDN Controller, ARP protocol and IDPS.

2.1 Conventional Networks

Conventional computer networks are built up of many individual devices such as routers and switches, with the primary purpose of sending, forwarding and receiving packets amongst themselves. These devices perform their purpose by operating two functional planes: Control plane and Data plane. The Control plane is concerned with drawing the network topology, routing table, storing a list of destination addresses and configurations defined by a network engineer whereby it can act accordingly to define what to do with network packets. The Data plane forwards packets as defined by the Control Plane to another device [13].

These devices function independently within large networks, with devices forwarding traffic based on their individual configuration, and implementations. Therefore, devices decide how to forward each individual packet according to their own routing tables, visible surrounding networks and configurations. Ensuring the network runs predictably and reliably depends upon network engineers, who correctly and consistently configure these devices with a clear understanding of the network and its complexity, as well as, the device is functioning within [14].

2.1.1 Problems with Conventional Networks

As these devices are vertically integrated these planes are being controlled independently within each device, leading to traditional computer networks becoming increasingly complex and hard to manage systems.

They involve different types of devices such as routers, switches, network address translators, server load balancers, and intrusion detection systems. Routers and switches usually run complex, distributed controlled software that is proprietary.

Network administrators are typically forced to configure individual network devices using configuration interfaces that vary between vendors and different specific low level configuration languages, even between different products from the same vendor. Although some network management tools offer a central configuration for the network, these systems still operate with vendor-specific protocols, mechanisms, and configuration interfaces.

While standard networking IP protocols emerge and develop, such as Routing Information Protocol (*RIP*) and Open Shortest Patch First (*OSPF*) [15], vendors often develop their own unique set of features and protocols designed to run on their systems. Such as the Cisco Discovery Protocol (*CDP*) [16], used to share information between Cisco specific devices. Vendors using different mechanisms cannot communicate with one another, resulting in extremely poor interoperability between different devices.

Data centres are concerned that networks remaining reliable and agile. Meaning they often focus on using devices from specific vendors which use proprietary features and protocols, resulting in no solutions to the fundamental issue. Vendors focus on extending the ability of their IP protocols on their existing infrastructure, increasing the complexity with individually configured devices making it difficult for new network architectures to emerge.

These fundamental issues of operation have increased complexity and inflated both the capital and the operational costs of running a network, while slowing innovation of network infrastructure designs such as Software Defined Networks [1] [17].

2.2 Software-Defined Networks

2.2.1 Programmable Networks

The issues outlined within traditional networks are previously defined hurdles, aggravated over time, with new technology and a shift to a more connected and cloud-based internet. The desire to move away from the difficulty that vertically integrated devices bring, into programmable networks has been in slow development for more than 20 years. Campbell et. al. researched the existing concepts for programmable networks in 1999 as seen in *'A survey of programmable networks'* [18].

The goal of programmable networks is to simplify the complexity of networking systems and allowing customers to easily configure and introduce new services and protocols programmatically with their devices without being tied to the difficulties of specific vendors and their systems.

The concept of separating the Data plane and Control plane has been discussed in countless papers, the Internet Engineering Task Force (IETF) began considering various ways to decouple the planes and proposed the interface standard published in 2004 by Yang et. al. in *"Forwarding and Control Element Separation (ForCES)"*. [19]

2.2.2 SDN Operation

The development of programmable networks resulted in the current form of SDN; an advanced and programmable approach to computer network architectures. Meaning the aforementioned concepts to decouple the Control plane and Data plane. The separation of these planes creates a new hierarchal layout using centralised management and configuration over an entire network, the separation morphs network switches into simple forwarding

devices and the Control plane is implemented within a central controller, although in principle it is physically distributed. The central controller is the entity that dictates the flow of traffic in the network. The logical centralisation of the control logic is within a programmable software module that runs in a standard server within the network.

SDN offers a number of benefits, firstly, is it simpler and less error-prone when modifying the network rules and policies, rather than via low-level individual device configurations. Second, the central control program can automatically react to unforeseen changes and continue to forward traffic within the network while maintaining the high-level rules. Third, the centralisation of the Control plane via a controller with global knowledge of the topology and network state simplifies the development of newer and more advanced network functionality. This ability to programmatically configure the network to Control the Data plane is the crucial feature of the SDN paradigm. Providing new methods to solve age-old problems in networking, while simultaneously enabling the use of sophisticated network rules and flows, such as security and dependability [20].

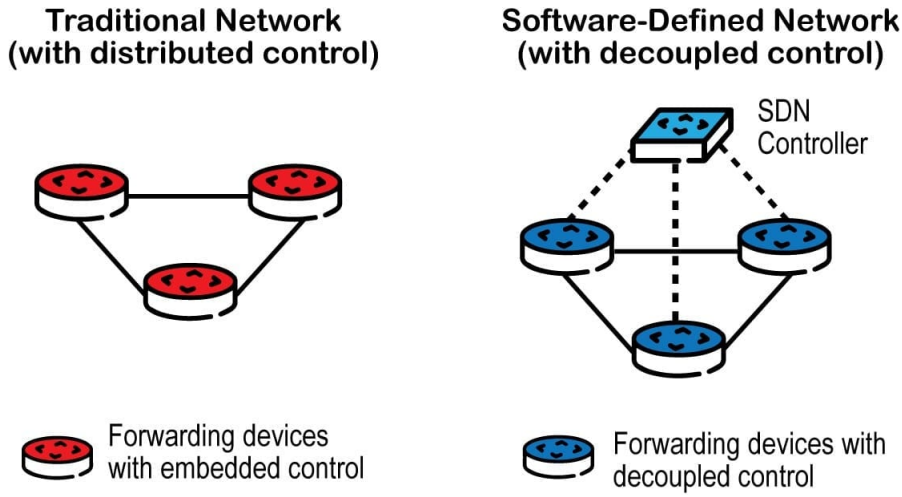

Figure 2.1: Traditional Networks Compared to SDN [21].

Figure 2.1 is the comparison of traditional networks and SDN, which visually represents the switches communication with the central SDN controller. Whilst traditional networks devices direct routes for individual packets, SDN devices use a new approach called flow tables. Each routing device on an SDN has internal flow tables, as they receive unknown packets, it communicates with the SDN controller to determine how the routing of said packet should be handled. The controller uses Deep Packet Inspection *(DPI)* to inspect the traffic in detail such as source address, destination address, protocols, etc. The controller then creates flow rules based on the packet

inspection, which the routing device will then install and use for all subsequent packets that match the flow rule. Flow rule's offer the ability for routing devices to be instructed by a central server; how to route traffic without being relied on for all packets. A flow rule is generally built up of the matching fields within a packet, such as a packet's protocol, source and destination, and the action the switch must make on the packet; (i,e forward or drop). The controller will build up the flow tables within all the switches and routers to create a unified network across routers, switches and firewalls, allowing network-wide configuration and flexibility [22].

2.2.3 Structure

An SDN structure is built up of several key components which are the decoupled planes in a traditional network, as seen in Figure 2.2 [23].

Application: This application layer is where the software designed by a network engineer resides, this layer is where the programmable configuration software runs. Communicating with the controller via the controllers own dedicated API.

Control: The layer consists simply of SDN controllers, this software is what communicates with the network infrastructure (south-side) using an API such as OpenFlow [24], to the Application layer above.

Infrastructure: This encapsulates all the devices within the network that act within the Data plane, routing traffic, such as switches, routers and firewalls dictated by the Application via the controller.

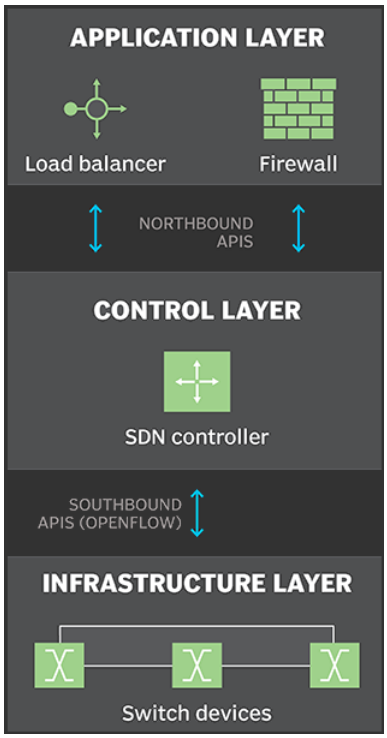

Figure 2.2: SDN architecture [25].

2.2.4 Controllers

Controllers are the primary component of an SDN, which as discussed dictates the networks behaviour. The controller is what offers engineers the central programmable configuration, which communicates with switches over an API such as OpenFlow, [24] which is discussed further in Section 2.3. There are several popular open source SDN controllers such as POX [26], Daylight [27], ONOS [28] and Ryu [29], the proposed IDPS focuses on the Ryu controller as discussed further in Section 2.6.

2.3 OpenFlow

This section covers OpenFlow and it's functionality within the SDN paradigm, this section refers to the OpenFlow 1.3 documentation unless stated otherwise [30].

2.3.1 Origin

OpenFlow is a communications protocol which originated from a system called *Ethane* proposed by Casado et. al. at Standford University in 2007, introducing simple flow-based ethernet switches with a centralised controller paradigm that manages the forwarding and routing of traffic [31]. McKeown Et. Al. expanded on this idealogy in 2008 using Standford Universitys network for testing, following Ethane, OpenFlow was designed on the same concept [24]. OpenFlow is a key role within SDN with enough momentum to start the Open Networking Foundation *(ONF)* in 2011 [32].

2.3.2 Operation

OpenFlow a controller to access network switches or routers to configure the forwarding plane and determine the flow of network packets across networks and devices; as seen in Figure 2.3. An OpenFlow switch can contain one or more flow tables, performing packet matching for forwarding or communicating via the OpenFlow channel to the central controller. The switch communicates events to the controller and the controller manages the switch via the OpenFlow protocol. The controller can then add, modify, and delete flow entries within the switches flow tables, both reactively and proactively; leaving the actual forwarding of flow table matching packets to the switch at traditional speed for the duration of those rules. Packets which do not match an existing rule within the switch are forwarded to the controller, can then dictate where to be forwarded and if a flow rule is created.

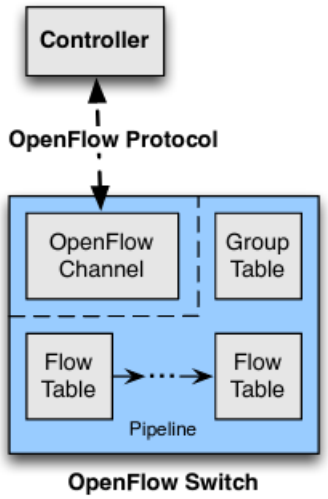

Figure 2.3: Main components of an OpenFlow switch [30].

2.3.3 Flow Tables

The following Table 2.1 outlines the main components of a flow rule within a table. Each flow table rule contains:

Match Field	Priority	Counters	Instructions	Timeouts	Cookie

Table 2.1: Main components of a flow rule in a flow table [30].

- **Match fields:** On receiving an incoming packet the OpenFlow switch performs a lookup within the flow table to match the incoming packets fields and value's to a flow rule. These may also consist of the input port and packet headers.
- **Priority:** Incoming packets may match with multiple rules, therefore, each rule has a priority which the switch use to dictate it's actions.
- **Counters:** Each flow table, flow rule, port, queue, group, group bucket, meter and meter band, contain a counter which are updated when a incoming packet matches a flow rule. These values can be polled by a counter using the OpenFlow protocol.
- **Instructions:** Each flow rule contains an instruction that is executed when a packet matches the flow rules fields. These instructions result in changes to the packet routing, action set and/or processing, such as *Apply-Actions*, *Clear-Actions* or *Write-Actions*.
- **Timeouts:** Flow rules contains a *idle_timeout* and a *hard_timeout*. The *idle_timeout* removes the flow rule once the value reaches zero, which is set by the controller and is reset when a matching packet enters. The *hard_timeout* is set by the controller and simply applies the rule until the value reaches zero.
- **Cookie:** Extra data included by the controller, can be used by the controller to filter flow statistics, flow modification and flow deletion.

An OpenFlow switch will contain a special flow rule for table misses, the flow rule ensures the switch can process incoming packets that do not match any installed flow rules. The actions available to the flow rules can be either sending the packet to the controller to dictate how to proceed, drop the packet or forward the packet on its defined route. The table miss rule does not exist by default and must be set by the controller or the OpenFlow switch itself, this rule can be modified or deleted by the controller and may contain a timeout rule.

2.3.4 OpenFlow Channel

The OpenFlow channel is the interface that allows communication between the OpenFlow switch and the controller. This channel lets the controller configure and manage the switch, receive events, and send packets out the switch. The OpenFlow channel doesn't offer any means of authentication, and therefore should be encrypted using TLS, however, this is not enforced and may run directly over TCP [30].

The protocol supports three types of messages, controller to switch, asynchronous, and symmetric. The proposed system utilises the controller to switch and asynchronous messages, the key OpenFlow messages are described below:

Controller to Switch

Initiated by the controller and used to directly manage or inspect the switch.

- **Feature:** The controller on establishment of the OpenFlow channel connection can request the identity and capabilities of the switch by a features request message. The controller uses this message to reconfigure the switch with it's expected flow rules.
- **Modify-State:** These messages are sent from the controller to the switch to either add, delete or modify flow rules.
- **Read-State:** A mechanism introduced to allow the controller to collect various pieces of information, such as the configuration, statistics and capabilities of the switch.

Asynchronous

Asynchronous messages are initiated by the switch and update the controller of network events occurring within the switch.

- **Packet-In:** When a table miss occurs, the switch can forward the incoming packet to the controller within a *Packet-In* message.
- **Flow-Removed:** Informing the controller that a flow rule has been removed from the table, either from being deleted by the controller or due to a *timeout* rule within the flow rule.
- **Error:** The switch can notify the controller of any errors it encounters.

2.3.5 Version Comparison

OpenFlow has seen several upgrades between the initial version 1.0 to version 1.5, the primary changes between versions have been on the underlying architecture and flow table changes. Version 1.1 offering the addition of multiple tables per switch, version 1.2 greatly increased the matching capabilities of the flow rules; introduction granularity flow tables. OpenFlow 1.3 expands upon the matching possibilities and introduces the ability to support IPv6. OpenFlow 1.3 has been widely adopted by vendors such as HP supporting OpenFlow 1.0 and 1.3 in the *HP 5920* switch series [33]. The performance between the popular 1.0 and 1.3 OpenFlow versions as

discussed by Šulák et. al. [34], discover that wider flow matching rules offer greater throughput, however, the narrow matching offered by OpenFlow 1.3 is greater when using quality of service techniques; with negligible differences elsewhere.

As this project is implementing an ARP IDPS service, the advantage of installing granularity flow rules catered to specific ARP requests and replies outweigh the higher throughput of ARP packets. Therefore when this paper refers to OpenFlow it's referring to version 1.3, the difference in matching rules can be seen in the tables A.1.1 and A.2 within the appendices.

2.4 Open vSwitch

The Open vSwitch (*OvS*) is open source multi-layer virtual switch, it can operate as software within another machine; such as inside a switch, router or virtual machine. OvS is designed to enable network automation forwarding functions to programmatic extensions, while still supporting standard management interfaces and protocols such as OpenFlow, NetFlow, sFlow, IPv6 and 802.1Q virtual LANS *(VLAN)*. OvS has been developed by a large open-source community that requires the ability for transactional configuration database for C and Python bindings, offering more flexibility within the development of this project. To support OvS, it comes with several beneficial tools to aid in the configuration and management of the switch, such as *ovs-ofctl* a utility for managing and configuring OpenFlow switches. Alongside *ovs-vsctl* used for managing and updating the configuration of the flow-based switch, both tools aid in the setup and control of the switch [35].

As this report is implementing within the SDN paradigm it requires an OpenFlow compatible virtual switch, therefore, will be using OvS as the software switch inside the router using OpenWRT which is discussed in the next Section 2.5.

2.5 OpenWRT

OpenWRT is an open-source highly extensible Linux operating system targeted for embedded devices, typically for wireless routers. Compared to vendor created operating systems which create a static firmware, OpenWRT is a fully writable filesystem, with a dedicated resource package manager called Open Package Management *(OPKG)*. OpenWRT removes the constraint of applications and configuration options forced by the vendor, allowing the developer and user to customise the device through the use of the package manager and Linux based applications. This offers the ability to install tools such as *Arping*, *Tcpdump* and *Iproute2*, these are common and affective networking tools which are used within this project. The *OPKG* also contains the Open vSwitch package version *2.11.3*, which supports OpenFlow 1.0 to 1.4 [36].

A router powered by the OpenWRT operating system offers all the necessary tools, software and features required to be installed as a SDN switch.

2.6 Ryu Controller

The Ryu Controller is a component-based software-defined networking framework, written in Python3 it provides components with a well-defined API that eases the development of creating new network management tools, control applications and services. Ryu supports a variety of protocols for managing network devices, such as the OpenFlow protocol from version 1.0 to 1.5. The API Ryu offers all the types of communication on the OpenFlow channel discussed in Section 2.3, creating events within the proposed Python IDPS service, including full use of the matching flow fields required for this project. As the Ryu controller is an ongoing project managed on *GitHub*, while it can be installed through Python3's *pip* command, this project opts to install Ryu through *GitHub* to obtain the latest revision [29].

2.6.1 Brief Controller Comparison

As SDN has seen a large growth primarily within the community, there have been several popular community developed open source SDN controllers. Table 2.2 below is a brief comparison featuring a handful of popular projects:

Name	OpenFlow Support	Language	Latest Release
Ryu	1.0 - 1.5	Python	12/06/2020
POX	1.0	Python	23/11/2017
NOX	1.0	C++	14/02/2014
ONOS	1.0 - 1.5	Java	02/06/2020
FloodLight	1.0 - 1.5	Java	19/06/2020
DayLight	1.0 - 1.3	Java	25/03/2020

Table 2.2: subset of Open source SDN Controller options.

Due to NOX and POX being the earliest open-source SDN controller's developed by Stanford University, they have been used in a variety of research papers [9] [10] [11]. These controllers have lacked development and have not been kept up to date with the Openflow protocol, only supporting the initial Openflow version 1.0. Therefore, they lack newer performance advancements, matching field capabilities, multiple table capability and security adjustments. Ryu has been chosen for this project due to constant updates, support of the latest OpenFlow Protocol, written in Python3 and is an open-source with community-driven documentation.

2.7 Tools

This section covers several tools that are used within the development and testing of the IDPS, this covers pre-existing tools and utilities that are commonly used for networking and ARP spoofing.

- **Iproute2:** Iproute2 abbreviated as (*IP*) which is a collection of utilities for controlling and monitoring a series of aspects of networking within Linux, such as, routing, network interfaces, tunnels and traffic control [37]. Used within this report to alter network interfaces such as enabling promiscuous mode.
- **Tcpdump:** Tcpdump is a network packet analyser that runs via the command-line interface, relying on the libpcap library it offers the visibility of TCP/ARP/IP and other packets being forwarded over a network [38]. A key utility to test the SDN network as can view the network's traffic to determine if the SDN switch is correctly forwarding and dropping packets, generating *pcap* files containing the network traffic.
- **Wireshark:** Wireshark is another network packet analyser that runs within a GUI, offering many ways to view and filter a variety of protocols within network traffic. Wireshark can also read *pcap* files created by Tcpdump [39]. This is used primarily to view and easily parse *pcap* files generated by Tcpdump.
- **Arping:** Arping is a utility for probing hosts on a network, similar to *ping* except probing by sending Link-Layer frames using ARP requests [40]. This is used when performing normal ARP requests, as well as spoofing ARP requests.
- **Python Scapy:** Scapy is a Python library that does packet manipulation [41]. The library is used within the ARP proxy implemented within the IDPS, as well as a custom testing tool used to test a variety of ARP attacks.
- **Python PSUtil:** PSUtil is a Python library that performs system monitoring by retrieving information on system utilization (CPU, memory, disks, network) [42]. Used by the proposed IDPS testing tool to gather CPU load while the IDPS is running.

2.8 Address Resolution Protocol *(ARP)*

The Address Resolution Protocol *(ARP)* is a communication protocol defined in 1982 by RFC 826, [43] used for discovering Link-Layer addresses (MAC address), and the associated internet layer address, this report focuses on an IPv4 address. This mapping is a critical function in the internet protocol suite, as it enables hosts in a local network to communicate.

The structure of ARP packets is shown in Figure 2.4 which visually demonstrates ARP encapsulated within an IPv4 network running within an Ethernet Frame. The packet header specifies the type of protocol in use (ARP) at each layer, as well as containing the operation code for ARP, request *(1)* and reply *(2)*. The primary payload of the packet contains four addresses, the hardware and protocol addresses of the source and target hosts.

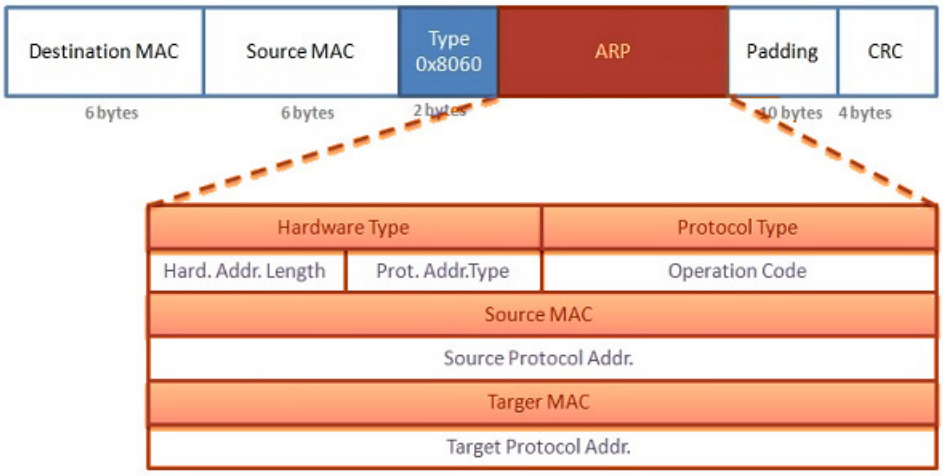

Figure 2.4: ARP packet structure.

2.8.1 ARP poisoning

Due to the early development of ARP, security was an afterthought compared to speed, therefore the basic security problem with ARP is that it's stateless, treating each request or reply independently from any previous connection. As a consequence, the ARP protocol has no method to authenticate the sender of an ARP Request or Reply packet, or any means of checking the integrity and validity.

As a result, it is relatively simple for an adversary to broadcast an ARP request packet with forged source IP-MAC in the ARP header and Ethernet frame. There are numerous ways to achieve this effect, such as the adversary using it's MAC address in the Ethernet Frame and the ARP header, while using the IP address of its target, each attack relies on the misinformation provided in the Ethernet Frame and the ARP headers. When the target receives the spoofed ARP packet, it updates it may update internal ARP cache table with the attackers forged IP and MAC pair. Therefore, when the target attempts to communicate with the original host, the packets will be

destined to the MAC of the adversary specified in the forged IP and MAC pair. ARP poisoning may be used in several attacks, such as:

- **DoS attacks:** The adversary can prevent communication between two or more hosts, also referred to as ARP flooding.
- **Host impersonation attack:** The adversary can receive packets destined for a host and impersonate and reply to these packets on their behalf. This can be achieved through the simple tool Arping, which is used for testing within this project by falsifying MAC address.
- **Man In The Middle (MITM) attack:** The adversary can monitor and save all the packets between two communicating hosts.

2.9 Intrusion Detection & Prevention System *(IDPS)*

Traditional networks were designed with security in mind, specifically common network security involves network firewalls, these devices protect networks by using a static set of rules to permit or deny network connections. It implicitly prevents intrusions, assuming an appropriate and accurate set of rules have been defined outlining the attack. Firewalls greatest limitation is its focus on limiting access between networks to prevent intrusion and do not detect or prevent an attack that originated within the network. Intrusion Detection and Prevention Systems *(IDPS)* controls the forwarding of traffic within and through a network and protect it against adversaries, often IDPS are software applications designed to monitor networks by inspecting traffic for intrusion's and taking the necessary action to prevent an attack. This form of detection allows a network to be secured from incoming and outgoing traffic, as well as, internally forwarded traffic. Furthermore, the use of DPI does not rely on a specific set of static rules like traditional firewalls, allowing the IDPS to function effectively in a changing network and threat landscape [44].

A Network Intrusion Detection and Prevention Systems (*NIDPS*) is deployed at a strategic point within the network, where it can monitor inbound and outbound traffic, to and from all the devices on the network [45]. This is the type of IDPS the proposed system is following, relying on SDN to allow monitoring and altering of traffic flow; as demonstrated in Figure 2.5.

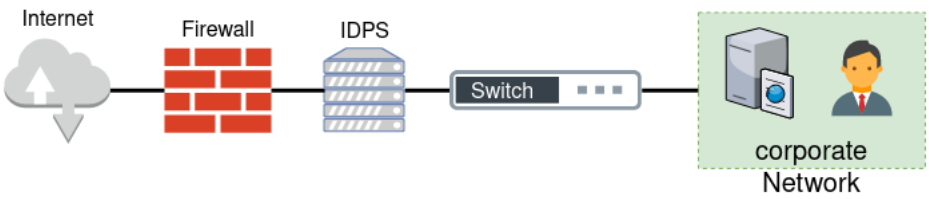

Figure 2.5: Intrusion Detection & Prevention System Diagram

The IDPS performs a real-time deep inspection on every packet that is forwarded within the network, detecting any malicious or suspicious packets, whereby it can perform one of the following actions:

- Dropping the related packets to the session detected exploiting traffic, block the offending source IP address or MAC address.
- Reprogram or reconfigure the firewall preventing an attack occurring again in the same format.
- Remove or replace any malicious content on the network during or following an attack. This is achieved by repackaging ARP payloads, removing header information deemed malicious and injecting safe information.

2.9.1 Detection Methods

IDPS typically employ the use one of three detection methods: signature based, anomaly based, and stateful policy based:

- **Signature Based:** The approach uses predefined signatures of well-known network attacks. When an attack is detected that matches a known signature, the system takes necessary prevention action.
- **Anomaly Based:** This approach monitors for any suspicious or unexpected packets on the network. If an anomaly is detected, the system blocks access from the suspicious host device.
- **Policy Based:** This approach requires network engineers to configure specific security policies according to their given network infrastructure, an event which breaks a security policy is considered an attack.

3 Literature Review

In this chapter, recent work regarding the security within SDN is researched, initially looking into SDN security and IDPS solutions that have made a big impact or are still relevant despite bring older. The focus shifts on recent and successful papers regarding the mitigation of ARP poisoning related attacks, using IDPS. A comparison of these research papers is then briefly discussed, followed by a summary review of the papers and the relationship to the project implemented in the rest of this paper.

3.1 SDN security

As SDN reaches a more stable position within its development, research shifts from SDN design into more specific areas of its function; with a large focus on security. The security research tackles a range of possible vulnerabilities, such as security exploits within the protocols a SDN may rely upon, and using SDN to resolve existing network vulnerabilities to more elaborate issues and proposals, such as a distributed SDN system for security; this section is dedicated into reviewing two papers related to SDN security.

3.1.1 OpenFlow vulnerabilities

This subsection is reviewing the paper by Samociuk, Dominik [46], which explores the lack of security within the OpenFlow protocol. Specifically, the OpenFlow Protocol handshake, which does not require the SDN controller to authenticate switches and the controller is not required to authorise switches access to the controller. This lack of security within the protocol itself leaves it vulnerable to adversaries imitating a switch to access the controller, resulting in DoS attacks that disrupt traffic. Additionally, the lack of authentication and authorization in the OpenFlow handshake can be exploited by malicious switches for covert malicious traffic, bypassing the Data plane and potentially Control plane security mechanisms. One of the proposed solutions to resolving this lack of authentication is by the addition of Transport Layer Security *(TLS)*, which ensures communication between the controller and switches are encrypted and authenticated. OpenFlow, however, does not enforce TLS since version 1.1, instead only encourages the use of TLS.

Benton et. al. [47] had researched the extent of this vulnerability within existing controllers and switches by vendors which incorporate OpenFlow before Samociuk Dominik's paper. Discovering and highlighting that the vast majority of SDN implementations have a widespread failure of security, specifically due to the lack of TLS within the control plane within the implementation of OpenFlow.

3.1.2 IDPS using SDN: Defending against port-scanning and DoS

The vast majority of security research has taken advantage of SDN's ability to DPI, the SDN controller inspects, in detail, the packets and their contents when being sent over the network. SDN relies on this functionality to generate routing flows, therefore this existing feature means SDN is ideal for implementing an IDPS by inspecting packets for malicious intent and deciding whether to drop, route or allow them. The paper by Birkinshaw et. al [48]is one such paper that takes advantage of the SDN paradigm.

The team developed an IDPS within a SDN focusing on defence against port-scanning techniques and DoS attacks. The team mitigated these threats by implementing and testing two connection-based techniques, namely the Credit-Based Threshold Random Walk *(CB-TRW)* and Rate Limiting *(RL)*. Using these two techniques the team developed a new algorithm called Port Bingo, designed to mitigate port-scanning attacks; furthermore using flow statistics alongside Quality of Service *(QoS)* to mitigate DoS attacks. The team extensively tested these algorithms within a purpose-built virtual testbed which consisted of an Open vSwitch, POX based controller and other hosts for attackers and victims.

The IDPS relies on DPI on received packets the controller obtains from the PacketIn event via the switch, to detect malicious intent within the network traffic. The Port Bingo algorithm for detection and mitigation of port scanning consisted of the expectation that an adversary would send a large number of packets to a large range of ports. Therefore, scanning the network traffic between pairs of hosts and comparing the ports to a list of prioritised port destinations, such as 80 for HTTP and 22 for SSH, in order of accessibility, would show any anomalies within the traffic. An anomaly was considered when an accumulation of SYN packets sent to a host on these bingo ports, reached an accumulated value equal too or higher than a threshold value, which had been building within a maximum period used for tracking. To avoid blacklisting a valid machine, the tracked connection flow rule is deleted after a predetermined time.

Mitigation of DoS attacks was also achieved by relying on the Open vSwitch queueing on switch ports, which can be used to apply for QoS flow entries. The algorithm developed would query the switch periodically to search for anomalies within the traffic by looking for excessive byte count or packet count in either TCP, UDP or ICMP packets. Similar to the port scanning resolution, the algorithm implements a flow rule to enqueue the offending packets on the egress switch port.

The Port Bingo TCP algorithm and QoS DoS mitigation successfully detected attacks, with correct flow entries automatically applied within offending switches which drops the malicious packets.

3.2 Related Work: SDN based ARP poisoning detection and prevention

The following papers focus on ARP detecting and preventing and shaped the development of this project, each subsection contains the paper's implementation, the testing applied and a summary of limitations.

3.2.1 Securing ARP in software defined networks

A paper by Alhrabi et. al. [9], proposes an algorithm designed to mitigate ARP poisoning in IPv4 ARP which could be adapted to work on IPv6 Neighbor Discovery Protocol *(NDP)*. The paper's focal point is *Regular ARP* over *Proxy ARP*, where ARP is handled by the hosts themselves over the SDN controller taking control and acting as a proxy server for ARP packets. Their algorithm named *'SARP_NAT'* is loosely based on Network Address Translation *(NAT)*. The IDPS relies on the SDN controller receiving all ARP packets within the network for DPI. The SARP_NAT algorithm records the ARP requests into a pending list, the algorithm NAT's the potentially malicious source MAC address and Protocol address turning the ARP request into a sanitised ARP request with dummy values which are then forwarded into the network. When the targeted machine replies to this ARP request the destination is broadcasted, where the algorithm can authenticate using the ARP handled list. Once the algorithm has confirmed the ARP reply and request as valid, it can re-perform the MAC NAT to ensure the ARP packets have reached the correct destinations.

Testing was carried out on the SARP_NAT algorithm using the POX SDN controller, with a linear topology containing 64 hosts and a tree topology with a depth of 2 and 8 fanout for a total of 64 hosts. Several measurements were tested, the CPU load caused by the algorithm over several different scenarios with an estimated reduction in CPU overhead of 22%, as well as, the Round Trip Time *(RTT)* which measured the time to handle an ARP request. The test demonstrated a significant speed advantage whereby, the ARP packets using the SARP_NAT algorithm have a 32% shorter RTT due to the *l2_learning* component for regular ARP packets.

The IDPS implemented in this paper proved extremely effective, however, the metrics used for the CPU load and RTT appear skewed. The paper uses the *l2_learning* component of POX, which contacts the controller for all-new traffic adding a flow rules and standard ARP request and replies due to the broadcasting nature of the protocol; making normal ARP behave slower. The proxy element is also slightly slower due to the constant NATing of ARP request and replies as it does not actively insert flow rules to speed up traffic. Furthermore, the *'SARP_NAT'* algorithm can fall victim to spoofed gratuitous ARP replies, receiving two ARP replies with different values in the source MAC address field, one from the genuine target host, and one from the attacker. Unfortunately, the *'SARP_NAT'* component cannot determine which one is the genuine reply and which one is poisoned.

3.2.2 Distributed responder ARP: Using SDN

As briefly mentioned in the above discussed paper, this paper utilises the concept of an *Proxy ARP*, *Distributed Responder ARP: Using SDN to re-engineer ARP from within the network* by Mark Matties takes [10]. The paper uses a SDN to create an architecture using a service called Distributed Responder for ARP *'DR-ARP'*, while other papers focus on using the SDN controller as an IDPS; this paper uses the SDN primarily for the routing feature in the flow tables. DR-ARP runs as a separate Python service.

Switches are inserted with a set of flow rules that forward ARP requests into the DR-ARP responder service, which maintains an ARP service table; consisting of the standard ARP table containing pairs of MAC address to an expected IP address, plus meta-data on all ARP transactions. The DR-ARP then transmits the newly crafted and valid ARP replies, which the SDN network forwards directly to the requesting host's ARP. Depending on the load and network scale, the service may be a single responder or distributed across multiple responders; which can update each other's ARP service tables. If the service fails to find a matching host in its tables for the ARP request, the service itself sends an ARP request expecting the host itself to reply to update it's local ARP service table. This design prevents an adversary from poisoning ARP with an ARP request replay attack as the service acts as an authenticating ARP proxy demonstrated in Figure 3.1.

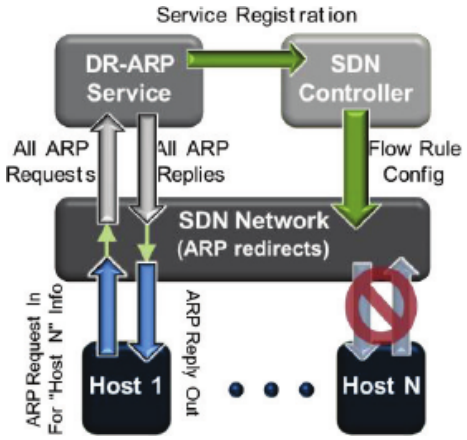

Figure 3.1: Distributed Responder ARP Architecture [10].

Testing was performed within a virtual network on a single machine, using namespaces with ping processes to simulate machines with their own ARP tables. The DR-ARP responder was successfully in acting as an ARP proxy and limiting and validating ARP calls, by dropping invalid ARP packets. This service greatly increased the RTT of the ARP requests from 1ms to 6ms, the author summed this up in part due to inefficient speed a separate Python service using libpcap library to DPI the ARP requests. This implementation also has no ARP flood protection and relies on regular ARP to discover new hosts for it's internal ARP table, which could be poisoned directly to the service causing all subsequent ARP replies from the proxy to be incorrect.

3.2.3 Preventing ARP poisoning attack utilizing SDN paradigm

Masoud et. al. proposes a new algorithm aimed in detecting and mitigating ARP poisoning for a network regardless whether hosts are assigned a dynamic *(SDN_DYN)* or a static IP address *(SDN_STA)* [11].

The algorithm detects the SDN_DYN hosts by inspecting the DHCP packets between the DHCP server and hosts, registering hosts paired MAC address and IP address; handed to them by external the DHCP server. Ensuring static IP SDN_STA addressed machines are addressed, the algorithm monitors new traffic and detect hosts that do not have an existing match within the known hosts table. This paper is the only one that maps the network's topology automatically and accurately registers static IP addressed hosts. Using the list of hosts gathered from SDN_DYN and SDN_STA the algorithm residing within a Ryu SDN controller inspects all ARP reply packets, if the IP address or the MAC address contained in any ARP reply is not the same as the recorded mapped value; the reply is dropped.

Masoud et. al. tested the algorithm on the physical HP-2920-24G switch with SDN enabled, using physical machines connected to the SDN with a mix of Linux and Windows as seen in Figure 3.2. While using the open-source Ryu SDN controller t manage the new algorithm, and an external DHCP server. The team used the software Cain and Abel [49] to perform the ARP poisoning, which managed to assert their algorithm, mitigating all ARP spoofing attacks [11].

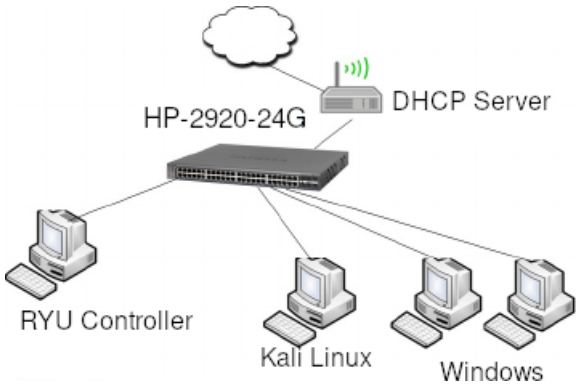

Figure 3.2: Physically Tested setup [11].

The paper introduces an interesting approach utilising the DHCP server to detect new hosts on the network, however, the paper does not take into account the validation of the DHCP server; as it may be possible to run a malicious DHCP server to produce false-positive MAC and IP pairings. The paper also makes a point to indicate that all ARP replies and requests are forwarded to the controller forwarding each validated ARP packet; any subsequent packets need to be re-validated and no flow rules are actively inserted within the switch. This slows down the ARP traffic and flood the controller due to the constant re-validation, rather than inserting a rule for the valid replies.

3.2.4 Mitigating ARP Spoofing Attacks in SDN

The following paper by Abdel Salam et. al. [12], proposes a similar solution to ARP poisoning mitigation as the prior paper by Masoud et. al. The team focused on using the POX SDN controller with the *l2_learning* module. On startup they implemented flow rules to the SDN switches forwarding all ARP and DHCP packets directly to the controller. The controller is designed to monitor the DHCP traffic, upon the DHCP offering an IP address to a host; to then track the paired MAC and IP addresses into a known host list. ARP requests and reply packets received by the controller are regarded as spoofed by the IDPS if their source or destination ethernet header differs from the known host list, or if the controller receives an ARP with a MAC address that is not within it's known host list and dropped. To further protect the SDN, the team implement a controller DoS protection which is similar to a QoS. The implemented controller monitors the ARP throughput on each interface to detect anomalies in ARP traffic, this is achieved by counting the ARP packets; when a large amount is detected the controller implements a rule to temporarily drop traffic from the offending interface. If any of these stated situations occur, the controller registers the event as an attack signature, installing a flow rule on the switch, preventing further incoming traffic.

The POX controller was tested within a virtual Mininet [50] network on a virtual machine, with a range of ARP attacks. The testing results indicated success in mitigating the attacks with a negligible load on the CPU of the controller, the ARP DoS flood was detected within *0.7* seconds while the ARP request and reply attack were detected within around *0.100* milliseconds. The implemented solution ensured throughput speeds dropped only by a fraction from *77.9* Mbps to *72.2* Mbps during an ARP flood, compared to the throughput without any protection.

Similar to the prior paper this implementation utilises the DHCP server, however, it does not take into account static IP addresses unless explicitly defined by the user within the known host's file. The paper also fails to indicate if the DHCP server is validated, leaving it vulnerable to a malicious DHCP server to produce false-positive MAC and IP pairings. The flow rules installed by the controller drop all traffic from the host, the adversary could inject poisoned ARP packets using a known host's MAC address. Using the controller to block the MAC address of a valid host as a form of DoS attack using the ARP protocol.

3.3 ARP IDPS Summary and Comparison

3.3.1 Comparison

Referring to Table 3.1 the papers researched all utilise SDN in some form to mitigate and prevent ARP poisoning, primarily using the SDN's central controller as a DPI to examine packets, mitigating ARP spoofing by dropping or forwarding rules for specific hosts. The exception being Mark Matties [10], in which it runs alongside the controller installing flow rules initially to have DHCP and ARP packets forwarded to itself, then acting as a proxy ARP server by handling ARP replies itself. Only Abdel Salam et. al. [12] actively installs flow rules within switches to drop or forward packets for the future, with the other three opting to receive or drop all within the controller, which increases the controller's processing requirement.

IDPS features	[9]	[10]	[11]	[12]
Utilises DHCP server			X	X
Supports Static IP Addresses	X	X	X	
ARP spoofing mitigation	X	X	X	X
ARP Flood protection		X		X
Acts as proxy ARP	X	X		
Actively inserts flow Rules				X

Table 3.1: ARP SDN IDPS Design and functionality comparison

The testing carried out within these research papers differs vastly, this is seen in Table 3.2 which summaries the testing environment and the type of metrics gathered during testing. Two papers [11] [9] implement and test the ARP IDPS within a physical testbed, with other papers opting to virtualise the environment. However, these papers lack testing metrics, instead of focusing on whether it's successful or not in mitigating a Man-In-The-Middle attack. The other three papers used a variety of metrics, such as the RTT, Detection time and most interestingly paper [12] tested data throughput; where they test the traffic throughput between hosts during an active ARP attack. This testing demonstrates the effect an ARP attack could have on the performance of the host itself, rather than on the controller.

Environment / Testing	[9]	[10]	[11]	[12]
Physical testbed	X		X	
Virtual testbed		X		X
ARP mitigation Testing	X	X	X	X
CPU Load Testing	X			X
RTT Testing	X	X		
Data throughput Testing				X
Detection Time Testing				X

Table 3.2: ARP SDN IDPS Test environment and metrics

3.3.2 Summary

The papers [10] [11] [12] all maintain a list of known hosts comprised of the MAC address and IP address, the solution has been widely used and gives the controller a list to authenticate against. However, only [11] [12] utilises the DHCP server to aid in the creation of the known hosts, yet, do not validate the DHCP packets themselves, this report's proposed system developed a mechanism to verify the DHCP offers. Another issue raised when relying on the DHCP server is the use of static IP addresses, only [11] appears to accommodate for this situation. The IDPS outlined in this paper uses a similar approach in regards to building the network topology utilising DHCP server and supporting static IPs.

The paper by Abdel Salam et. al. [12] utilises the SDN paradigm of actively installing new flow rules within the switches, whilst the other's constantly receive all ARP packets for forwarding or dropping. Requiring to re-authenticate ARP requests and replies causing overhead on the controller, subsequently impeding on the RTT of the ARP packets, due to the forwarding through the controller. These papers likely used the older OpenFlow protocol 1.0 which does not support some key flow rule matching capabilities that OpenFlow 1.3 supports such as ARP specific rules, see Section 2.3. The proposed IDPS resolves the issue by using the newer OpenFlow 1.3 matching fields, which can match ARP packets directly, discussed in Section 2.3. Installing a flow rule for an ARP reply that has been verified reduces traffic to the controller while ensuring a low RTT for ARP, mitigating the stated issues while ensuring the ARP replies are valid.

The examined papers mix between proxy ARP [9] [10] and regular ARP [11] [12]. Proxy ARP offers the advantage of always receiving ARP replies from a valid source, depending on how the proxy builds it's known MAC list. However, it requires larger overhead due to validating and replying to all requests. Whilst the regular ARP has less overhead after authenticating, yet relies on hosts to request and reply to one another. The proposed system remedies this by using a hybrid of the two implementations, relying on regular ARP between hosts using the central IDPS to validate and install flow rules, however, to avoid the issue highlighted in the examination of paper [12] where a MAC address that's blocked can result in a false-positive block of a valid MAC address. The controller temporarily acts as an ARP proxy server for blocked host using the known host list.

Research is often carried out in an easier to obtain virtual environment on standard machines, however, both [9] and [11] tested using a physical testbed. This approach gave the research teams more accurate testing results and expectations of their algorithms in real-world settings, this system follows the same methodology of testing the IDPS on real hardware; this offers more accurate results and accurate testing of the CPU load on the controller.

4 Design & Implementation

This chapter defines the functionality of the project in its entirety, covering the purpose, overview, project requirements, philosophy on utilising methods from prior research papers and designing a new system to mitigate ARP poisoning, accompanied by a high-level design of the IDPS.

4.1 Purpose

The purpose of the proposed IDPS system, which is an extension to an SDN controller, is to detect a variety of ARP poisoning attacks that are then prevented, finally mitigating further attacks launched by the adversary via blocking the intruding MAC address from transmitting ARP packets. The proposed IDPS takes full advantage of the SDN flow tables unlike the prior researched papers [9]- [12], by actively inserting flow rules for specific verified ARP packets rather than the conventional method of forwarding and re-validating all ARP packets within the IDPS. Furthermore, this IDPS system proposes a new hybrid paradigm of ARP IDPS and ARP proxy. Whereby relying on the IDPS to detect poisoned ARP packets, actively inserting flow rules to forward all pre-verified packets and an ARP proxy to manage blocked MAC's ARP traffic to ensure other traffic is not impeded, preventing an adversary from causing a false-positive ARP block on a victim host. Papers [9] [10] rely solely on an ARP proxy as mitigation, ARP proxies handle all ARP packets, slowing down the RTT for ARP while requiring a large overhead on the device to handle the traffic. The goal of the hybrid paradigm counters the disadvantage by relying on an ARP proxy for a small subset of packets, preventing DoS via false-positives while minimising the overhead.

4.2 Operation Overview

Figure 4.1 is a high-level flow chart that covers the overview in a visual representation, the implemented IDPS code is available within Appendix A.1. The IDPS functions by using a *mac_knownlist*, a paired list of known devices on the network, containing the MAC address, last known IP address and the IP origin; DHCP, Static or Unknown. This list enables the IDPS to cross-reference ARP packets to known hosts, aiding in detecting ARP packets attempting to spoof other devices. The IDPS initially sets SDN switches to forward all ARP and DHCP packets directly to the controller, with the default table-miss flow rule to forward traffic as a normal switch; this default rule allows all other traffic to flow like a traditional network and not impede on normal traffic; while minimising the number of packets the controller processes. The IDPS awaits two events, *PacketIn*, when a switch receives an ARP or DHCP packet and *FlowRemoved*, when a flow rule is removed from *idle_timeout* flag trigger.

The DHCP packets are used to update the *mac_knownlist*, by observing the DHCP IP address offers, the IDPS keeps an updated list of devices and their IP addresses. The DHCP server is also validated, using DPI to verify the MAC and IP address of the DHCP server, which is stored within *dhcp_whitelist*. These packets are inspected and forwarded, only dropped when a DHCP offer from an unverified DHCP server is detected.

When an ARP packet enters the controller, a check on the packet's MAC and IP address to the *mac_knownlist* is performed, if new and unused MAC and IP addresses are present, it's then added to the list as a static IP. ARP packets go through a series of DPI validations to detect any potential ARP poisoning methods, see 4.2. An ARP packet that has been verified gets forwarded back into the network and functions as regular ARP. A flow rule matching the specific verified ARP packet with an *idle_timeout* is inserted on the switch. Actively inserting flow rules for verified ARP packets prevents the controller from resource deprivation by re-validating packets, the idle timeout is added to support the changing MAC and IP allocation by the DHCP. If, however, the ARP packet is considered malicious, the IDPS drops the packet and actively insert flow rules to drop any ARP packets from the MAC address; this again includes an idle timeout for similar reasons. During which the IDPS controller acts as an ARP proxy, replying to the blocked MAC ARP requests using the MAC and IP pair stored in the *mac_knownlist*; this is to prevent DoS attacks, where an adversary produces a false positive ARP block by having valid hosts MAC address blocked.

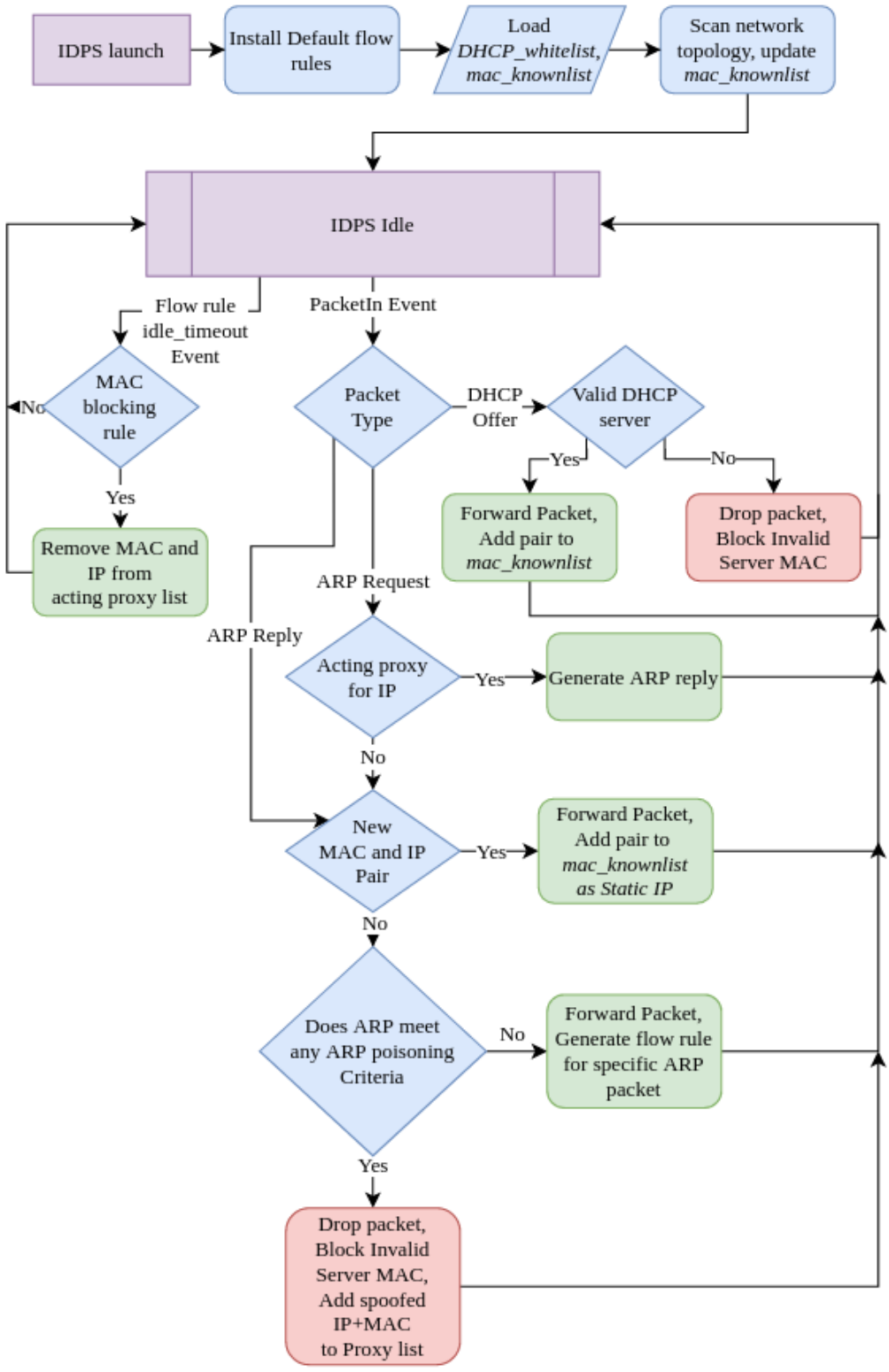

Figure 4.1: IDPS Flow-Chart

4.3 Feature Requirements

The requirements listed in Table 4.1 are key components and features that are required to achieve the desired functionality for this proposed IDPS. They are split between *Primary* priority which is required and cannot function correctly without, and *Support* priority that is not required for the IDPS to function, however, greatly aid in its functionality and useability. The requirements were chosen as the core functionality required for the proposed IDPS and to meet the limitations discussed in other IDPS researched prior.

ID	Function	Description	Priority
F1	Detects and prevents poisoned ARP packets	IDPS views ARP packets detecting and mitigating any poisoned ARP packets	Primary
F2	Known IP to MAC Pairing list	IDPS contains an internal and external list of known network pairings of MAC to IP addresses	Primary
F3	DHCP IP address allocation	IDPS updates known pairing by observing DHCP address allocation	Primary
F4	DHCP Validation	IDPS verifies a DHCP IP offer	Support
F5	Static IP address allocation	IDPS can support static IP addresses alongside DHCP offers within it's known pairings	Primary
F6	MAC blocking	IDPS can block specific MAC addresses from sending ARP packets	Primary
F7	ARP Proxy Server	IDPS can act as an ARP proxy server, by replying to ARP requests	Primary
F8	ARP poisoning log	All ARP poisoning attacks are logged externally	Support
F9	DHCP & ARP packet capture	Switch forwards DHCP and ARP packets, where IDPS can monitor the traffic	Primary
F10	Does not impede regular traffic	No other traffic will be impeded during the actions of the IDPS	Support

Table 4.1: IDPS Feature Requirements

4.3.1 ARP Detection & Mitigation Criteria

Table 4.2 contains a list of different types of ARP poisoning methods and their matching criteria, the proposed IDPS was required to match ARP packets containing one of these signatures. This set of criteria was chosen due to being common ARP poisoning techniques [51]. The importance of the Ethernet Frame fields and ARP packet header fields is covered in the chapter 2 Section 2.8. The criteria are split between request packets and reply packets, a ARP packet is considered poisoned if the packet satisfies one of the required conditions. The ID supplied to each criterion is used within the implementation to log specific types of ARP poisoning methods, and the testing when verifying the functionality.

ID	Attack	Packet Condition	Priority
A1	ARP Request poisoning		
A1.1	ARP IP known spoof	Source IP in the ARP header does not match with the source MAC address from *mac_knownlist*	Primary
A1.2	ARP IP unknown spoof	Source MAC in the ARP header does not exist inside the *mac_knownlist* yet the IP address does	Primary
A1.3	Source MAC mismatch	Source MAC in ether header does not match the Source MAC in ARP header	Primary
A2	ARP Reply poisoning		
A2.1	ARP IP SRC spoof	Source IP in the ARP header does not match with the source MAC address from *mac_knownlist*	Primary
A2.2	ARP IP DST spoof	Destination IP in the ARP does not match with the Destination MAC address from *mac_knownlist*	Primary
A2.3	Source MAC mismatch	Source MAC in ethernet does not match the Source MAC in ARP	Primary
A2.4	Destination MAC mismatch	Destination MAC in ethernet does not match the Destination MAC in ARP	Primary
A2.5	Destination masked	Destination MAC in ethernet has a value of "FF:FF:FF:FF:FF:FF"	Secondary

Table 4.2: IDPS ARP detection & Mitigation Criteria

4.4 Methods

The section explains in-depth the SDN structure used for implementation and the technical methods implemented within the proposed IDPS to achieve the requirements and criteria that were set forth.

4.4.1 SDN Structure

A brief overview of the software and protocols which are used for within the IDPS proposal, how it affects the implementation and what features these SDN tools may have that are advantageous to the system.

Controller

The IDPS proposal is implemented as an extension to the Ryu SDN Controller. One reason for using Ryu rather than the popular researched POX controller such as the papers [9]- [11], is Ryu has kept up to date with protocols. POX only supports OpenFlow 1.0, while Ryu supports OpenFlow up to 1.5, the implementation takes advantage of the updated OpenFlow 1.3 packet matching fields for flow tables as discussed alongside a comparison of controller's in the Chapter 2 Section 2.3. The IDPS relies on the Ryu controller for events triggered by the OpenFlow protocol, such as *Features* to establish switch connection, *PacketIn* to know when a packet is sent to the IDPS for processing and *Flow-Removed* for when the switch has a flow rule removed likely due to idle timeouts.

SDN Switch

The SDN switch implemented is the Open vSwitch, using the OpenFlow 1.3 protocol for communication between the switch and controller. OvS supports several SDN protocols most notably supporting OpenFlow up to 1.5, which is ideal for the proposed IDPS. As previously stated the IDPS relies on OpenFlow 1.3 therefore the proposed IDPS requires similar flow packet matching criteria as OpenFlow 1.3 to function as designed.

4.4.2 Flow Rules

Flow rules are the methodology of how the SDN switches dictates actions on specific packets forwarded within the network, these flow rules are often created by the SDN controller. The proposed IDPS set's initial flow rules and actively inserts flow rules for specific packets, unlike current research proposals which simply focus either forwarding or dropping from within the controller itself.

Default Flow Rules

The code Snippet 4.1 are the flow rules directly installed on the OvS switch after the IDPS launches. The first flow rule matches all ARP packets, directing them to the controller for potential ARP poisoning. Similar to ARP packets, the second flow rule forwards all DHCP packets to the controller for DHCP MAC to IP pairing. As DHCP runs on the application layer where

ARP runs on the Link-Layer there is no matching filter for DHCP directly, therefore the matching criteria are UDP packets from the standard DHCP port *67*. The two rules trigger a *PacketIn* event within the IDPS when a packet is sent to the controller from the switch. The last flow rule has the priority level *0*, being the table-miss entry rule. When a packet fails to meet any higher priority flow rules it ends at the table-miss rule. The proposed system only handles ARP packets and therefore the default flow rule is to forward the traffic as a conventional switch.

```
priority=1,arp actions=CONTROLLER:65509
priority=1,udp,tp_src=67 actions=CONTROLLER:65509
priority=0 actions=NORMAL
```

Code Snippet 4.1: Initial Flow Rules

4.4.3 MAC and IP Address Pairing

As stated the IDPS relies on a MAC to IP pairing (*mac_knownlist*), used to detect a difference between known hosts and an ARP packets fields. The list is generated in three key ways labeled as such: network scanning on launch using a new class `networkControl` to perform an ARP scan *(None)* type, detecting new devices sending ARP packets *(Static)* type and by detecting DHCP IP address offers *(DHCP)* via the *PacketIn* event. This functionality is managed by a newly designed class called `csvMACListTracker` inside the IDPS for managing and controlling an internal and external CSV file pairing; an external file allows for manual alterations, visibility of the *mac_knownlist* and re-usability on restarts.

Storage Format

The *mac_knownlist* is stored within a CSV file format and internally in the IDPS. The first value storing the MAC address and the second containing a Python dictionary which stores the following values: Last known IP address and the origin of said IP address. The snippet below in 4.2 is an example of the storage format.

```
00:00:aa:bb:cc:dd,"{'LastIP':'192.168.4.1', 'Type':'Static'}"
dc:a6:32:0d:13:44,"{'LastIP':'192.168.4.2', 'Type':'DHCP'}"
b8:27:eb:9c:86:b7,"{'LastIP':'192.168.4.106', 'Type':'DHCP'}"
b8:27:eb:97:23:a2,"{'LastIP':'192.168.4.178', 'Type':'None'}"
```

Code Snippet 4.2: Known MAC list pairing format

Static IP addresses

The first scenario the IDPS detects static IP addresses via new ARP packets. An incoming ARP packet entering the IDPS is compared to the *mac_knownlist*, if the MAC source address is unknown and the IP address is unused it's added to the list as a static IP address. Alternatively, the MAC and IP pairing can be manually inserted into the *mac_knownlist* CSV file.

DHCP IP addresses

The second scenario is the IDPS using DPI on DHCP packets, the IDPS forwards all DHCP packets, however, watches for the DHCP server allocating and offering an IP address to a device. The proposed IDPS introduces a new mechanism from the other researched papers which is an initial verification check on the DHCP offer before forwarding the packet or any other actions, the check incorporates a similar technique of verifying MAC and IP addresses pairings to the DHCP server. A file called *dhcp_whitelist* stores a list for valid DHCP servers and is used to verify the DHCP offer. This prevents a malicious machine attempting to trick the IDPS from inserting false MAC to IP address pairings into the *mac_knownlist* by impersonating a DHCP server. Once confirmed the IDPS either adds the new MAC and IP pairing to the *mac_knownlist* for future reference, or updating the existing MAC address within the file with the newly allocated IP address; preventing any potential false-positive MAC blocking from old IP addresses.

4.4.4 ARP Packet Handling

The IDPS handles the *PacketIn* event for ARP and DHCP packets separately, expecting any incoming ARP packets to be potentially malicious. The initial check is performed by comparing the Ethernet frame source MAC address and the ARP source IP address to the *mac_knownlist*, this is to handle the potential static IP addresses; matching and unknown values in both are classified as static devices as discussed in Section 4.4.3. The second check performed concerns the ARP proxy, which cross-references ARP requests destination IP address to the proxy servers handle list, determining if the ARP proxy is required to reply as discussed in Section 4.4.5.

Finally, the IDPS scans ARP packets for potential poisoning by performing similar DPI on ARP requests and ARP replies, breaking down the packet into the Ethernet frame and ARP headers to cross-reference the field values. The IDPS performs a series of checks and verifications on these fields while cross-referencing the *mac_knownlist* looking for any discrepancies between the packets MAC and IP pairings, the methods used to detect ARP poisoning are outlined in the requirements Table 4.2.

Safe ARP Packet Handling

ARP packets that are designated safe are forwarded back to the switch, proceeding normally and to their destined device. A flow rule which matches the specific packet is created using the OpenFlow 1.3 protocol matching fields, where the flow rule uses the specific Ethernet frame headers and ARP headers. As previously stated many researched ARP IDPS's do not actively insert flow rules, forwarded every ARP packet to the controller, to be checked and re-sent, this slows down the ARP traffic throughput as well as increases the CPU load of the controller. To avoid this the proposed IDPS actively installs flow rules to the SDN switch for each verified ARP packet.

The flow rule forwards ARP replies and requests in one direction between these two devices as a normal switch. The flow rule is usually inserted as a

pair, the initial flow rule being the ARP request, the second flow rule being the ARP reply. As seen below in Snippet 4.3. The flow rules are given a *idle_timeout* value, the SDN switch removes these flow rules once no packet triggers the rule for the set amount of time. The purpose of implementing this features is to prevent the switch from forwarding ARP packets with the matching MAC to IP pair for extended periods, preventing cases when the DHCP server offers the same IP to another device, it should not be able to spoof the IP of another device as would be possible if the flow rule remained.

```
1 idle_timeout=120, priority=2,arp,dl_src=b8:27:eb:97:23:a2,
    dl_dst=ff:ff:ff:ff:ff:ff,arp_spa=192.168.4.178,arp_sha=b8
    :27:eb:97:23:a2 actions=NORMAL
2 idle_timeout=120, priority=2,arp,dl_src=b8:27:eb:9c:86:b7,
    dl_dst=b8:27:eb:97:23:a2,arp_spa=192.168.4.106,arp_sha=b8
    :27:eb:9c:86:b7 actions=NORMAL
```

Code Snippet 4.3: ARP packets Flow Rules

Poisoned ARP Packet Handling

ARP packets designated malicious are first dropped by the controller to prevent the packet from reaching its destination. Followed by inserting the adversaries MAC address and IP address pairing into the ARP proxy, where the proxy is now aware of the MAC address to act as a proxy for; discussed further below in Section 4.4.5. Finally, flow rules are created and inserted on the switch as seen in code Snippet 4.4, which when applied actively block ARP packets from the MAC address.

The first flow rule forwards all ARP requests querying for an IP last used by the block MAC address (*192.168.4.106*) to the controller. The IDPS knows the IP address is used by the blocked MAC address by referring to the *mac_knownlist*, declaring the IP belonging to the blocked MAC address *8:27:eb:9c:86:b7*. This flow rule works in conjunction with the proxy, where the requests for a blocked MAC get forwarded directly to the IDPS for the ARP proxy to reply.

The second rule is the MAC blocking flow rule, any ARP packets destined for the blocked MAC address are applied with the *clear_actions* rule, dropping the packet within the switch. The IDPS applies *send_flow_rem* flag to this flow rule, the flag informs the switch that upon the removal of the flow rule due to the *idle_timeout*, the switch inform the controller via the *Flow-Removed* event.

```
1 priority=4,arp,arp_tpa=192.168.4.106 actions=CONTROLLER:65509
2 priority=3,arp,dl_src=b8:27:eb:9c:86:b7 actions=clear_actions
    idle_timeout=300, send_flow_rem
```

Code Snippet 4.4: Blocking MAC's ARP packets Flow Rules

4.4.5 Proxy ARP Operation

The ARP proxies functions to manage ARP requests that are destined for an IP address associated with a blocked MAC address, the purpose of using an ARP proxy for blocked MAC's is to prevent potential DOS attack's

whereby an adversary produces a false-positive block on a MAC address. The implemented class `proxyServer` manages the ARP proxy server that is incorporated within the IDPS, containing a list similar to *mac_knownlist* of blocked MAC addresses and known IP addresses; updated by the IDPS from MAC addresses that produced malicious ARP packets.

The proxy is idle within the IDPS until a *PacketIn* event is triggered and the packet is an ARP request for a IP address that is paired to a MAC address within the proxies list. The ARP proxy performs DPI on the ARP request to generate an ARP reply packet, this is achieved within the class using the Python library *Scapy* as discussed in the Background Section 2.7.

The ARP reply packet created uses the ARP request to the reply. However, due to the flow rule in prior Snippet 4.4, the switch drops all ARP packets from the blocked MAC address dropping the proxies ARP reply. The ARP proxy circumvents this dropping flow rule by combining two features, firstly the proxy generated ARP reply contains the blocked MAC address within the Ethernet frame of the packet while using the IDPS's MAC address within the ARP protocol; creating an impersonated reply packet as seen in Wireshark Figure 4.2.

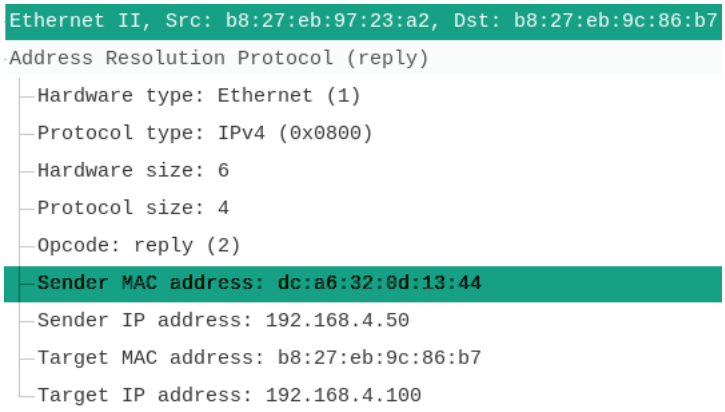

Figure 4.2: Exp 2: Wireshark Proxy ARP

The mixed source MAC addresses create a distinctive ARP reply which the IDPS can create and insert flow rule specifically tailored to. The unique flow rule is strict in the ARP packets it forwards, preventing the adversaries MAC addresses from attempting any other ARP poisoning methods. This is possible due to the flow rule matching capabilities of OpenFlow 1.3, which offers the ability to match Link-Layer protocols such as ARP.

The code Snippet 4.5 below is an example of the flow rule, which has a higher priority ensuring the switch attempts to match it before other ARP flow rules. In the example flow rule is tailored for the ARP proxy based on the the block MAC address *b8:27:eb:9c:86:b7* with the IP *192.168.4.106*; the IDPS MAC address seen as *dc:a6:32:0d:13:44*.

```
1  priority=10,arp,dl_src=b8:27:eb:9c:86:b7,arp_spa
    =192.168.4.100,arp_sha=dc:a6:32:0d:13:44 actions=NORMAL
```

Code Snippet 4.5: Proxy ARP reply Flow Rule for blocked MAC

The ARP proxy relies on the *Flow-Removed* event trigger, generated when the flow rule that is blocking ARP packets from the blocked MAC address *idle_timeout*, which is discussed in Section 4.4.4. This event informs the IDPS that a MAC address is no longer actively blocked, and subsequently that the two flow rules need to be deleted. The flow rules being the ARP request packet for the blocked MAC's IP address that is forwarded to the controller rule in Snippet 4.4, as well as, the ARP proxy reply flow rule designed for the ARP proxy to reply to requests from the block MAC addresses in Snippet 4.5.

4.4.6 Logging

The IDPS provides live on-screen logging of events such as DHCP offers, ARP packets and ARP poisoning, while also providing file logging for ARP poisoning detections. These logs are generated and managed by a newly created mechanism inside the class `csvLogger` generating one file per day. The separation in events and ARP poisoning prevents an attack being lost in the logs. The log file is stored in the CSV format which allows the log to be directly imported to several tools that support the CSV format, such as spreadsheets. The log format is explained below in Table 4.3.

Name	Information
Date/Time	The Date and Time the incident occurred, using the format: YYYY-MM-DD HH/MM/SS/USEC(6 places)
ARP poisoning ID	The ARP poisoning ID is to define what type of ARP poisoning was detected. The ID is the same format as seen in requirements Table 4.2.
Offending MAC Address	The MAC address that is attached to the malicious ARP packet.
Packet	The offending ARP packet as seen by the IDPS and Ryu controller. This contains, Ethernet frame fields and ARP frame fields.
Action Taken	The action taken by the IDPS on detection of the ARP poisoning, likely MAC blocking

Table 4.3: IDPS Log file format

An example of the logging CSV file is seen below in Snippet 4.6. The log shows ARP poisoning ID *1.3*, ARP impersonation verified by observing the packet showing the difference in MAC source address between the Ethernet frame and ARP headers. This attempt led to the MAC *b8:27:eb:9c:86:b7* being temporarily blocked.

```
1 2020-08-16 02:44:12.937436,1.3,b8:27:eb:9c:86:b7,"ethernet(
    dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='b8:27:eb:9c
    :86:b7'), arp(dst_ip='192.168.4.178',dst_mac='ff:ff:ff:ff:
    ff:ff',hlen=6,hwtype=1,opcode=1,plen=4,proto=2048,src_ip
```

```
='192.168.4.1',src_mac='b8:27:eb:97:23:a2')", MAC block
300s
```

Code Snippet 4.6: IDPS log example

4.5 File Structure

The entire IDPS functions primarily within a single directory, except logs which are stored within its directory. The IDPS itself is stored within a single Python3 script, called upon using the Ryu controller to run alongside as an extension. Below is a summary of the files in the folder structure.

- **ARP_IDPS.py** The IDPS Python3 code, Code in Appendix A.1.
- **ARP_IDPS_tool.py** A simple tool to easily List, Add and Delete values in the two CSV files, Code in Appendix A.2.
- **mac_knownlist.csv** The local *mac_knownlist* file, duplicated from the IDPS file, an example in Appendix A.3.
- **dhcp_whitelist.csv** The local *dhcp_whitelist* file, an example in Appendix A.4.
- **logs/log_XXXXXXXX.csv** An example log file in the Logs directory, the log file contains a date stamp in the format YYYYMMDD, an example in Appendix A.5.

4.6 Algorithm Overview

The two algorithms below give an overview of the operation of the IDPS in pseudo-code, it's split between the two core events triggered in the IDPS, the *Flow-Removed* seen in algorithm 4.3 and *PacketIn* seen in algorithm 4.4.

Algorithm 1: IDPS FlowRemoval Event Algorithm

Requires: FlowRemoval OpenFlow Event

```
1 Proxy_list = ()
2 Function FlowRemoval(flw: flow rule):
3     if flow is in Proxy_list then
4         Delete Block Flow rule flw.eth.src.MAC
5         Delete Proxy Flow rules flw.eth.src.MAC
6         Proxy_list remove flw.eth.src.MAC
7 end
```

Figure 4.3: IDPS FlowRemoval Event Algorithm

Algorithm 2: IDPS PacketIn Event Algorithm

```
   Requires: PacketIn OpenFlow Event
 1 mac_knownlist = mac_knownlist.csv
 2 dhcp_whitelist = dhcp_whitelist.csv
 3 Proxy_list = ()
 4 Function PacketIn(pkt: packet):
 5     if pkt == DHCP then
 6         if DHCP.offer.src.mac is in dhcp_whitelist then
 7             mac_knownlist add DHCP.offer.pair
 8         else
 9             Block Flow rule DHCP.offer.src.mac
10             Proxy Flow rules DHCP.offer.src.mac
11             Proxy_list add DHCP.offer.src.mac
12     else if pkt == ARP Req and pkt.arp.dst.ip is in Proxy_list then
13         Drop ARP.request
14         ARP reply to ARP.eth.src.mac from Proxy_list.mac
15     else if pkt == ARP and (pkt.eth.src.mac and pkt.arp.src.ip)
        not in mac_knownlist then
16         mac_knownlist add ARP.pair
17     else if pkt == ARP then
18         if (ARP.eth.src.mac and ARP.arp.src.ip) !=
            mac_knownlist.pair then
19             Set ARP.attack = True
20         else if (ARP.eth.dst.mac and ARP.arp.dst.ip) !=
            mac_knownlist.pair then
21             Set ARP.attack = True
22         else if ARP.eth.src.mac not in mac_knownlist and
            ARP.arp.src.ip is in mac_knownlist then
23             Set ARP.attack = True
24         else if ARP.eth.src.mac != ARP.arp.src.mac then
25             Set ARP.attack = True
26         else if ARP.eth.dst.mac != ARP.arp.dst.mac then
27             Set ARP.attack = True
28         else if ARP.opcode == Reply and ARP.eth.dst.mac ==
            "FF:FF:FF:FF:FF:FF" then
29             Set ARP.attack = True
30         if ARP.attack == True then
31             Block Flow rule ARP.eth.src.mac
32             Proxy Flow rules ARP.eth.src.mac
33             Proxy_list Add ARP.eth.src.mac
34             Log ARP poisoning by ARP.eth.src.mac
35         else
36             Add Flow rule ARP.eth.src.mac to ARP.eth.dst.mac
37             Forward ARP packet
38 end
```

Figure 4.4: IDPS PacketIn Event Algorithm

5 Testbed Implementation

To evaluate the performance of the proposed IDPS, a physical testbed was been implemented. This chapter covers the SDN testbed network architecture and topology, components and networking tools used as part of the testbed and as part of the testing.

5.1 Components

The testbed consists of a TP-Link Archer AC1750 router using OpenWRT firmware installed Open vSwitch with OpenFlow 1.3 and dnsmasq service acting as DHCP server, RaspberryPi 4 as the SDN controller and IDPS and finally two RaspberryPi 3's acting as a Benign host and Attacker. This section covers, in more detail, the network components, their setup, services and software.

5.1.1 TP-Link Archer AC1750 C7

The TP-Link Archer AC1750 C7 router is an off-the-shelf router, the vendor firmware supplied by TP-link does not support OpenFlow or SDN features. Therefore, the open-source firmware OpenWRT Section 2.5 was installed onto the router. The Archer router was chosen as it is extensively supported by OpenWRT, with detailed instructions within their documentation [52].

OpenWRT Configuration

The Figure 5.1 is provided by OpenWRT, consisting of the default inner structure of the network.

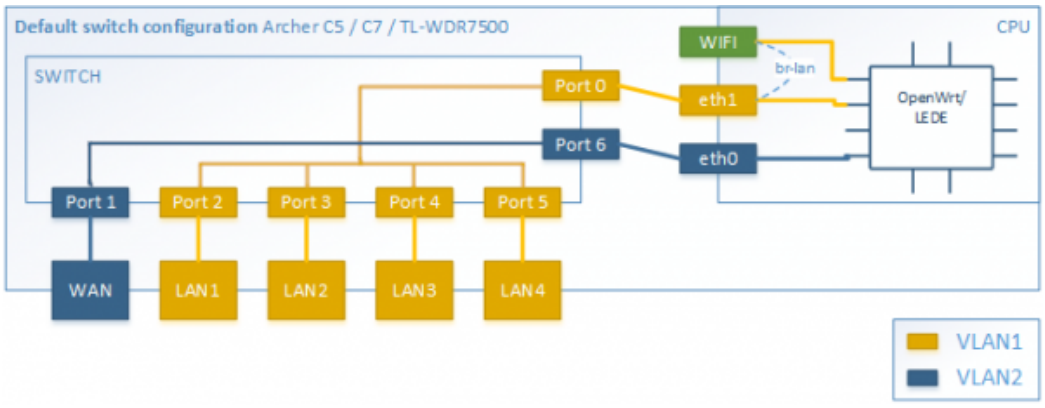

Figure 5.1: Default Router internal configuration [52].

The router consists of two core components, switch and CPU. When devices are connected to *Port 2* and *Port 3* they can communicate through the switch without any traffic being seen within the CPU. As traffic passed through the switch is not visible to the CPU, the OpenFlow switch running

within the router cannot monitor any traffic, preventing the SDN switch inside the router from being able to control the flow of traffic.

SDN Configuration

Figure 5.2 shows the implemented SDN configuration for OpenWRT on the Archer router. The switch is bypassed and forces traffic to flow through the CPU. This is achieved by each physical port on the switch being allocated its static VLAN, with each device on a separate VLAN they are unable to directly communicate with each other. These VLAN's provide each physical port on the router as a virtual interface connected to *eth1*, this common interface within the CPU allows the different VLANs to send traffic between each other while also having traffic visible within the CPU and the OvS switch. The configuration file `/etc/config/network` seen in Appendix A.6, is the interface configuration file for OpenWRT and implements this inner network structure.

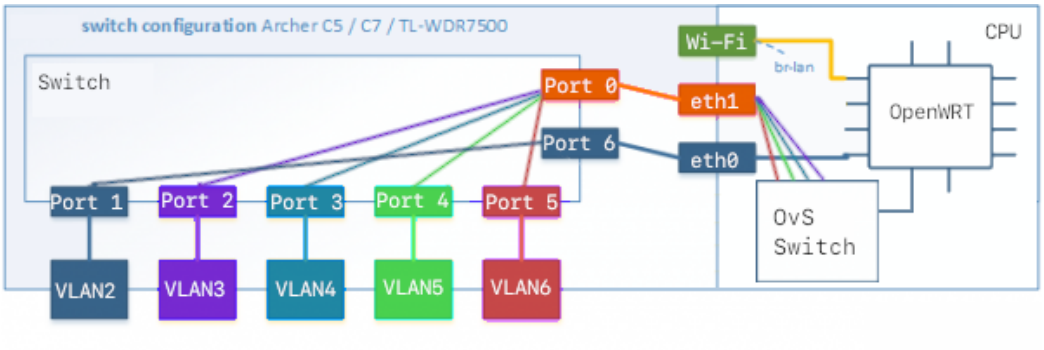

Figure 5.2: SDN switch internal configuration

Open vSwitch

The OpenWRT firmware does not directly support SDN within its configuration. However, the package manager (*OPKG*) contains the Open vSwitch package for installation seen in Section 2.5, which is then configured via the terminal. The package can be installed in the OpenWRT terminal using the commands in Snippet 5.1.

```
$ opkg update && opkg install openvswitch
```

Code Snippet 5.1: OvS installation

The package comes with the *ovs-vsctl* tool which is used to create an OvS bridge with SDN switch capabilities. The Snippet 5.2 is the command used to create the *br-ovs* bridge with the OpenFlow 1.3 protocol, the IP address of the Ryu controller, the virtual interfaces and setting `set-fail-mode` state set as *Standalone*, setting the forwarding control as a traditional switch when failing to connect to the SDN controller. The second command sets the bridge the IP address *192.168.4.1*, that's used as a gateway and the DHCP server IP address.

```
# Create OVSwitch Bridge using the 4 VLANS
$ ovs-vsctl add-br br-ovs -- set bridge br-ovs protocols=
   OpenFlow13 -- set-controller br-ovs tcp:192.168.4.2:6633
   -- set-fail-mode br-ovs standalone -- add-port br-ovs eth1
   .3 -- add-port br-ovs eth1.4 -- add-port br-ovs eth1.5 --
   add-port br-ovs eth1.6
# Set Bridge IP address
$ ip a a 192.168.4.1/24 dev br-ovs
```

Code Snippet 5.2: OvS setup commands

The SDN switch can be verified by using the `ovs-vsctl show` command, which also indicates if the Ryu controller has connected to the switch.

```
root@OpenWrt:~$ ovs-vsctl show
b29855e0-97a9-482d-a7a0-1436473f395a
    Bridge br-ovs
        Controller "tcp:192.168.4.2:6633"
          is_connected: true
        fail_mode: standalone
        Port "eth1.6"
            Interface "eth1.6"
        Port "eth1.4"
            Interface "eth1.4"
        Port "eth1.3"
            Interface "eth1.3"
        Port "eth1.5"
            Interface "eth1.5"
        Port br-ovs
            Interface br-ovs
                type: internal
    ovs_version: "2.11.3

```

Code Snippet 5.3: OvS Verification

dnsmasq DHCP server

The OpenWRT firmware uses the tool *dnsmasq* [53] as it's DHCP and DNS server, the testbed utilises this service to configure a DHCP server on the OvS bridge and SDN network; which the IDPS monitors to determine hosts on the network. This is done simply by updating the configuration file stored in `/etc/dnsmasq.conf`, seen below in Snippet A.7, to offer IP addresses within the SDN subnet *192.168.4.0/24*, the configuration file allows setting static DHCP hosts and IP addresses for the devices; such the DHCP reserving and offering the Ryu controller *192.168.4.2*.

```
dhcp-range=192.168.4.50,192.168.4.200,8h
dhcp-host=d0:50:99:82:e7:2b,192.168.4.2 #Controller
dhcp-host=b8:27:eb:97:23:a2,192.168.4.50 #Host one
dhcp-host=b8:27:eb:9c:86:b7,192.168.4.100 #Host Two
```

Code Snippet 5.4: OpenWRT dnsmasq config file

5.1.2 Pi4 SDN Controller

A RaspberryPi 4 B is utilised as the SDN controller, specifications of which are ARM Quad-core Cortex-A72, 2GB RAM and 1GB NIC card, using the latest version of Raspbian codename *Buster*; a Debian based OS for the RaspberryPi single-board computer [54]. This device is connected directly to the TP-Link Archer router on VLAN 3 on the first SDN switch interface.

Ryu setup

The Ryu controller was choosen due to being Python3 based, easily installed and runs on an ARM-based chip, while supporting OpenFlow up to 1.5. Ryu is not available as a packaged installation in Raspbian, however, is open source and relies on GitHub as it's versioning repository. To download Ryu, the following command is seen in Snippet 5.5:

```
$ git clone https://github.com/faucetsdn/ryu .
$ pip install .
```

Code Snippet 5.5: Ryu Installation

Executing the Ryu controller with a specified component, such as the proposed IDPS, is achieved by simply calling the Ryu manager and the Python3 script containing the IDPS as seen in Snippet 5.6:

```
$ sudo ryu-manager ./ARP_IDPS.py
```

Code Snippet 5.6: Ryu Execution

5.1.3 Pi3 Hosts

The testbed consists of two RaspberryPi 3 B+ devices, the specification being ARM Cortex-A53, 1GB Ram with a 1GB NIC over USB 2.0 [55], running the latest version of Raspbian. As these two devices are the same specification the testbed offers a more balanced testing platform. Similar to the Pi4 the benign host *phy-host-one* and attacker host *phy-host-two* are connected on the TP-Link Archer router via VLAN's, 4 and 6 respectively.

ARP Poisoning Configuration

Configuration on the network interface is required on the attacking host to sniff packets and perform ARP Poisoning attacks. Specifically, the networking interface needs to be configured into promiscuous mode. Promiscuous mode opens the interface to forward all network traffic it receives to the CPU rather than passing only the Ethernet Frames that the host is destined to receive. Enabling promiscuous mode is done through the *Iproute2* tool as seen in Snippet 5.7:

```
ip link set eth0 promisc on
```

Code Snippet 5.7: Enable Promiscuous Mode

5.1.4 Network Time Protocol

The Raspberry Pi's within the testbed are required to have the same internal time, as part of testing requires noting the time taken for packets to travel across the network in milliseconds. Raspberry Pi's do not contain a real-time clock onboard, therefore, the time on each Pi can drift. To ensure valid results during testing the testbed uses the Network Time Protocol (*NTP*), the standard networking protocol for clock synchronization between devices. NTP can achieve better than one-millisecond accuracy in small local LAN's, which the testbed meets the criteria for, resulting in the Pi's having accurate *Stratum 2* time. The RaspberryPi4 and controller is set as an NTP server and the two RaspberryPi3's are NTP clients, meaning the two RaspberryPi3's get the time from the controller; ensuring devices are configured with the same time, as seen in Snippet 5.8. The configurations used for both are available in Appendix A.10 and A.11.

```
pi@phy-host-one:~ $ ntpq -pn
     remote           st t when poll reach   delay   offset
==============================================================
*192.168.4.2         2 u  513  512  377    1.077    0.151
```

Code Snippet 5.8: NTP Sync to Controller

5.1.5 Network Summary

Table 5.1 summarises the devices connected to the Testbed and their respective interfaces, IP addresses and MAC addresses.

Device	Device Hostname	Network Interface	IP address	MAC Address
Open vSwitch	openwrt	br-ovs	192.168.4.1	00:00:aa:bb:cc:dd
Ryu Controller	phy-ryu	eth1.3	192.168.4.2	dc:a6:32:0d:13:44
Benign Host	phy-host-one	eth1.4	192.168.4.50	b8:27:eb:97:23:a2
Attacker	phy-host-two	eth1.6	192.168.4.100	b8:27:eb:9c:86:b7

Table 5.1: Testbed Hosts network Configurations

5.1.6 Network Architecture

The network devices are connected in a star network topology, whereby every device is connected with the TP-Link Archer AC1750 acting as the central hub containing the SDN switch. The implementation is the simplest network form and forces all traffic in the network to pass through the SDN OvS switch. Figure 5.3 below shows the testbed environment, including the services they provide, while Figure 5.4 is a picture of the physical testbed.

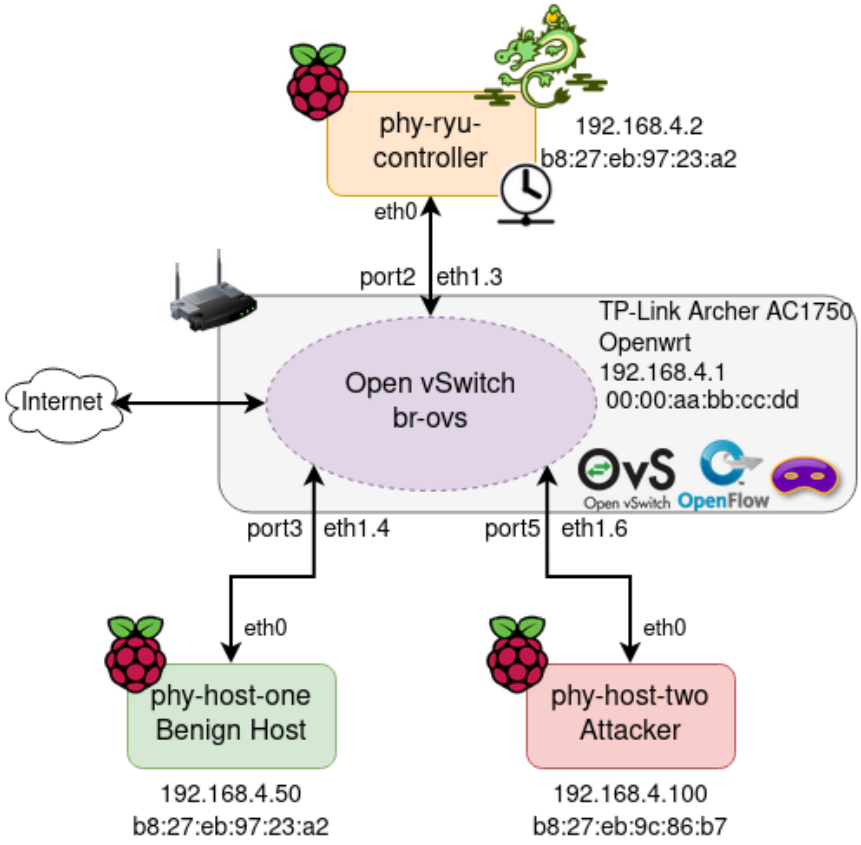

Figure 5.3: Testbed Network Topology

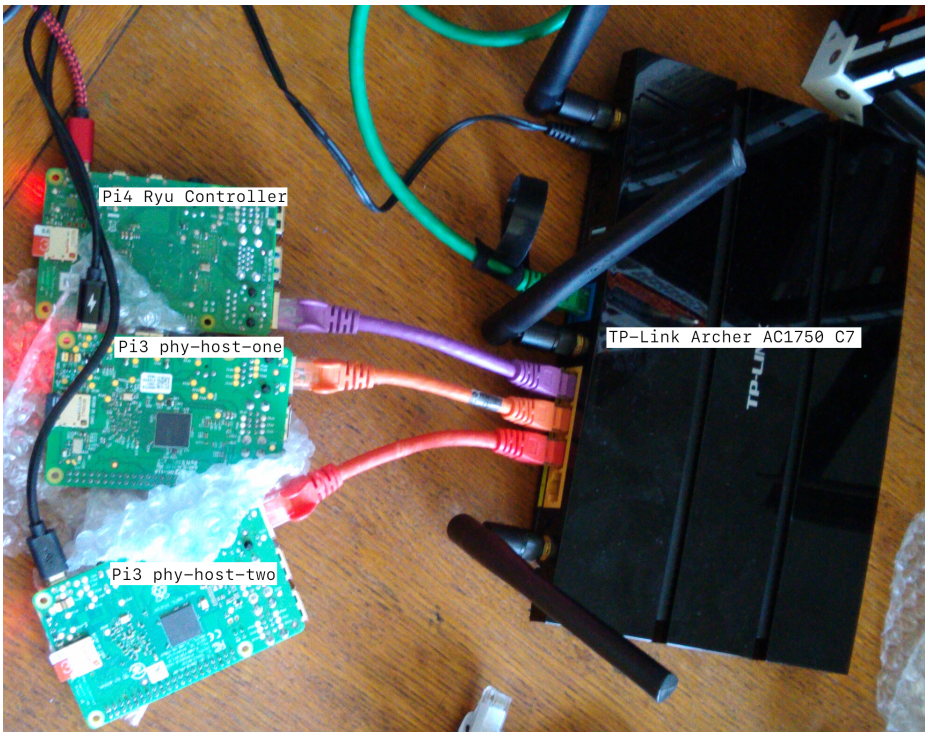

Figure 5.4: SDN switch internal configuration

5.2 Custom Tools

As part of testing and development two custom tools were specificaly created, this section briefly covers their purpose, functionality and implementation.

5.2.1 IDPS Configuration Tool

off An external command-line interface tool called *ARP_IDPS_tool* was created to offer user configuration of the CSV files the IDPS relies upon, such as *mac_knownlist* and *dhcp_whitelist*. This allows users to add Static IP addresses and MAC address pairings to the *mac_knownlist*, or adding or deleting DHCP servers from the *dhcp_whitelist*. This tool is a simple Python3 script which incocprates the `csvMACListTracker` class, the Snippet 5.9 is the help menu from the tool which is available in the Appendix A.2.

```
pi@phy-ryucontroller-01:~ $ ./ARP_IDPS_tool.py -h
usage: [-h] [-l] [-a ADD] [-d DELETE] [-f] file

Simple tool to Manage ARP IDPS CSV files

positional arguments:
  file        CSV file to use

optional arguments:
  -h, --help  show this help message and exit
  -l          List CSV file
  -a ADD      Add Mac to CSV
  -d DELETE   Delete row Containg MAC to CSV
  -f          Flush entire CSV file
```

Code Snippet 5.9: ARP_IDPS_Tool Help Menu

5.2.2 IDPS Testing Tool

As the proposed system was developed to meet a set of criteria (*Table 4.2*) for detecting ARP Poisoning, a tool is required to generate the specific ARP poisoning packets. To avoid using a variety of tools that implement these attacks in their versions, resulting in different packets and timings, a custom tool was specifically developed. Additionally, there is the option of measuring CPU load, required when measuring the proposed system on the controller. The tool is called *ARP_IDPS_test* and ran via the command line, the source code can be seen in Appendix A.12. The tool uses simple command-line options to determine what ARP poisoning packets or test to perform and the parameters the test should use, such as how many times to perform the action and the frequency.

ARP Request & Reply

The tool generates standard and poisoned ARP packets for testing the proposed IDPS effectiveness, relying on the Python3 *scapy* library to create and send the individual ARP packets. The library offers a simple function to create packets as seen in Snippet 5.10, with arguments to insert any combination of MAC and IP pairings in Ethernet Frame and ARP Headers.

```
scapy.Ether(dst=DSTMAC, src=SRCMAC) / scapy.ARP(op="who-has",
    hwsrc=SRCMAC, psrc=SRCIP, hwdst=DSTMAC, pdst=DSTIP)
```

Code Snippet 5.10: Scapy ARP packet creation

This flexibility offered by *scapy* allows the generation of ARP packets meeting the proposed IDPS criteria, such as the mismatch of source MAC address in the Ethernet Frame and ARP Header (*A2.3*).

CPU Testing

The CPU load function is achieved using the Python library *PSutil* as discussed in Section 2.7. The library offers many resource system monitoring options, however, the tool relies on the library for measuring the total CPU load consistently over an extended period. This is achieved by measuring each CPU core activity and dividing it by the total core's to calculate the average load.

Log File's

Each test performed by the tool produces a CSV log file similar to the proposed IDPS logging as described in Section 4.4.6. The CSV format ensures the logs can be easily imported into spreadsheet software and easily comparable against the IDPS log, the tool in Appendix A.13 is one such tool used to calculate the time difference between the logs for RTT testing.

Table 5.2 and Snippet 5.11 are an example log used for testing an ARP request Poisoning criteria.

Name	Information
Packet #	Packet Number
Date/Time	The Date and Time when a packet was sent, down to microseconds
Response	If the ARP request had a reply

Table 5.2: IDPS tool log example

```
0,2020-08-24 16:41:59.314034,0
```

Code Snippet 5.11: Test tool log Example

Example Usage

The Snippet 5.12 is an example of using the testing tool from the Attacker host, the tool performs an ARP poison using the *A1.1* criteria, attempting the ARP every ten seconds twenty five times. The full help menu can be seen in Appendix A.14

```
  sudo ./ARP_IDPS_test.py -p 1.1 -c 25 -t 10
```

Code Snippet 5.12: ARP Poison test tool Usage

6 Experiments and Results

A total of three experiments were performed on the proposed and implemented IDPS discussed in Chapter 4, the IDPS code is available within Appendix A.1. The experiments consisted of testing the IDPS at detecting a variety of poisoned ARP packets, the impact the proposed IDPS has on ARP traffic RTT and the overhead imposed on the controller. Each experiment was performed on the physical testbed see Chapter 5, any alterations to the testbed setup is explicitly stated within the experiment itself.

6.1 Experiment Limitations

The experiments used the specifically created testing tool from Section 5.2.2 to gather results. The tool relies on the Python3 *Scapy* library to send and receive packets, the library introduces a delay in packet speed as it pulls Link-Layer ARP packets out to the application and vice versa. This delay is noticeably visible when comparing RTT of ARP packets using the *arping* tool and *scapy* library, as seen in Table 6.1 which contains the average RTT of an ARP request.

Tool	ARP 1	ARP 2	ARP 3	Average
arping	579.1usec	582.0usec	609.1usec	0.59ms
scapy	42.2ms	42.1ms	42.0ms	42.1ms

Table 6.1: arping And Scapy ARP RTT Limitaition

Therefore, all testing relies on the *scapy* library to maintain a consistent testing environment, this includes normal ARP packets and poisoned ARP packets. The only exception being the Ryu Controller which receives packets via the OpenFlow *PacketIn* event.

6.2 Experiment 1: Variety of Poisoning Detection and Mitigation

The first experiment aimed to verify the proposed IDPS core operation. The experiment was performed to determine that the proposed IDPS met the objective of detecting and mitigating poisoned ARP packets, regardless of specific metrics such as RTT and CPU usage. This experiment uses the earlier defined ARP poisoning criteria Table 4.2 also seen below in Table 6.2, to perform a variety of ARP poisoning attacks conducted against the Benign host from the Attacker host. To determine how accurate the proposed system is at detecting and mitigating the poisoned ARP packets, and how quickly there detected.

ID	Attack
A1	ARP Request
A1.1	ARP IP known spoof
A1.2	ARP IP unknown spoof
A1.3	Source MAC mismatch
A2	ARP Reply
A2.1	ARP IP SRC spoof
A2.2	ARP IP DST spoof
A2.3	Source MAC mismatch
A2.4	Destination MAC mismatch
A2.5	Destination masked

Table 6.2: IDPS ARP detection Criteria

6.2.1 Experiment Setup

The tests in this experiment use the MAC address and IP address pairing in the *mac_knownlist* seen below Snippet 6.1.

```
00:00:aa:bb:cc:dd,"{'LastIP':'192.168.4.1', 'Type':'DHCP'}"
dc:a6:32:0d:13:44,"{'LastIP':'192.168.4.2', 'Type':'DHCP'}"
b8:27:eb:97:23:a2,"{'LastIP':'192.168.4.50', 'Type':'DHCP'}"
b8:27:eb:9c:86:b7,"{'LastIP':'192.168.4.100', 'Type':'DHCP'}"
```

Code Snippet 6.1: Known MAC list pairing file

The ARP attacks were conducted and logged by *phy-host-two* using the specifically created Python attacker tool in Section 5.2.2. The proposed IDPS running alongside the Ryu Controller maintains it's own log file containing attemped and prevented ARP poisoning packets.

6.2.2 Measurements

The focus of this expierment is validation of the detection and mitigation. Therefore, the metrics gathered for the experiment are the following:

- **Count:** The amount of poisoned ARP packet that were detected, E.G dropped and MAC blocked by IDPS.
- **Time:** The IDPS detection time, which is calculated by subtracting the IDPS detection time and ARP message sent time by Attacker.

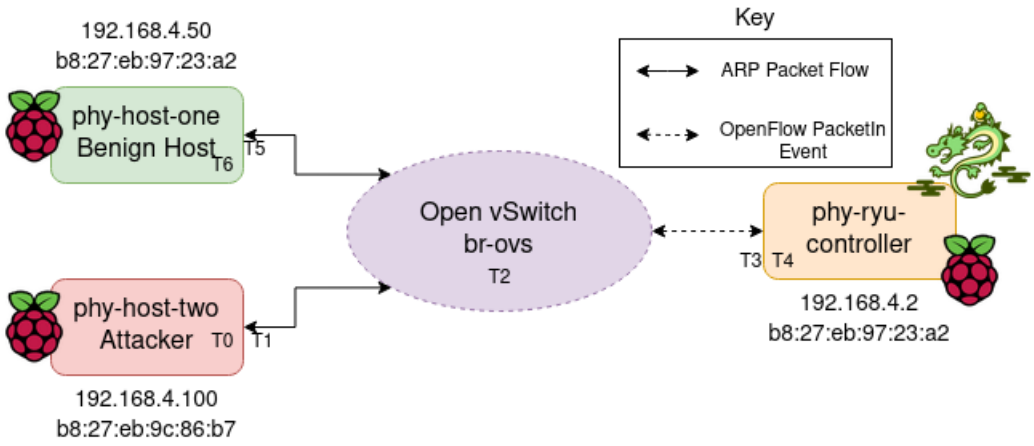

Figure 6.1: Exp 1: Setup and Measurement Points

Figure 6.1 shows the testbed setup containing the potential sampling points to detect ARP packets and their detection time, seen as **T0** to **T6**. This experiment uses **T0** and **T4** sampling points to determine the packet detection & prevention time, these points are the testing tool sending an ARP packet and the proposed IDPS detecting and preventing the ARP packet. The time difference seen between Attacking tool sending the packet log and the IDPS prevention log is compared to calculate the time it took to detect the poisoned ARP. The testing tool sends a poisoned ARP packet every *10* seconds with the IDPS setting an *idle_timeout* on blocked MAC addresses to *5* seconds, this means the flow rule actively blocking the MAC has been removed between packets. This is conducted fifty times to calculate the average detection and prevention time.

6.2.3 Poisoned ARP Request Detection & Mitigation

The poisoned ARP requests packets are criteria *A1.1*,*A1.2* and *A1.3*. These are tested by host *phy-host-two* sending poisoned ARP packets to *phy-host-one*, where the IDPS should detect and block the *phy-host-two* MAC address.

Test A1.1 - ARP IP spoof

Test *A1.1* is the most common type of poisoned ARP, which consists of an Attacker impersonating a host to a Victim, tested by the Attacker host sending an ARP request to the Victim with a spoofed source IP address in the ARP header. In the test the Attacker is impersonating the gateway IP address *192.168.4.1*, to the Benign host which the IDPS knows by the *mac_knownlist* is not owned by his MAC address; as seen in Snippet 6.2.

```
1 Ether(MAC_DST=FF:FF:FF:FF:FF:FF, MAC_SRC=b8:27:eb:9c:86:b7)
2 ARP(MAC_SRC=b8:27:eb:9c:86:b7, IP_SRC=192.168.4.1, MAC_DST=FF
    :FF:FF:FF:FF:FF, IP_DST=192.168.4.50)
```

Code Snippet 6.2: Scapy A1.1 Test

Test A1.2 - ARP Unknown MAC IP spoof

Test *A1.2* is the same as the prior test *A1.2* except the IDPS does not know the adversaries MAC address, therefore the MAC address is blocked but no proxy given. This test requires the use of the *IDPS Config tool* discussed in Section 5.2.1, this is to remove the Attackers MAC address from the *mac_knownlist*.

Test A1.3 - ARP Source MAC Mismatch

The last ARP request test conducted detects and mitigates ARP IP poisoning when the source MAC address is mismatched. Specifically, when the Ethernet frame source MAC address differs from the source MAC address within the ARP payload, as seen in Snippet 6.3.

```
Ether(MAC_DST=FF:FF:FF:FF:FF:FF, MAC_SRC=b8:27:eb:9c:86:b7)
ARP(op="who-has", MAC_SRC=b8:27:eb:9c:86:ff, IP_SRC
    =192.168.4.100, MAC_DST=FF:FF:FF:FF:FF:FF, IP_DST
    =192.168.4.50)
```

Code Snippet 6.3: Scapy A1.3 Test

6.2.4 Poisoned ARP Reply Detection & Mitigation

The poisoned ARP reply packets are criteria *A2.1*,*A2.2*,*A2.3*,*A2.4* and *A2.5*. These tests differ in how they are produced, ARP reply packets are often sent as a reply to an ARP request. However, these tests relied on the custom attacker tool instead, sending gratuitous ARP replies to *phy-host-one*, whereby the receiver has not requested the ARP reply.

Test A2.1 & A2.2 - ARP IP spoof

Testing *A2.1* is similar to *A1.1* however, instead of an ARP request, sending a gratuitous ARP reply. Whereby the IDPS detects poisoned packets if the IP address within the ARP belongs to the MAC address. Criteria *A2.1* checks the Source IP and MAC pairing as seen in Snippet 6.4.

```
Ether(MAC_DST=b8:27:eb:97:23:a2, MAC_SRC=b8:27:eb:9c:86:b7)
ARP(op="is-at", MAC_SRC=b8:27:eb:9c:86:b7, IP_SRC
    =192.168.4.1, MAC_DST=b8:27:eb:97:23:a2, IP_DST
    =192.168.4.50)
```

Code Snippet 6.4: Scapy a25.1 Test

Criteria *A2.2* checks the destination IP and MAC pairing as seen in Snippet 6.5.

```
Ether(MAC_DST=00:00:aa:bb:cc:dd MAC_SRC=b8:27:eb:9c:86:b7)
ARP(op="is-at", MAC_SRC=b8:27:eb:9c:86:b7, IP_SRC
    =192.168.4.100, MAC_DST=00:00:aa:bb:cc:dd, IP_DST
    =192.168.4.50)
```

Code Snippet 6.5: Scapy A2.2 Test

Test A2.3 & 2.4 - ARP MAC Mismatch

Testing criteria of *A2.3* and *A2.4* is a simple validation for poisoned ARPs by ensuring the MAC addresses within the Ethernet Frame and ARP Header match. Criteria *A2.3* is the source MAC addresses match, while *A2.4* is the verification of the destination MAC addresses. Shown in Snippets 6.6 and 6.7 respectively.

```
Ether(MAC_DST=b8:27:eb:97:23:a2, MAC_SRC=00:00:aa:bb:cc:dd)
ARP(op="is-at", MAC_SRC=b8:27:eb:9c:86:b7, IP_SRC
    =192.168.4.100, MAC_DST=b8:27:eb:97:23:a2, IP_DST
    =192.168.4.50)
```

Code Snippet 6.6: Scapy A2.3 Test

```
Ether(MAC_DST=00:00:aa:bb:cc:dd) MAC_SRC=b8:27:eb:9c:86:b7)
ARP(op="is-at", MAC_SRC=b8:27:eb:9c:86:b7, IP_SRC
    =192.168.4.100, MAC_DST=b8:27:eb:97:23:a2, IP_DST
    =192.168.4.50)
```

Code Snippet 6.7: Scapy A2.4 Test

Test A2.5 - ARP Masked Destination

The final test and criteria *A2.5* is a validating the gratuitous ARP reply by MAC destination, ensuring the packet is addressed and not being broadcasted. Therefore, the ARP verification checks the MAC destination is not *FF:FF:FF:FF:FF:FF*, as seen in Snippet 6.8.

```
Ether(MAC_DST=FF:FF:FF:FF:FF:FF, MAC_SRC=b8:27:eb:9c:86:b7)
ARP(op="is-at", MAC_SRC=b8:27:eb:9c:86:b7, IP_SRC
    =192.168.4.100, MAC_DST=FF:FF:FF:FF:FF:FF, IP_DST
    =192.168.4.50)
```

Code Snippet 6.8: Scapy A2.5 Test

6.2.5 Results

Experiment 1 results are presented in Table 6.3 and Figure 6.6, which contains the ARP poison criteria tested, attempts made, detection rate and the average time taken to detect and prevent the ARP packet. The results indicated the proposed IDPS can accurately detect and prevent poisoned ARP packets within *58.1ms*, this average time is extremely efficient considering Table 6.1 indicates an ARP RTT takes approximately *42.1ms*. The *16ms* difference between a RTT ARP and detection and prevention of a poisoned ARP is a minimal time difference, demonstrating the efficiency of the proposed IDPS. The average detection time between ARP requests and ARP reply attacks shows no significant difference in detection times regardless of the type of ARP poisoning technique deployed.

Furthermore, the IDPS demonstrated an accurate assessment of ARP packets by accurately detecting every poisoned ARP packet forwarded through the SDN network, as testing involved standard ARP packets between hosts during testing there was no indication of miss identification.

ID	Attack	Attacks	Detected	Detection Time Avg
A1	**ARP Request**			
A1.1	ARP IP known spoof	50	50	59.9ms
A1.2	ARP IP unknown spoof	50	50	57.5ms
A1.3	Source MAC mismatch	50	50	59.3ms
A2	**ARP Reply**			
A2.1	ARP IP SRC spoof	50	50	59.3ms
A2.2	ARP IP DST spoof	50	50	58.1ms
A2.3	Source MAC mismatch	50	50	57.6ms
A2.4	Destination MAC mismatch	50	50	56.6ms
A2.5	Destination masked	50	50	56.4ms
	Total/Avg	**400**	**400**	**58.1ms**

Table 6.3: Exp 1: Total Result's Table

Figure 6.2 displays the network capture during the testing of criteria *A1.1*, the OpenFlow *PacketIn* event is triggered as seen by packet *9* and *11*, where the IDPS drops the ARP packet which cannot be seen in the monitoring as it's never forwarded into the OvS switch.

	Source	Destination	Port	Protocol	Info
5	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_ECHO_REQUEST
6	192.168.4.2	192.168.4.1	49552	TCP	6633 → 49552 [ACK] Seq=1 Ack=9 Win
7	192.168.4.2	192.168.4.1	49552	OpenFlow	Type: OFPT_ECHO_REPLY
8	192.168.4.1	192.168.4.2	6633	TCP	49552 → 6633 [ACK] Seq=9 Ack=9 Win
9	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_PACKET_IN
10	192.168.4.2	192.168.4.1	49552	TCP	6633 → 49552 [ACK] Seq=9 Ack=111 W
11	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_PACKET_IN
12	192.168.4.2	192.168.4.1	49552	TCP	6633 → 49552 [ACK] Seq=9 Ack=213 W
13	192.168.4.2	192.168.4.1	49552	OpenFlow	Type: OFPT_FLOW_MOD
14	192.168.4.1	192.168.4.2	6633	TCP	49552 → 6633 [ACK] Seq=213 Ack=105
15	192.168.4.2	192.168.4.1	49552	OpenFlow	Type: OFPT_FLOW_MOD
16	192.168.4.1	192.168.4.2	6633	TCP	49552 → 6633 [ACK] Seq=213 Ack=217

Figure 6.2: Exp 1: A1.1: Wireshark PacketIn Event, Blocked ARP

Figure A.7 and Figure 6.4 are the IDPS live log of detection and prevention during testing criteria *A2.1*. The first figure is the IDPS indicating and logging the detection of the poisoned ARP and blocking further traffic, the second figure is the OpenFlow rules set within the OvS switch that are actively inserted to carry out the IDPS MAC blocking.

```
ARPID: (2.1) ARP spoofed | MAC b8:27:eb:9c:86:b7 -> 00:00:aa:bb:cc:dd | for
 IP: 192.168.4.1
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Acting as proxy server in meantime for 192.168.4.100
```

Figure 6.3: Exp 1: ARP A2.1 IDPS Detection Log

```
root@OpenWrt:~# ovs-ofctl -O OpenFlow13 dump-flows br-ovs
 cookie=0x0, duration=44.315s, table=0, n_packets=0, n_bytes=0, priority=10
,arp,dl_src=b8:27:eb:9c:86:b7,arp_spa=192.168.4.100,arp_sha=dc:a6:32:0d:13:
44 actions=NORMAL
 cookie=0x0, duration=44.318s, table=0, n_packets=0, n_bytes=0, priority=4,
arp,arp_tpa=192.168.4.100 actions=CONTROLLER:65509
 cookie=0x0, duration=44.315s, table=0, n_packets=0, n_bytes=0, idle_timeou
t=60, send_flow_rem priority=3,arp,dl_src=b8:27:eb:9c:86:b7 actions=clear_a
ctions
```

Figure 6.4: Exp 1: A2.1: OvS OpenFlow Rules After Detection

Figure 6.6 shows a combined plot chart containing all the results for the *400* ARP packets, including the trend line as an average. The chart shows a clear similarity in detection time and the speed in which a malicious ARP packet is prevented. Individual results for each criteria can be seen in Appendix A.6.1, which contains the IDPS log for detecting the poisoned ARP and a plot chart containing all the results.

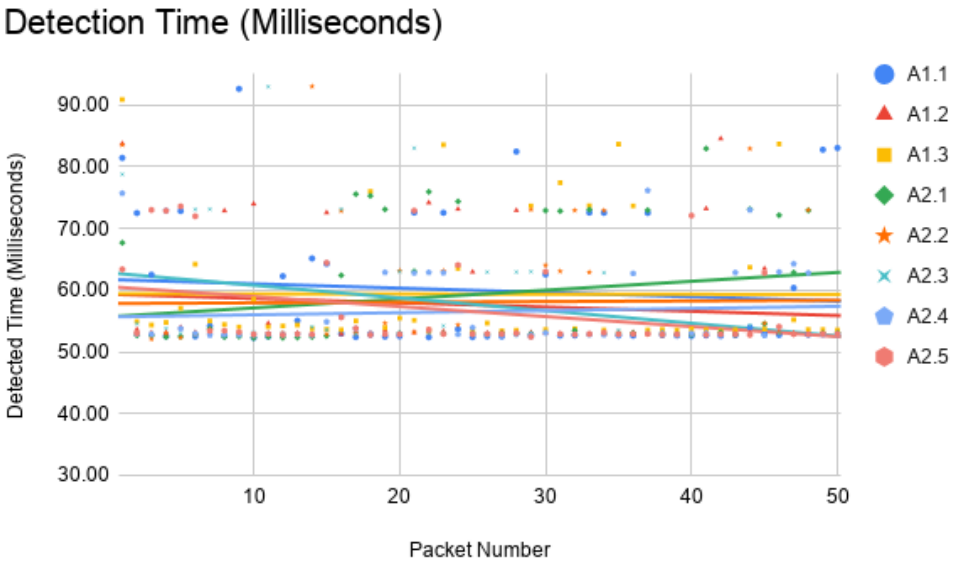

Figure 6.5: Exp 1: Summary of ARP Poisoned Detection Time (Millisecond)

6.3 Experiment 2: ARP RTT Time

As the proposed IDPS intends to have a minimal effect on the functionality and speed of ARP reply request packets, experiment 2 measured the impact the proposed system had on the ARP round trip time (RTT). As the experiment focused on ARP RTT time with the proposed IDPS functioning, the testbed was the same, with the exception that the attacker host was not actively ARP poisoning; instead of using regular ARP and measuring the RTT in different scenarios.

6.3.1 Experiment Setup

The experiment uses the custom testing tool implemented in Section 5.2.2 on *phy-host-two* host, acting as a Benign host within this experiment. There are four different scenarios an ARP request traverses the IDPS SDN. Therefore, the experiment is repeated in the following scenarios:

- **Traditional Network:** A base RTT set within a traditional network using the custom tool. Disabling the SDN controller, enables the OvS fail mode to *Standalone*; functioning as a non-SDN switch.
- **Existing Flow Rule:** The RTT through the proposed SDN system after the specific ARP flow rule has been installed by the IDPS.
- **New Flow Rule** Thee RTT of a new ARP request/reply encounter whereby the controller forwards ARP and flow rule being installed.
- **Proxy Server** The RTT when the proposed IDPS is acting as a proxy server and consistently replying for a blocked MAC address.

6.3.2 Measurements

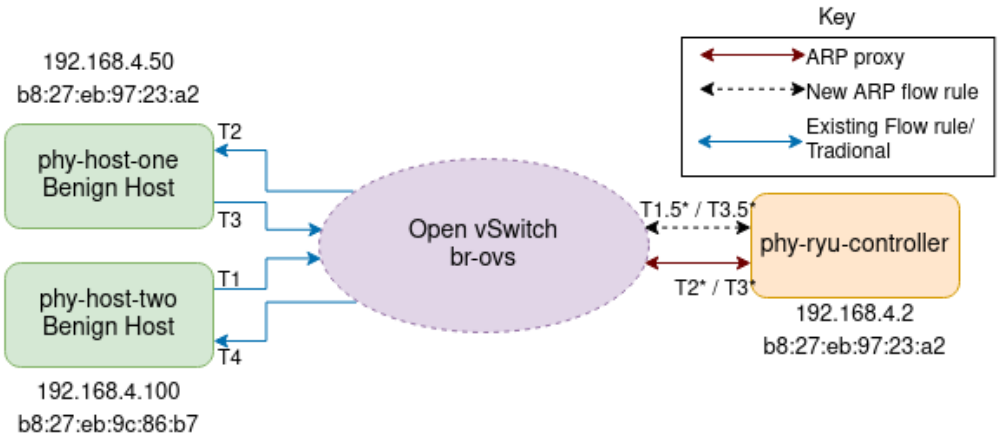

Figure 6.6: Exp 2: RTT Measurement Sampling Points

Figure 6.6 is the experiment setup and shows the time sampling points measuring the RTT. The RTT is the measurement of time for the responding host (*phy-host-two*) to receive a response to a ARP request. Therefore, the

initial time is when the ARP request was sent from the host and the RTT is when an ARP reply was received. In each scenario an ARP request is continuously sent every *10* seconds from *phy-host-two* to *phy-host-one*, this occurred for a total of fifty times. The tool was used within *phy-host-two*, therefore the sampling points used within the experiment are **T1** and **T4**, the figure also helps demonstrate the potential routes.

6.3.3 Results & Analysis

Table 6.4 contains the average, minimum and maximum RTT for the fifty ARP reply/request tests for each four scenarios. The bar charts in Figure 6.7 is the overall results for all *400* ARP reply/request RTT.

Scenario	Average RTT	Minimum RTT	Maximum RTT	Percent Increase from Base
Baseline	45.44ms	42.02ms	62.87ms	N/A
Existing Flow Fule	46.60ms	42.04ms	62.66ms	2.6%
New Flow Rule	96.52ms	74.5ms	128.82ms	112.5%
Proxy Server	125.94ms	92.9ms	167.2ms	212.5%

Table 6.4: Exp 2: Avg/Min/Max RTT in Each Scenario

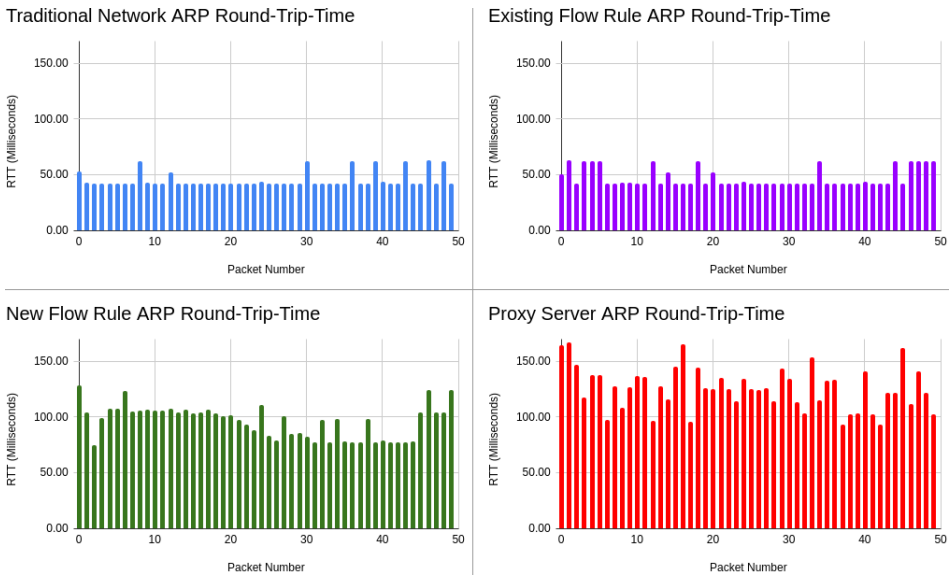

Figure 6.7: Exp 2: All Four ARP Round-Trip-Time Bar Charts, Avaiable in Appendix Section A.6.2.

The results suggest that the proposed system has minimal effect on the ARP RTT once a flow rule has been established, with the RTT *45.44ms* and *46.60ms* for Baseline and SDN flow rules respectively. This is due to the proposed system actively installing specific ARP flow rules for verified packets rather then re-verifying, which slows down all ARP packets.

This is seen in the table 6.4, for *New Flow Rule* scenario, as the ARP packet is forwarded to the controller, the RTT has increased due to waiting for IDPS verification. Figure 6.8 is a Wireshark capture of the communication between the switch and controller where OpenFlow sends a *PacketIn* event due to the new ARP packet, and the *Flow_mod* event to install the ARP flow rule.

	Source	Destination	Port	Protocol	Info
4	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_PACKET_IN
5	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_PACKET_OUT
6	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=111 Ack=109 Win=
7	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_FLOW_MOD
8	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=111 Ack=229 Win=
9	b8:27:eb:9c:86:b7	ff:ff:ff:ff:ff:ff		ARP	Who has 192.168.4.1? Tell 192.168.4.100
10	00:00:aa:bb:cc:dd	b8:27:eb:9c:86:b7		ARP	192.168.4.1 is at 00:00:aa:bb:cc:dd
11	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_PACKET_IN
12	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_PACKET_OUT
13	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_FLOW_MOD
14	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=195 Ack=431 Win=

Figure 6.8: Exp 2: Wireshark Adding New ARP Flow Rule

As the proposed system actively installs flow rules for ARP packets, the RTT of *96.52ms* is expected as the packet is added to the flow rules and verified. The next ARP packet sent travels with an *existing flow rule*, therefore, expected to be around *46.60ms* due to not triggering a *PacketIn* event and being forwarded by the switch as seen in Figure 6.9. Appendix A.17 contains the entire pcap file demonstrating the lack of *PacketIn* and OpenFlow communication once a flow rule is installed.

	Source	Destination	Port	Protocol	Info
15	b8:27:eb:9c:86:b7	ff:ff:ff:ff:ff:ff		ARP	Who has 192.168.4.1? Tell 192.168.4.100
16	00:00:aa:bb:cc:dd	b8:27:eb:9c:86:b7		ARP	192.168.4.1 is at 00:00:aa:bb:cc:dd

Figure 6.9: Exp 2: Wireshark Verified ARP Flow Rule Exists

The RTT for the proxy server is on average *125.94ms*, 212% slower than the baseline RTT. The results indicate a large overhead to RTT for ARP requests forwarded to the ARP proxy. The increased RTT is due to two factors. Firstly, the ARP reply and request packets have an extra path to traverse to reach the proposed IDPS, the extra forwarding slows down the ARP packet. Secondly, the Python3 library *scapy* was used to create and send the ARP reply rather than the intended ARP module service, *Scapy* pulls the ARP packet out of the Link-Layer to manage in Python which slows the response.

Figure 6.10 is a comparison line chart for the four scenarios that RTT was taken. Better illustrating the minimal affect the proposed system with a *2.6%* increase for RTT has compared to traditional networks. Furthermore, the Figure shows the sporadic RTT that was captured for the proxy server scenario. All RTT results are available in Appendix Section A.6.2.

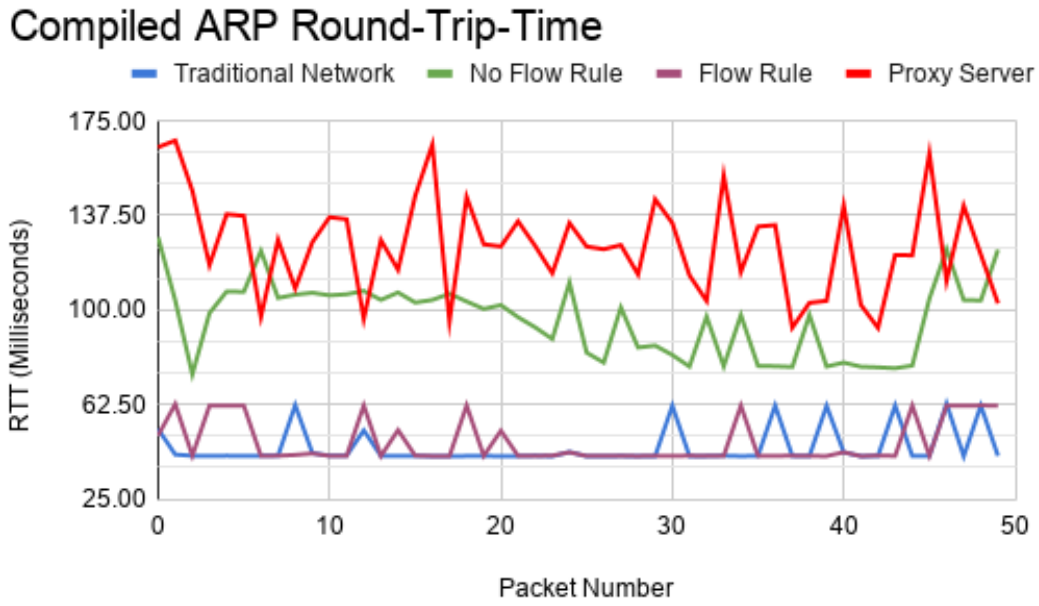

Figure 6.10: Exp 2: Full Round-Trip-Time Comparison Line Chart

6.4 Experiment 3: CPU Overhead

Experiment 3 was performed to evaluate the overhead imposed by the IDPS. The proposed IDPS aimed to reduce CPU overhead by actively installing flow rules, avoiding the constant CPU usage of validating ARP packets. The experiment measured the average CPU load of the controller and during several scenarios, this was performed on the implemented testbed containing a RaspberryPi4 as the SDN Ryu Controller.

6.4.1 Experiment Setup

Similar to the prior experiment relying on the custom testing tool to generate results, this round ran on the *phy-ryu-controller* instead. The tool gathers the CPU load for the Ryu Controller and IDPS system at regular intervals during three different scenarios:

- **Idle system:** The CPU load of the RPI4 running without the IDPS functioning, acting as a baseline. Therefore, no SDN traffic is being forwarded and IDPS not running on the controller.
- **IDPS system:** The CPU load of the IDPS functioning during normal traffic, normal ARP. This encapsulates DHCP server running and normal ARP messages being passed.
- **Proxy Server:** The CPU load as the ARP proxy is consistently replying for a blocked MAC address. This situation involves *phy-host-one* sending an ARP request to blocked MAC address of *phy-host-two*, every *1* second during testing.

6.4.2 Measurements

As stated this experiment gathers the RaspberryPi4 CPU load of the SDN controller, the testing tool calculates the average CPU load from the four CPU core's every *10* seconds for a total of *15* minutes.

6.4.3 Results & Analysis

The results displayed in Table 6.5 contain the Average, Minimum and Maximum CPU loads for each scenario. It is evident from the results that the CPU load during the functioning of the IDPS is negligible, increasing on average from *0.22%* to *0.3%*. These values show a success in the minimal resource load the IDPS consumes. However, the ARP proxy functionality has a significant impact on the CPU load, increasing the load on average by *24.3%*. The significant increase in CPU load during ARP proxy replies is likely due to the mix of running on a relatively low powered CPU, and the Python library *scapy* was used to handle the ARP replies.

Scenario	Average CPU load	Minimum CPU Load	Maximum CPU Load
Idle	0.22%	0.1%	1%
IDPS Running	0.3%	0.1%	1.7%
Proxy Server	24.5%	0.2%	25.9%

Table 6.5: Exp 3: Avg/Min/Max RTT in Each Scenario

Figure 6.11 are the results for each test reading per scenario, the stacked Figure 6.12 is a stepped chart that visually demonstrates the vast difference in CPU load between the scenario's Idle and IDPS to the proxy server. It is unlikely in the given testbed that the ARP proxy would be replying every second. However, these results indicate it is unlikely to expect the proposed IDPS ARP proxy to function in a large network.

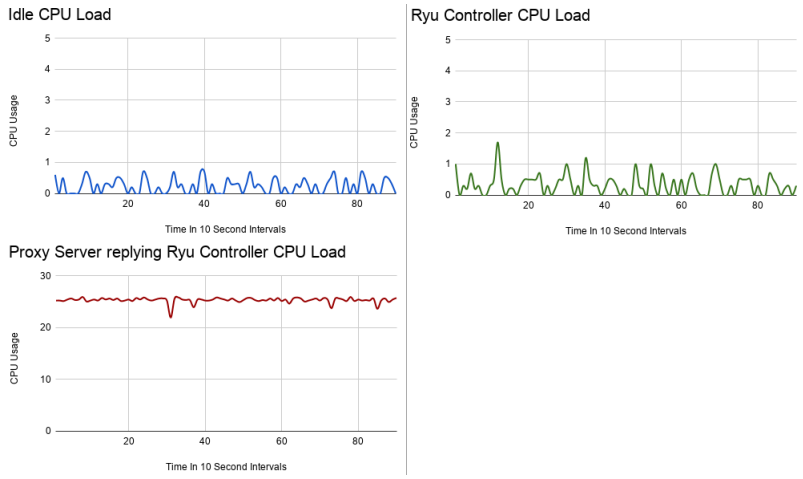

Figure 6.11: Exp 3: CPU Load Charts, Avaiable in Appendix A.6.3.

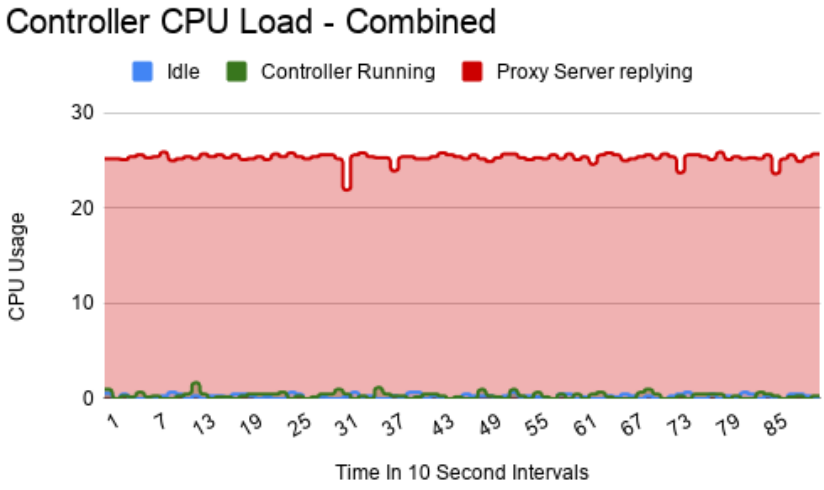

Figure 6.12: Exp 3: Combined CPU Load Stepped Area Chart

7 Conclusion

The proposed IDPS emphasises and utilises the flexible capabilities of the SDN paradigm, taking advantage of the ability to instantly drop packets and insert specifically crafted flow rules within OpenFlow 1.3. This report demonstrates the proposed innovative hybrid IDPS tested within a physical SDN network can detect and prevent a variety of poisoned ARP packets outlined in Table 4.2, without impeding on the network traffic RTT. Achieved by actively installing Link-Layer specific flow rules using the OpenFlow 1.3 protocol for forwarding verified ARP packets at traditional network speeds while also actively inserting flow rules which drop blocked MAC addresses that would otherwise impede the Controller. Furthermore, the developed IDPS could function autonomously, generating a MAC to IP pairing by utilising the DHCP server and checking all unknown ARP packets; preventing the need for manual updates to the network topology.

However, the hybrid IDPS had a significant impact on the CPU load of the controller, specifically the embedded ARP proxy server segment produced a high CPU load and increases the ARP RTT. This imposed on the SDN network, which meant partially failure in meeting the second objective of a minimal impact on the network and minimal CPU load.

A physical testbed was implemented and configured to specifically test the developed IDPS output to reflect it's use in a real-world environment. Several unique tools were developed specifically for configuring the developed IDPS and for testing the functionality within the testbed.

7.1 Limitations & Alterantives

The proposed IDPS has several limitations some of which were highlighted during the testing phase of the report, while others were not as apparent. These limitations are briefly listed and explained:

1. **Scapy:** The Python3 Library *scapy* is utilised to create ARP packets. This introduced a significant delay in receiving and sending ARP packets which was witnessed during RTT testing (Section 6.3.3). Optimisation methods were employed, yet failed to make any significant improvements.
2. **ARP Proxy:** This limitation ties into the previous, the ARP proxy produced a high CPU, load discussed in Section 6.4.3 and a large delay in RTT within Section 6.3.3. This was due to utilising *scapy* for ARP packets, which when running in Python3 are slower then low-level kernel modules that usually handle ARP packets. The solution would be to find an alternative library or create a specific ARP packet library, alternatively implementing the ARP proxy in another language which runs alongside the IDPS.

3. **Limited Hardware:** The hardware used to implement the SDN network is relatively low powered and lacked certain functionality, such as the router not supporting SDN by default and the Raspberry Pi's using ARM Cortex processors. Ideally, the proposed IDPS is tested within an SDN network using a typical networking infrastructure.
4. **CSV Files:** CSV files for maintaining the lists were used out of simplicity, yet the simplicity has some major downsides. Firstly, the CSV files require an order and every field needs to be populated, the workaround was using the second field as a Python3 Dictionary and storing the CSV files locally, therefore, only accessible by the controller and IDPS. Switching to a Structured Query Language (*SQL*) database would provide a better storage format, more flexibility as well as offering the advantage of being accessed by multiple SDN Controllers.

7.2 Future work

Future iterations of the proposed system would research beyond IPv4 ARP packets, as IPv4 is slowly being decremented for IPv6. ARP is decremented within the IPv6 address space, instead introduces the Neighbor Discovery Protocol (*NDP*) which brings many improvements over ARP. Future work would incorporate the NDP into the IDPS, bringing protection for IPv4 and IPv6 networks, supporting a range of networks designed to function with both protocols.

The DHCP server utilisation provided by the hybrid IDPS is a dynamic approach of generating a *mac_knownlist* and topology, however, further research could better utilise the DHCP server. For instance when a flow rule is inserted for a verified ARP packet, it's set with a static *idle_timeout* value; however by utilising the DHCP offer a *hard_timeout* could be implemented to the flow rule using the DHCP lease time. A verified flow rule could exist for the same amount of time the DHCP lease exists for, minimising the frequency of times the IDPS is required to verify MAC and IP pairings.

The ARP proxy primarily functions to prevent false-positive MAC blockings by adversaries which can effectively perform a DoS on key services. However, the performance demonstrated during testing in Section 6.3.3 and Section 6.4.3 demonstrated the partial failure of the second objective of minimal imposition on the network. Further developmen of this report would see a redesign and rewrite of the functionality such as, implementing an ARP packet replay feature, where the ARP proxy is not required to recreate an ARP reply every time; instead, creating an ARP reply once and replaying the packet as required.

Further testing would ideally be performed on the IDPS, as testing was focused on functionality and the system's footprint. However, further testing is required to measure the IDPS within a busy network to verify the effectiveness when faced with hundreds of ARP packets per second. This report does not test for possible false-positive MAC address blocking while proccessesng a large quantity of packets nor does it test the data throughput during these situations.

Bibliography

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, pp. 20–40, 2013.

[2] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 493–512, 2013.

[3] S. Higginbotham, "How google is using openflow to lower its network costs," Apr 2012. [Online]. Available: https://gigaom.com/2012/04/09/how-google-is-using-openflow-to-lower-its-network-costs/

[4] S. K. Tayyaba, M. A. Shah, N. S. A. Khan, Y. Asim, W. Naeem, and M. Kamran, "Software-defined networks (sdns) and internet of things (iots): A qualitative prediction for 2020," *network*, vol. 7, no. 11, 2016.

[5] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, 2012.

[6] S. Y. Nam, D. Kim, and J. Kim, "Enhanced arp: preventing arp poisoning-based man-in-the-middle attacks," *IEEE communications letters*, vol. 14, no. 2, pp. 187–189, 2010.

[7] CVE, "Cve - cve-1999-0667," 1999. [Online]. Available: https://www.cvedetails.com/cve/CVE-1999-0667/

[8] C. Doman, "Arp spoofing used to insert malicious adverts," Oct 2017. [Online]. Available: https://cybersecurity.att.com/blogs/labs-research/arp-spoofing-used-to-insert-malicious-adverts

[9] T. Alharbi, D. Durando, F. Pakzad, and M. Portmann, "Securing arp in software defined networks," in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*. IEEE, 2016, pp. 523–526.

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

[11] M. Z. Masoud, Y. Jaradat, and I. Jannoud, "On preventing arp poisoning attack utilizing software defined network (sdn) paradigm," in *2015 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*. IEEE, 2015, pp. 1–5.

[12] A. M. AbdelSalam, A. B. El-Sisi, and V. Reddy, "Mitigating arp spoofing attacks in software-defined networks," *in ICCTA, at alexandria, Egypt*, 2015.

[13] F. M. Ramos, D. Kreutz, and P. Verissimo, "Software-defined networks: On the road to the softwarization of networking," *Cutter IT journal*, 2015.

[14] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting edge of ip router configuration," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 21–26, 2004.

[15] P. Rakheja, P. Kaur, A. Gupta, and A. Sharma, "Performance analysis of rip, ospf, igrp and eigrp routing protocols in a network," *International Journal of Computer Applications*, vol. 48, no. 18, pp. 6–11, 2012.

[16] S. R. Rodriguez, "Topology discovery using cisco discovery protocol," *arXiv preprint arXiv:0907.2121*, 2009.

[17] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the complexity of network management." in *NSDI*, 2009, pp. 335–348.

[18] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, "A survey of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 2, pp. 7–23, 1999.

[19] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and control element separation (forces) framework," RFC 3746, April, Tech. Rep., 2004.

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

[21] Lanner, "how sdn is fixing existing network bottlenecks with more hardware at the edge," Jun 2019. [Online]. Available: https://www.lanner-america.com/blog/sdn-fixing-existing-network-bottlenecks-hardware-edge/

[22] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *International Conference on Passive and Active Network Measurement*. Springer, 2015, pp. 347–359.

[23] open networking foundation, "software-defined networking definition - open networking foundation 2020," 2020. [Online]. Available: https://www.opennetworking.org/sdn-definition/

[24] 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.

[25] M. Rouse, "What is software-defined networking," Aug 2019. [Online]. Available: https://searchnetworking.techtarget.com/definition/software-defined-networking-SDN

[26] Noxrepo, "noxrepo/pox," Jan 2012. [Online]. Available: https://github.com/noxrepo/pox

[27] Z. K. Khattak, M. Awais, and A. Iqbal, "Performance evaluation of opendaylight sdn controller," in *2014 20th IEEE international conference on parallel and distributed systems (ICPADS)*. IEEE, 2014, pp. 671–676.

[28] 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 *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 1–6.

[29] R. Community, "Ryu sdn framework." [Online]. Available: https://ryu-sdn.org/

[30] OpenFlow, "Openflow 1.3 specification." [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf

[31] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *ACM SIGCOMM computer communication review*, vol. 37, no. 4, pp. 1–12, 2007.

[32] "Open networking foundation," Aug 2020. [Online]. Available: https://www.opennetworking.org/

[33] H. Packard. [Online]. Available: https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c04089227

[34] V. Šulák, P. Helebrandt, and I. Kotuliak, "Performance analysis of openflow forwarders based on routing granularity in openflow 1.0 and 1.3," in *2016 19th Conference of Open Innovations Association (FRUCT)*. IEEE, 2016, pp. 236–241.

[35] "Multilayer open virtual switch." [Online]. Available: https://www.openvswitch.org/

[36] "Openwrt project." [Online]. Available: https://openwrt.org/docs/

[37] S. Hemminger, "iproute2." [Online]. Available: https://github.com/shemminger/iproute2

[38] M. Richardson, "Tcpdump/libpcap public repository." [Online]. Available: https://www.tcpdump.org/

[39] "Wireshark go deep." [Online]. Available: https://www.wireshark.org/

[40] A. Kuznetsov, "arping(8) - linux man page." [Online]. Available: https://linux.die.net/man/8/arping

[41] P. Biondi, "Scapy." [Online]. Available: https://scapy.net/

[42] G. Rodola, "Psutil." [Online]. Available: https://pypi.org/project/psutil/

[43] D. C. Plummer, "Rfc 826: An ethernet address resolution protocol, 1982," 2008.

[44] P. S. Kenkre, A. Pai, and L. Colaco, "Real time intrusion detection and prevention system," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Springer, 2015, pp. 405–411.

[45] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (idps)," National Institute of Standards and Technology, Tech. Rep., 2012.

[46] D. Samociuk, "Secure communication between openflow switches and controllers," *AFIN 2015*, vol. 39, 2015.

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

[48] C. Birkinshaw, E. Rouka, and V. G. Vassilakis, "Implementing an intrusion detection and prevention system using software-defined networking: Defending against port-scanning and denial-of-service attacks," *Journal of Network and Computer Applications*, vol. 136, pp. 71–85, 2019.

[49] "Cain and abel." [Online]. Available: https://sectools.org/tool/cain/

[50] M. Team, "Mininet." [Online]. Available: http://mininet.org/

[51] N. Tripathi and B. Mehtre, "Analysis of various arp poisoning mitigation techniques: A comparison," in *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. IEEE, 2014, pp. 125–132.

[52] "Openwrt project." [Online]. Available: https://openwrt.org/toh/tp-link/archer-c7-1750

[53] S. Kelley, "Dnsmasq." [Online]. Available: http://www.thekelleys.org.uk/dnsmasq/doc.html

[54] "Raspberry pi 4 model b." [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/

[55] "Raspberry pi 3 model b." [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

A Appendix

A.1 OpenFlow

A.1.1 OpenFlow 1.0 Matching table

Argument	Value	Description
in_port	Integer 16bit	Switch input port.
dl_src	MAC address	Ethernet source address.
dl_dst	MAC address	Ethernet destination address.
dl_vlan	Integer 16bit	Input VLAN id.
dl_vlan_pcp	Integer 8bit	Input VLAN priority.
dl_type	Integer 16bit	Ethernet frame type.
nw_tos	Integer 8bit	IP ToS (actually DSCP field, 6 bits).
nw_proto	Integer 8bit	IP protocol or lower 8 bits of ARP opcode.
nw_src	IPv4 address	IP source address.
nw_dst	IPv4 address	IP destination address.
tp_src	Integer 16bit	TCP/UDP source port.
tp_dst	Integer 16bit	TCP/UDP destination port.
nw_src_mask	Integer 8bit	IP source address mask specified as IPv4 address prefix.
nw_dst_mask	Integer 8bit	IP destination address mask specified as IPv4 address prefix.

Table A.1: OpenFlow 1.0 Matching table [29].

A.1.2 OpenFlow 1.3 Matching table

Argument	Value	Description
in_port	Integer 32bit	Switch input port
in_phy_port	Integer 32bit	Switch physical input port
metadata	Integer 64bit	Metadata passed between tables
eth_dst	MAC address	Ethernet destination address
eth_src	MAC address	Ethernet source address
eth_type	Integer 16bit	Ethernet frame type
vlan_vid	Integer 16bit	VLAN id
vlan_pcp	Integer 8bit	VLAN priority
ip_dscp	Integer 8bit	IP DSCP (6 bits in ToS field)
ip_ecn	Integer 8bit	IP ECN (2 bits in ToS field)
ip_proto	Integer 8bit	IP protocol
ipv4_src	IPv4 address	IPv4 source address
ipv4_dst	IPv4 address	IPv4 destination address

tcp_src	Integer 16bit	TCP source port
tcp_dst	Integer 16bit	TCP destination port
udp_src	Integer 16bit	UDP source port
udp_dst	Integer 16bit	UDP destination port
sctp_src	Integer 16bit	SCTP source port
sctp_dst	Integer 16bit	SCTP destination port
icmpv4_type	Integer 8bit	ICMP type
icmpv4_code	Integer 8bit	ICMP code
arp_op	Integer 16bit	ARP opcode
arp_spa	IPv4 address	ARP source IPv4 address
arp_tpa	IPv4 address	ARP target IPv4 address
arp_sha	MAC address	ARP source hardware address
arp_tha	MAC address	ARP target hardware address
ipv6_src	IPv6 address	IPv6 source address
ipv6_dst	IPv6 address	IPv6 destination address
ipv6_flabel	Integer 32bit	IPv6 Flow Label
icmpv6_type	Integer 8bit	ICMPv6 type
icmpv6_code	Integer 8bit	ICMPv6 code
ipv6_nd_target	IPv6 address	Target address for ND
ipv6_nd_sll	MAC address	Source Link-Layer for ND
ipv6_nd_tll	MAC address	Target Link-Layer for ND
mpls_label	Integer 32bit	MPLS label
mpls_tc	Integer 8bit	MPLS TC
mpls_bos	Integer 8bit	MPLS BoS bit
pbb_isid	Integer 24bit	PBB I-SID
tunnel_id	Integer 64bit	Logical Port Metadata
ipv6_exthdr	Integer 16bit	IPv6 Extension Header pseudo-field
pbb_uca	Integer 8bit	PBB UCA header field
tcp_flags	Integer 16bit	TCP flags (EXT-109 ONF Extension)
actset_output	Integer 32bit	Output port from action set metadata

Table A.2: OpenFlow 1.3 Matching table [29].

A.2 Ryu IDPS Code and Snippets

A.2.1 Ryu IDPS Code

```
import os.path
import csv # Store MAC Lists
from ast import literal_eval
import time
from datetime import datetime
import socket
import scapy.all as scapy
import netifaces as ni
from ryu.base import app_manager
from ryu.controller import ofp_event, dpset
from ryu.controller.handler import CONFIG_DISPATCHER,
    MAIN_DISPATCHER, HANDSHAKE_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet, ether_types, in_proto
from ryu.lib.packet import arp, dhcp, ipv4
from ryu.ofproto import ofproto_v1_3

#TODO Optimisation - Scapy slow ARP replies, re-parsing
    function calls
#TODO Add Better functionality to get topology of network E.G
     Scan function Give knownlist a Time lease like DHCP
#TODO Use DHCP lease time left to set how long flow rule
    exists for

#NOTES########
#TODO Hardcoded OVS IP subnet

MAC_KNOWNLIST = "./mac_knownlist.csv"
DHCP_WHITELIST = "./dhcp_whitelist.csv"
LOG_FILE = "./logs/log_{}.csv"

BLOCK_TIME = 60
ARP_ALLOW_TIME = 120

# Simple local function to convert MAC's into hostnames so i
    can read log easier
def convertMAC(mac):
    known_macs = {"b8:27:eb:97:23:a2": "Host-One",
                  "b8:27:eb:9c:86:b7": "Host-Two",
                  "dc:a6:32:0d:13:44": "Ryu-Controller",
                  "00:00:aa:bb:cc:dd": "OVSwitch",
                  "ff:ff:ff:ff:ff:ff": "Broadcast"}
    return known_macs.get(mac, mac)


class networkControl():
    def __init__(self):
        self.hosts = {}
        self.hostMAC = ni.ifaddresses('eth0')[ni.AF_LINK][0][
    "addr"]
        self.hostIP = ni.ifaddresses('eth0')[ni.AF_INET][0]["
```

```
    addr"]

    def networkScan(self, ip):
        self.ip = ip
        arp_r = scapy.ARP(pdst=ip)
        br = scapy.Ether(dst='ff:ff:ff:ff:ff:ff')
        request = br/arp_r
        answered, unanswered = scapy.srp(request, timeout=1)
        for i in answered:
            self.hosts[i[1].hwsrc] = i[1].psrc


class proxyServer():
    def __init__(self):
        self.hosts = {}
        self.proxyMAC =  ni.ifaddresses('eth0')[ni.AF_LINK
    ][0]["addr"]

    def exists(self, mac):
        if mac in self.hosts.keys():
            return True
        return False

    def existsByIP(self, IP):
        if IP in self.hosts.values():
            return True
        return False

    def retriveMacFromIP(self, IP):
        for MAC, itIP in self.hosts.items():
            if itIP == IP:
                return MAC
        return False

    def createArpReply(self, src_mac, dst_mac, dst_ip):
        scapy.sendp(scapy.Ether(dst=dst_mac, src=src_mac) /
                    scapy.ARP(op="is-at", hwsrc=self.proxyMAC
    , psrc=self.hosts[src_mac], hwdst=dst_mac, pdst=dst_ip),
                    iface="eth0", verbose=False)

class csvLogger():
    def __init__(self, csv_file):
        self.csv = csv_file.format(datetime.now().strftime("%
    Y%m%d"))
        if os.path.exists(self.csv) is False:
            with open(self.csv, 'w'): pass

    def add(self, attackID, mac, packet, action):
        with open(self.csv,'a', newline='') as fd:
            writer = csv.writer(fd)
            writer.writerow([datetime.now(), attackID, mac,
    packet, action])

class csvMACListTracker():
    def __init__(self, csv_file):
        self.csv = csv_file
```

```
98          self._lastModified = None
99          self.macDict = {} # Live running list (Avoid cosntant
     file reads)
100         self.macDictTemplate = {"Rules": False, "LastIP":
    None} # Template, python passes by reference
101
102         self._readCsv()
103
104     def add(self, MAC, MACDict = None):
105         if MACDict == None:
106             MACDict = self.macDictTemplate
107         # Either updates CSV file with MAC with new MACDict,
    or appends
108         # Updates local Dict of MACs
109         try:
110             if MAC not in self.macDict.keys():
111                 with open(self.csv,'a', newline='') as fd:
112                     writer = csv.writer(fd)
113                     writer.writerow([MAC, MACDict])
114             else:
115                 with open(self.csv,'w', newline='') as fd:
116                     writer = csv.writer(fd)
117                     for MACExisting, MACDictExisting in self.
    macDict.items():
118                         if MACExisting == MAC:
119                             writer.writerow([MAC, MACDict])
120                         else:
121                             writer.writerow([MACExisting,
    MACDictExisting])
122             self.macDict[MAC] = MACDict
123         except OSError as exc:
124             print(exc)
125
126     def exists(self, MAC):
127         self._readCsv()
128         if MAC in self.macDict.keys():
129             return True
130         return False
131
132     def retriveFromIP(self, IP):
133         for MAC, Dict in self.macDict.items():
134             if Dict["LastIP"] == IP:
135                 return MAC
136         return False
137
138     def _readCsv(self):
139         if os.path.exists(self.csv):
140             if self._lastModified is None or os.path.getmtime
    (self.csv) > self._lastModified:
141                 with open(self.csv,'r', newline='') as fd:
142                     macList = csv.reader(fd)
143                     self.macDict = dict(macList)
144
145                 for MAC, Dict in self.macDict.items():
146                     self.macDict[MAC] = literal_eval(Dict)
147
```

```
148                 self._lastModified = time.time()
149         else:
150             with open(self.csv, 'w'): pass
151 
152 
153 class L2ARPIDPS(app_manager.RyuApp):
154     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
155 
156     def __init__(self, *args, **kwargs):
157         super(L2ARPIDPS, self).__init__(*args, **kwargs)
158         self.datapath = {}
159         self.mac_to_port = {}
160         self.proxyserver = proxyServer()
161         self.mac_knownlist = csvMACListTracker(MAC_KNOWNLIST)
162         self.dhcp_list = csvMACListTracker(DHCP_WHITELIST)
163         self.threatLogger = csvLogger(LOG_FILE)
164         self.setInitialFlows = False
165 
166     def loggerStore(self, pktid, message, mac, packet, action
    , level="warn"):
167         if level == "warn":
168             self.logger.warning("%s", message)
169         elif level == "critical":
170             self.logger.critical("%s", message)
171         self.threatLogger.add(pktid, mac, packet, action)
172 
173     @set_ev_cls(dpset.EventDP, HANDSHAKE_DISPATCHER)
174     def connectionChange(self, ev):
175         datapath = ev.dp
176         connected = ev.enter
177 
178         if connected:
179             self.datapath = datapath
180         if connected and self.setInitialFlows is False:
181             # Scanning Current Network for Topology
182             self.logger.info("Scanning network")
183             network_Control = networkControl()
184             network_Control.networkScan('192.168.4.1/24')
185             for mac, ip in network_Control.hosts.items():
186                 MACDict = {"LastIP": ip, "Type": None}
187                 self.mac_knownlist.add(mac, MACDict = MACDict
    )
188                 self.logger.info("Lan Machines: MAC: %s\tIP:
    %s", mac, ip)
189 
190             # Adding Own IP + Mac based on Interface
191             MACDict = {"LastIP": network_Control.hostIP, "
    Type": "DHCP"}
192             self.mac_knownlist.add(network_Control.hostMAC,
    MACDict = MACDict)
193             self.logger.info("Lan Machines: MAC: %s\tIP: %s",
     network_Control.hostMAC, network_Control.hostIP)
194 
195             self.setInitialFlows = True
196 
197 
```

```
198     @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
    CONFIG_DISPATCHER)
199     def switch_features_handler(self, ev):
200         datapath = ev.msg.datapath
201         ofproto = datapath.ofproto
202         parser = datapath.ofproto_parser
203 
204         # ARP is always to be sent to controller
205         match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP)
206         actions = [parser.OFPActionOutput(ofproto.
    OFPP_CONTROLLER)]
207         self.add_flow(datapath, 1, match, actions)
208 
209         # DHCP is always to be sent to controller
210         match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_IP, ip_proto=in_proto.IPPROTO_UDP, udp_src=67)
211         actions = [parser.OFPActionOutput(ofproto.
    OFPP_CONTROLLER)]
212         self.add_flow(datapath, 1, match, actions)
213 
214         # install the table-miss flow entry, NORMAL uses
    traditional non-OpenFlow pipeline of the switch
215         match = parser.OFPMatch()
216         actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL
    ,
217                                           ofproto.
    OFPCML_NO_BUFFER)]
218         self.add_flow(datapath, 0, match, actions)
219 
220 
221     def handle_DHCP(self, msg):
222         datapath = msg.datapath
223 
224         pkt = packet.Packet(msg.data)
225         eth = pkt.get_protocols(ethernet.ethernet)[0]
226         dpid = datapath.id
227 
228         pkt_dhcp = pkt.get_protocol(dhcp.dhcp)
229         if pkt_dhcp.op == dhcp.DHCP_DISCOVER:
230             self.logger.info("%s | %s -> %s", "DHCP Discover"
    , convertMAC(eth.src), convertMAC(eth.dst))
231         elif pkt_dhcp.op == dhcp.DHCP_OFFER:
232             self.logger.info("%s | %s -> %s", "DHCP Offer",
    convertMAC(eth.src), convertMAC(eth.dst))
233 
234             if self.dhcp_list.exists(eth.src):
235                 self.logger.info("Valid DHCP Offer to %s : %s
    ", pkt_dhcp.chaddr, pkt_dhcp.yiaddr)
236                 self.logger.info("Adding %s : %s to %s",
    pkt_dhcp.yiaddr, pkt_dhcp.chaddr, self.mac_knownlist.csv)
237                 MACDict = {"LastIP": pkt_dhcp.yiaddr, "Type":
     "DHCP"}
238                 self.mac_knownlist.add(pkt_dhcp.chaddr,
    MACDict = MACDict)
239             else:
```

```
240                 self.loggerStore("DHCP".format(attackID), "
    Invalid DHCP offer | MAC {} -> {}".format(
241                                 "DHCP", eth.src, eth.dst), eth.
    src, pkt_dhcp, "MAC block for {}s".format(BLOCK_TIME))
242                 match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_IP, eth_src=eth.src)
243                 self.add_flow(datapath, 3, match, [], drop=
    True)
244                 return
245         else:
246             self.logger.info("DHCP: %s | %s -> %s", dpid,
    convertMAC(eth.src), convertMAC(eth.dst))
247 
248         self.send_packet(msg)
249 
250 
251     def handle_ARP(self, msg):
252         datapath = msg.datapath
253         ofproto = datapath.ofproto
254         parser = datapath.ofproto_parser
255         pkt = packet.Packet(msg.data)
256         eth = pkt.get_protocols(ethernet.ethernet)[0]
257         dpid = datapath.id
258         pkt_arp = pkt.get_protocol(arp.arp)
259         pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
260 
261         # Proxy Server
262         if pkt_arp.opcode == arp.ARP_REQUEST and self.
    proxyserver.existsByIP(pkt_arp.dst_ip):
263             self.proxyserver.createArpReply(self.proxyserver.
    retriveMacFromIP(pkt_arp.dst_ip), eth.src, pkt_arp.src_ip)
264             return
265 
266         if self.mac_knownlist.exists(eth.src) is False and
    self.mac_knownlist.retriveFromIP(pkt_arp.src_ip) is False:
267             self.logger.info("Adding new host %s to known
    list: %s to %s", eth.src, pkt_arp.src_ip, self.
    mac_knownlist.csv)
268             MACDict = {"LastIP": pkt_ipv4.src, "Type": "
    Static"}
269             self.mac_knownlist.add(eth.src, MACDict)
270 
271 
272         if pkt_arp.opcode == arp.ARP_REQUEST:
273             self.logger.info("%s in %s | %s -> %s | %s", "ARP
     Request", dpid, convertMAC(eth.src), pkt_arp.dst_ip,
    convertMAC(eth.dst))
274             attackID = "1."
275         elif pkt_arp.opcode == arp.ARP_REPLY:
276             self.logger.info("%s for %s | %s -> %s", "ARP
    Reply",  pkt_arp.src_ip, convertMAC(eth.src), convertMAC(
    eth.dst))
277             attackID = "2."
278 
279 
280         if self.mac_knownlist.exists(eth.src) and self.
```

```
    mac_knownlist.macDict[eth.src]["LastIP"] != pkt_arp.src_ip
    :
281             self.loggerStore("{}1".format(attackID), "ARPID:
    ({}1) ARP spoofed | MAC {} -> {} | for IP: {}".format(
282                             attackID, eth.src, self.
    mac_knownlist.retriveFromIP(pkt_arp.src_ip), pkt_arp.
    src_ip),
283                             eth.src, pkt, "MAC block for {}s
    ".format(BLOCK_TIME))
284         elif pkt_arp.opcode == arp.ARP_REQUEST and self.
    mac_knownlist.retriveFromIP(pkt_arp.src_ip) and self.
    mac_knownlist.exists(eth.src) is False:
285             self.loggerStore("{}2".format(attackID), "ARPID:
    ({}2) ARP spoofed IP from unknown MAC | MAC {} | for IP:
    {}".format(
286                             attackID, eth.src, pkt_arp.
    src_ip), eth.src, pkt, "MAC block for {}s".format(
    BLOCK_TIME))
287         elif pkt_arp.opcode == arp.ARP_REPLY and self.
    mac_knownlist.retriveFromIP(pkt_arp.dst_ip) != pkt_arp.
    dst_mac and eth.dst != "ff:ff:ff:ff:ff:ff":
288             self.loggerStore("{}2".format(attackID), "ARPID:
    ({}2) ARP spoofed IP | MAC {} |  IP: {}".format(
289                             attackID, pkt_arp.src_mac,
    pkt_arp.src_ip), eth.src, pkt, "MAC block for {}s".format(
    BLOCK_TIME))
290         elif eth.src != pkt_arp.src_mac:
291             self.loggerStore("{}3".format(attackID), "ARPID:
    ({}3) ARP source miss-match | MAC {} <-> {} | from IP: {}"
    .format(
292                             attackID, eth.src, pkt_arp.
    src_mac, pkt_arp.src_ip), eth.src, pkt, "MAC block for {}s
    ".format(BLOCK_TIME))
293         elif eth.dst != pkt_arp.dst_mac and pkt_arp.opcode ==
     arp.ARP_REPLY:
294             self.loggerStore("{}4".format(attackID), "ARPID:
    ({}4) ARP destination miss-match | MAC {} <-> {} | from IP
    : {}".format(
295                             attackID, eth.src, pkt_arp.
    src_mac, pkt_arp.src_ip), eth.src, pkt, "MAC block for {}s
    ".format(BLOCK_TIME))
296         elif eth.dst.upper() == "FF:FF:FF:FF:FF:FF" and
    pkt_arp.opcode == arp.ARP_REPLY:
297             self.loggerStore("{}5".format(attackID), "ARPID:
    ({}5) ARP ether dst is FF:FF:FF:FF:FF:FF | MAC {} <-> {} |
     from IP: {}".format(
298                             attackID, eth.dst, pkt_arp.
    dst_mac, pkt_arp.src_ip), eth.src, pkt, "MAC block for {}s
    ".format(BLOCK_TIME))
299         else:
300             self.logger.info("Adding ARP flow rule for %ss:\n
    \tMAC: %s\tIP: %s ->\n\tMAC: %s\tIP: %s", ARP_ALLOW_TIME,
    eth.src, pkt_arp.src_ip, eth.dst, pkt_arp.dst_ip)
301             match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP, arp_sha=eth.src, arp_spa=pkt_arp.src_ip,
    eth_dst=eth.dst, eth_src=eth.src)
```

```
302             actions = [parser.OFPActionOutput(ofproto.
    OFPP_NORMAL)]
303             self.send_packet(msg)
304             self.add_flow(datapath, 2, match, actions,
    idle_timeout=ARP_ALLOW_TIME)
305             return
306 
307         self.logger.warning("Temporarily blocking MAC: %s,
    idle timeout %ss", eth.src, BLOCK_TIME)
308         if  self.mac_knownlist.exists(eth.src):
309             self.logger.info("Acting as proxy server in
    meantime for %s ", self.mac_knownlist.macDict[eth.src]["
    LastIP"])
310             # Adds flow to direct ARP packets destined for IP
     of offending MAC to controller for Proxy
311             match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP, arp_tpa=self.mac_knownlist.macDict[eth.src][
    "LastIP"])
312             actions = [parser.OFPActionOutput(ofproto.
    OFPP_CONTROLLER)]
313             self.add_flow(datapath, 4, match, actions)
314             # Proxy ARP is always safe
315             match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP, arp_sha=self.proxyserver.proxyMAC, eth_src=
    eth.src, arp_spa=self.mac_knownlist.macDict[eth.src]["
    LastIP"])
316             actions = [parser.OFPActionOutput(ofproto.
    OFPP_NORMAL)]
317             self.add_flow(datapath, 10, match, actions)
318 
319             self.proxyserver.hosts[eth.src] = self.
    mac_knownlist.macDict[eth.src]["LastIP"]
320         else:
321             self.logger.info("Cannot act as proxy server for
    unknown MAC: %s ", eth.src)
322 
323         # Blocks Offending MAC address from sending ARP
    messages
324         match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP, eth_src=eth.src)
325         self.add_flow(datapath, 3, match, [], drop=True,
    idle_timeout=BLOCK_TIME, flag=ofproto.OFPFF_SEND_FLOW_REM)
326 
327 
328 
329     def add_flow(self, datapath, priority, match, actions,
    buffer_id=None, drop=False, idle_timeout=0, hard_timeout
    =0, flag=0):
330         ofproto = datapath.ofproto
331         parser = datapath.ofproto_parser
332 
333         if drop:
334             inst = [parser.OFPInstructionActions(ofproto.
    OFPIT_CLEAR_ACTIONS, [])]
335         else:
336             inst = [parser.OFPInstructionActions(ofproto.
```

```
337 OFPIT_APPLY_ACTIONS, actions)]
338
339         if buffer_id:
340             mod = parser.OFPFlowMod(datapath=datapath,
    buffer_id=buffer_id,
341                                     priority=priority, match=
    match,
342                                     instructions=inst,
    idle_timeout=idle_timeout,
343                                     hard_timeout=hard_timeout
    , flags=flag)
344         else:
345             mod = parser.OFPFlowMod(datapath=datapath,
    priority=priority,
346                                     match=match, instructions
    =inst,
347                                     idle_timeout=idle_timeout
    , hard_timeout=hard_timeout,
348                                     flags=flag)
349         datapath.send_msg(mod)
350
351     def drop_flow(self, datapath, priority, match):
352         ofproto = datapath.ofproto
353         parser = datapath.ofproto_parser
354
355         mod = parser.OFPFlowMod(datapath=datapath, priority=
    priority,
356                                 match=match, command =
    ofproto.OFPFC_DELETE, out_port = ofproto.OFPP_ANY,
357                                 out_group=ofproto.OFPG_ANY)
358         datapath.send_msg(mod)
359
360
361     def send_packet(self, msg):
362         datapath = msg.datapath
363         ofproto = datapath.ofproto
364         parser = datapath.ofproto_parser
365
366         pkt = packet.Packet(msg.data)
367         eth = pkt.get_protocols(ethernet.ethernet)[0]
368
369         # learn a mac address and port to avoid FLOOD next
    time.
370         dpid = format(datapath.id, "d").zfill(16)
371         self.mac_to_port.setdefault(dpid, {})
372         self.mac_to_port[dpid][eth.src] = msg.match["in_port"
    ]
373
374         # If dst known port, specify output, else flood
375         if eth.dst in self.mac_to_port[dpid]:
376             out_port = self.mac_to_port[dpid][eth.dst]
377         else:
378             out_port = ofproto.OFPP_FLOOD
379         actions = [parser.OFPActionOutput(out_port)]
380
```

```
381         data = None
382         if msg.buffer_id == ofproto.OFP_NO_BUFFER:
383             data = msg.data
384 
385         out = parser.OFPPacketOut(datapath=datapath,
    buffer_id=msg.buffer_id,
386                                   in_port=msg.match["in_port"
    ], actions=actions, data=data)
387         datapath.send_msg(out)
388 
389     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
390     def packet_in_handler(self, ev):
391         msg = ev.msg
392         datapath = msg.datapath
393 
394         pkt = packet.Packet(msg.data)
395         eth = pkt.get_protocols(ethernet.ethernet)[0]
396 
397         # Ignore LLDP and OpenFlow Local port
398         if eth.ethertype == ether_types.ETH_TYPE_LLDP:
399             return
400 
401         dpid = datapath.id
402 
403         if pkt.get_protocol(arp.arp):
404             self.handle_ARP(msg)
405         elif pkt.get_protocol(dhcp.dhcp):
406             self.handle_DHCP(msg)
407         else:
408             # Should not be triggered, the packet is
    forwarded to avoid potential disaster
409             pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
410             if eth.src not in self.mac_knownlist.macDict.keys
    ():
411                 self.logger.info("Adding %s : %s to %s", eth.
    src, pkt_ipv4.src, self.mac_knownlist.csv)
412                 MACDict = {"LastIP": pkt_ipv4.src, "Type": "
    Static"}
413                 self.mac_knownlist.add(eth.src, MACDict)
414             self.logger.info("%s in %s | %s -> %s", eth.
    ethertype, dpid, eth.src, eth.dst)
415             self.send_packet(msg)
416 
417 
418     @set_ev_cls(ofp_event.EventOFPFlowRemoved,
    MAIN_DISPATCHER)
419     def flow_removed_handler(self, ev):
420         msg = ev.msg
421         datapath = msg.datapath
422         parser = datapath.ofproto_parser
423         ofproto = datapath.ofproto
424 
425         if msg.reason == ofproto.OFPRR_IDLE_TIMEOUT:
426             self.logger.info("Flow Timed out")
427             macAddress = msg.match["eth_src"]
428 
```

```
429             self.logger.info('OFPFlowRemoved received:\n'
430                      '\tduration_sec=%d'
431                      '\tidle_timeout=%d hard_timeout=%d\n'
432                      '\tpacket_count=%d\n\tmatch.fields=%s',
433                      msg.duration_sec,
434                      msg.idle_timeout, msg.hard_timeout,
435                      msg.packet_count, msg.match)
436 
437             if self.proxyserver.exists(macAddress):
438                 self.logger.info("Dropping Proxy server flow
    for %s", self.proxyserver.hosts[macAddress])
439                 self.logger.info("\tFlow rule prevented %s
    packets", msg.packet_count)
440                 # Delete Block MAC flow
441                 match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP, arp_tpa=self.proxyserver.hosts[macAddress])
442                 self.drop_flow(datapath, 4, match)
443                 # Delete allow Proxy ARP flow
444                 match = parser.OFPMatch(eth_type=ether_types.
    ETH_TYPE_ARP, arp_sha=self.proxyserver.proxyMAC, eth_src=
    macAddress, arp_spa=self.proxyserver.hosts[macAddress])
445                 self.drop_flow(datapath, 10, match)
446                 del self.proxyserver.hosts[macAddress]
447 
448 
449     @set_ev_cls(ofp_event.EventOFPErrorMsg,
450             [HANDSHAKE_DISPATCHER, CONFIG_DISPATCHER,
    MAIN_DISPATCHER])
451     def error_msg_handler(self, ev):
452         msg = ev.msg
453         self.logger.info('OFPErrorMsg received: type=0x%02x
    code=0x%02x '
454                          'message=%s',
455                          msg.type, msg.code, utils.hex_array
    (msg.data))
```

Code Snippet A.1: Ryu Controller IDPS Code Python3

A.2.2 IDPS Tool Code

```
1 #!/usr/bin/python3
2 import os.path
3 import csv # Store Mac Lists
4 import argparse
5 from ARP_IDPS import csvMACListTracker as csvList
6 
7 def parseArguments():
8     parser = argparse.ArgumentParser(description='Simple tool
      to Manage ARP IDPS')
9     parser.add_argument('file', type=str, nargs=1, help='CSV
    file to use')
10     parser.add_argument('-l', dest="list", action="store_true
    ", help='List CSV file')
11     parser.add_argument('-a', dest="add", type=str, nargs=1,
    help='Add Mac to CSV')
12     parser.add_argument('-d', dest="delete", type=str, nargs
    =1, help='Delete row Containg MAC to CSV')
```

```
    parser.add_argument('-f', dest="flush", action="store_true", help='Flush entire CSV file')

    return parser.parse_args()

def main(args):
    if args.list:
        csvTrack = csvList(args.file[0])
        for Mac, MacDict in csvTrack.macDict.items():
            print("MAC: {}\tDict: {}".format(Mac, MacDict))
    elif args.add:
        csvTrack = csvList(args.file[0])
        csvTrack.add(args.add[0])
        print("Added {} to {}".format(args.add[0], args.file[0]))
    elif args.delete:
        csvTrack = csvList(args.file[0])
        if csvTrack.exists(args.delete[0]):
            with open(args.file[0],'w', newline='') as fd:
                writer = csv.writer(fd)
                for Mac, MacDict in csvTrack.macDict.items():
                    if Mac == args.delete[0]:
                        continue
                    writer.writerow([Mac, MacDict])
            print("Deleted")
        else:
            print("Could not find entry for: {}".format(args.delete[0]))
    elif args.flush:
        open(args.file[0], 'w').close()



if __name__ == "__main__":
    main(parseArguments())
```

Code Snippet A.2: IDPS Tool Code Python3

A.2.3 MAC Knownlist Example

```
00:00:aa:bb:cc:dd,"{'LastIP': '192.168.4.1', 'Type': None}"
b8:27:eb:9c:86:b7,"{'LastIP': '192.168.4.106', 'Type': None}"
b8:27:eb:97:23:a2,"{'LastIP': '192.168.4.178', 'Type': None}"
dc:a6:32:0d:13:44,"{'LastIP': '192.168.4.2', 'Type': 'DHCP'}"
```

Code Snippet A.3: mac_knownlist.csv example

A.2.4 DHCP whitelist Example

```
00:00:aa:bb:cc:dd,"{'LastIP': '192.168.4.1', 'Type': 'Static'}"
```

Code Snippet A.4: dhcp_whitelist.csv example

A.2.5 IDPS log Example

```
2020-08-20 14:44:12.937436,1.1,b8:27:eb:9c:86:b7,"ethernet(
    dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='b8:27:eb:9c
    :86:b7'), arp(dst_ip='192.168.4.178',dst_mac='ff:ff:ff:ff:
    ff:ff',hlen=6,hwtype=1,opcode=1,plen=4,proto=2048,src_ip
    ='192.168.4.1',src_mac='b8:27:eb:9c:86:b7')",MAC block for
     60s
```

Code Snippet A.5: IDPS log example

A.3 TP-Link Archer AC1750 C7 Setup

A.3.1 Network IF config

```
config interface 'loopback'
  option ifname 'lo'
  option proto 'static'
  option ipaddr '127.0.0.1'
  option netmask '255.0.0.0'

config globals 'globals'
  option ula_prefix 'fda7:0f01:b5c2::/48'

config interface 'lan'
  option type 'bridge'
  option ifname 'eth1'
  option force_link '1'
  option proto 'static'
  option ipaddr '192.168.3.1'
  option netmask '255.255.255.0'
  option ip6assign '60'

config interface 'wan'
  option ifname 'eth0.2'
  option proto 'dhcp'

config interface 'wan6'
  option ifname 'eth0.2'
  option proto 'dhcpv6'

config switch
  option name 'switch0'
  option reset '1'
  option enable_vlan '1'
  option enable_learning '0'

config switch_vlan
  option device 'switch0'
  option vlan '3'
  option ports '2 0t'

config switch_vlan
  option device 'switch0'
  option vlan '4'
  option ports '3 0t'

config switch_vlan
```

```
  option device 'switch0'
  option vlan '5'
  option ports '4 0t'

config switch_vlan
  option device 'switch0'
  option vlan '6'
  option ports '5 0t'

config switch_vlan
  option device 'switch0'
  option vlan '2'
  option ports '1 6t'

config interface
  option ifname 'eth1.3'
  option proto 'static'

config interface
  option ifname 'eth1.4'
  option proto 'static'

config interface
  option ifname 'eth1.5'
  option proto 'static'

config interface
  option ifname 'eth1.6'
  option proto 'static'
```

Code Snippet A.6: OpenWRT interface config file

A.3.2 dnsmasq Configuration File

```
dhcp-range=192.168.4.50,192.168.4.200,8h
dhcp-host=d0:50:99:82:e7:2b,192.168.4.2 #Controller
dhcp-host=b8:27:eb:97:23:a2,192.168.4.50 #Host one
dhcp-host=b8:27:eb:9c:86:b7,192.168.4.100 #Host Two
```

Code Snippet A.7: OpenWRT dnsmasq config file

A.3.3 Open vSwitch Configruation Setup

```
# Create OVSwitch Bridge using the 4 VLANS
$ ovs-vsctl add-br br-ovs -- set bridge br-ovs protocols=
    OpenFlow13 -- set-controller br-ovs tcp:192.168.4.2:6633
    -- set-fail-mode br-ovs standalone -- add-port br-ovs eth1
    .3 -- add-port br-ovs eth1.4 -- add-port br-ovs eth1.5 --
    add-port br-ovs eth1.6
# Set Bridge IP address
$ ip a a 192.168.4.1/24 dev br-ovs
```

Code Snippet A.8: OvS setup commands

A.3.4 TP-Link Archer dnsmasq/arp clear

```
#!/bin/sh
```

```
# Remove leases
rm /tmp/dhcp.leases
# Delete ARP table (OpenWRT method)
rm /proc/net/arp
# Clear IP neigh ARP table
ip neigh delete 192.168.4.* dev br-ovs
#Restart DHCP server
/etc/init.d/dnsmasq restart
```

Code Snippet A.9: OpenWRT DHCP clear Script

A.4 NTP Configs

A.4.1 Client

```
# /etc/ntp.conf, configuration for ntpd; see ntp.conf(5) for
    help

driftfile /var/lib/ntp/ntp.drift

# Leap seconds definition provided by tzdata
leapfile /usr/share/zoneinfo/leap-seconds.list

statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable


# pick a different set every time it starts up.  Please
    consider joining the
server 192.168.4.2

# By default, exchange time with everybody, but don't allow
    configuration.
restrict -4 default kod notrap nomodify nopeer noquery
    limited
restrict -6 default kod notrap nomodify nopeer noquery
    limited

# Local users may interrogate the ntp server more closely.
restrict 127.0.0.1
restrict ::1
```

Code Snippet A.10: Client ntp Config file

A.4.2 Server

```
# /etc/ntp.conf, configuration for ntpd; see ntp.conf(5) for
    help
driftfile /var/lib/ntp/ntp.drift

# Leap seconds definition provided by tzdata
leapfile /usr/share/zoneinfo/leap-seconds.list

# Enable this if you want statistics to be logged.
#statsdir /var/log/ntpstats/
```

```
9
10 statistics loopstats peerstats clockstats
11 filegen loopstats file loopstats type day enable
12 filegen peerstats file peerstats type day enable
13 filegen clockstats file clockstats type day enable
14
15 # pool.ntp.org maps to about 1000 low-stratum NTP servers.
      Your server will
16 # pick a different set every time it starts up.  Please
      consider joining the
17 # pool: <http://www.pool.ntp.org/join.html>
18 pool 0.debian.pool.ntp.org iburst
19 pool 1.debian.pool.ntp.org iburst
20
21
22 # By default, exchange time with everybody, but don't allow
      configuration.
23 restrict -4 default kod notrap nomodify nopeer noquery
      limited
24 restrict -6 default kod notrap nomodify nopeer noquery
      limited
25
26 # Local users may interrogate the ntp server more closely.
27 restrict 127.0.0.1
28 restrict 192.168.4.0 mask 255.255.255.0
29 restrict ::1
30
31 # Needed for adding pool entries
32 restrict source notrap nomodify noquery
33
34 # If you want to provide time to your local subnet, change
      the next line.
35 # (Again, the address is an example only.)
36 broadcast 192.168.4.255
37 broadcast 224.0.1.1
```

Code Snippet A.11: Server ntp Config file

A.5 Testing Tools / Code

A.5.1 IDPS Testing Tool Code

```
1 #!/usr/bin/python3
2 import os.path
3 import csv # Store Results
4 import psutil
5 import subprocess
6 import time
7 from datetime import datetime
8 import socket
9 import scapy.all as scapy
10 import netifaces as ni
11 import argparse
12
13
14 def parseArguments():
```

```
    parser = argparse.ArgumentParser(description='Simple tool to Test ARP IDPS')
    parser.add_argument('-p', dest="pois", type=str, choices=['1.1','1.2','1.3','2.1','2.2','2.3','2.4','2.5','rtt','cpu'], help='Spoof type by ID')
    parser.add_argument('-m', dest="mode", type=str, choices=['s', 'c', 'r'], help='Used for Reply attack testing, One device as (s)erver and other device as (c)lient, or (r)ply only')
    parser.add_argument('-t', dest="time", type=int, default=60, help='Time between tests')
    parser.add_argument('-c', dest="count", type=int, default=100, help='Times to test')
    parser.add_argument('-o', dest="output", type=str, help='Extension text to output file')

    return parser.parse_args()


def writeIntoCSV(filen, data):
    filen = filen+".csv"
    print("Writing results into {}".format(filen))
    with open(filen,'w', newline='') as fd:
        writer = csv.writer(fd)
        for row in data:
            writer.writerow(row)


def main(args):
    if args.pois:
        filename = "TestID_{}".format(args.pois)
        if args.output:
            filename = filename + "_{}".format(args.output)
        if args.pois == "1.1":
            data = []
            msg = scapy.Ether(dst="ff:ff:ff:ff:ff:ff", src=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy.ARP(op="who-has", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"], psrc="192.168.4.1", hwdst="ff:ff:ff:ff:ff:ff", pdst="192.168.4.50")
            for i in range(args.count):
                print("Test number {}..".format(i))
                test = [i,datetime.now()]
                answered,unanswered = scapy.srp(msg, verbose=False, timeout=1)
                if len(answered) > 0:
                    test.append("1")
                elif len(unanswered) > 0:
                    test.append("0")
                data.append(test)
                time.sleep(args.time)
            writeIntoCSV(filename,data)

        elif args.pois == "1.2":
            print("Ensure this machines MAC is unknown in IDPS")
```

```
56              data = []
57              msg = scapy.Ether(dst="ff:ff:ff:ff:ff:ff", src=ni
    .ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy.ARP(
    op="who-has", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK][0][
    "addr"], psrc="192.168.4.1", hwdst="ff:ff:ff:ff:ff:ff",
    pdst="192.168.4.50")
58              for i in range(args.count):
59                  print("Test number {}..".format(i))
60                  test = [i,datetime.now()]
61                  answered,unanswered = scapy.srp(msg, verbose=
    False, timeout=1)
62                  if len(answered) > 0:
63                      test.append("1")
64                  elif len(unanswered) > 0:
65                      test.append("0")
66                  data.append(test)
67                  time.sleep(args.time)
68              writeIntoCSV(filename,data)
69
70          elif args.pois == "1.3":
71              data = []
72              msg = scapy.Ether(dst="ff:ff:ff:ff:ff:ff", src=ni
    .ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy.ARP(
    op="who-has", hwsrc="00:00:00:01:02:03", psrc="
    192.168.4.100", hwdst="ff:ff:ff:ff:ff:ff", pdst="
    192.168.4.1")
73              for i in range(args.count):
74                  print("Test number {}..".format(i))
75                  test = [i,datetime.now()]
76                  answered,unanswered = scapy.srp(msg, verbose=
    False, timeout=1)
77                  if len(answered) > 0:
78                      test.append("1")
79                  elif len(unanswered) > 0:
80                      test.append("0")
81                  data.append(test)
82                  time.sleep(args.time)
83              writeIntoCSV(filename,data)
84
85          elif args.pois == "2.1":
86              if args.mode == "c":
87                  def respond_func(pkt):
88                      if pkt.pdst == ni.ifaddresses('eth0')[ni.
    AF_INET][0]["addr"] and pkt[scapy.ARP].op == 1:
89                          # create arp reply paket
90                          msg = scapy.Ether(dst=pkt.hwsrc, src=
    ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy.ARP
    (op="is-at", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK][0]["
    addr"], psrc="192.168.4.1", hwdst=pkt.hwsrc, pdst=pkt.psrc
    )
91                          scapy.srp(msg,verbose=False, timeout
    =1)
92                  print("Starting Sniffing...")
93                  scapy.sniff(iface="eth0", prn=respond_func,
    filter='arp and host 192.168.4.100', store=0)
94                  print("Stopped Sniffing...")
```

```
95              elif args.mode == "s":
96                  msg = scapy.Ether(dst="ff:ff:ff:ff:ff:ff",
    src=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy
    .ARP(op="who-has", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK
    ][0]["addr"], psrc=ni.ifaddresses('eth0')[ni.AF_INET][0]["
    addr"], hwdst="ff:ff:ff:ff:ff:ff", pdst="192.168.4.100")
97                  for i in range(args.count):
98                      print("Test number {}..".format(i))
99                      answered,unanswered = scapy.srp(msg,
    verbose=False, timeout=1)
100                     if len(answered) > 0:
101                         print("Responded")
102                     elif len(unanswered) > 0:
103                         print("Fuck")
104                     time.sleep(args.time)
105             elif args.mode == "r":
106                 #Cheat mode just send "is-at" a reply to no
    request
107                 data = []
108                 msg = scapy.Ether(dst="b8:27:eb:97:23:a2",
    src=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy
    .ARP(op="is-at", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK
    ][0]["addr"], psrc="192.168.4.1", hwdst="b8:27:eb:97:23:a2
    ", pdst="192.168.4.50")
109                 for i in range(args.count):
110                     print("Test number {}..".format(i))
111                     test = [i,datetime.now()]
112                     answered,unanswered = scapy.srp(msg,
    verbose=False, timeout=1)
113                     if len(answered) > 0:
114                         test.append("1")
115                     elif len(unanswered) > 0:
116                         test.append("0")
117                     data.append(test)
118                     time.sleep(args.time)
119                 writeIntoCSV(filename,data)
120         elif args.pois == "2.2":
121             if args.mode in ("c", "s"): print("not supported"
    )
122             else:
123                 data = []
124                 msg = scapy.Ether(dst="b8:27:eb:97:23:a2",
    src=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy
    .ARP(op="is-at", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK
    ][0]["addr"], psrc="192.168.4.100", hwdst="b8:27:eb:97:23:
    a2", pdst="192.168.4.1")
125                 for i in range(args.count):
126                     print("Test number {}..".format(i))
127                     test = [i,datetime.now()]
128                     answered,unanswered = scapy.srp(msg,
    verbose=False, timeout=1)
129                     if len(answered) > 0:
130                         test.append("1")
131                     elif len(unanswered) > 0:
132                         test.append("0")
133                     data.append(test)
```

```
134                     time.sleep(args.time)
135                 writeIntoCSV(filename,data)
136         elif args.pois == "2.3":
137             if args.mode in ("c", "s"): print("not supported"
    )
138             else:
139                 data = []
140                 msg = scapy.Ether(dst="b8:27:eb:97:23:a2",
    src="00:00:aa:bb:cc:ff") / scapy.ARP(op="is-at", hwsrc=ni.
    ifaddresses('eth0')[ni.AF_LINK][0]["addr"], psrc="
    192.168.4.100", hwdst="b8:27:eb:97:23:a2", pdst="
    192.168.4.50")
141                 for i in range(args.count):
142                     print("Test number {}..".format(i))
143                     test = [i,datetime.now()]
144                     answered,unanswered = scapy.srp(msg,
    verbose=False, timeout=1)
145                     if len(answered) > 0:
146                         test.append("1")
147                     elif len(unanswered) > 0:
148                         test.append("0")
149                     data.append(test)
150                     time.sleep(args.time)
151                 writeIntoCSV(filename,data)
152         elif args.pois == "2.4":
153             if args.mode in ("c", "s"): print("not supported"
    )
154             else:
155                 data = []
156                 msg = scapy.Ether(dst="b8:27:eb:97:23:a4",
    src=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy
    .ARP(op="is-at", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK
    ][0]["addr"], psrc="192.168.4.100", hwdst="b8:27:eb:97:23:
    a2", pdst="192.168.4.50")
157                 for i in range(args.count):
158                     print("Test number {}..".format(i))
159                     test = [i,datetime.now()]
160                     answered,unanswered = scapy.srp(msg,
    verbose=False, timeout=1)
161                     if len(answered) > 0:
162                         test.append("1")
163                     elif len(unanswered) > 0:
164                         test.append("0")
165                     data.append(test)
166                     time.sleep(args.time)
167                 writeIntoCSV(filename,data)
168         elif args.pois == "2.5":
169             if args.mode in ("c", "s"): print("not supported"
    )
170             else:
171                 data = []
172                 msg = scapy.Ether(dst="ff:ff:ff:ff:ff:ff",
    src=ni.ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy
    .ARP(op="is-at", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK
    ][0]["addr"], psrc="192.168.4.100", hwdst="ff:ff:ff:ff:ff:
    ff", pdst="192.168.4.50")
```

```
173                 for i in range(args.count):
174                     print("Test number {}..".format(i))
175                     test = [i,datetime.now()]
176                     answered,unanswered = scapy.srp(msg,
    verbose=False, timeout=1)
177                     if len(answered) > 0:
178                         test.append("1")
179                     elif len(unanswered) > 0:
180                         test.append("0")
181                     data.append(test)
182                     time.sleep(args.time)
183                 writeIntoCSV(filename,data)
184 
185         elif args.pois == "rtt":
186             data = []
187             msg = scapy.Ether(dst="ff:ff:ff:ff:ff:ff", src=ni
    .ifaddresses('eth0')[ni.AF_LINK][0]["addr"]) / scapy.ARP(
    op="who-has", hwsrc=ni.ifaddresses('eth0')[ni.AF_LINK][0][
    "addr"], psrc="192.168.4.100", hwdst="ff:ff:ff:ff:ff:ff",
    pdst="192.168.4.50")
188             for i in range(args.count):
189                 print("Test number {}..".format(i))
190                 test = [i,time.time()]
191                 answered = scapy.srp1(msg, verbose=False,
    timeout=1)
192                 if len(answered) > 0:
193                     test.append(answered.time)
194                     test.append((answered.time - test[1])
    *1000) # Convert seconds to milliseconds
195                     print("Returned in {}ms".format(test[-1])
    )
196                 data.append(test)
197                 time.sleep(args.time)
198             writeIntoCSV(filename,data)
199         elif args.pois == "cpu":
200             data = []
201             for i in range(args.count):
202                 print("Test number {}..".format(i))
203                 test = [i,datetime.now()]
204                 test.append(psutil.cpu_percent())
205                 data.append(test)
206                 time.sleep(args.time)
207             writeIntoCSV(filename,data)
208         else:
209             print("No idea")
210 
211 if __name__ == "__main__":
212     main(parseArguments())
```

Code Snippet A.12: IDPS Testing Tool Code Python3

A.5.2 IDPS & Testing tool log Timediff Comparison code

```
1 #!/usr/bin/python3
2 import time
```

```
import datetime
import os.path
import sys
import csv # Store MAC Lists

def readCsv(filen,pos):
    fileout = []
    if os.path.exists(filen):
        with open(filen,'r', newline='') as fd:
            vList = csv.reader(fd)
            for row in vList:
                fileout.append(row[pos])
    return fileout

datesOne= readCsv(sys.argv[1],int(sys.argv[2]))
datesTwo= readCsv(sys.argv[3],int(sys.argv[4]))
datetimeFormat = '%Y-%m-%d %H:%M:%S.%f'
highest = 0
lowest = 0
total = 0

for i in range(len(datesOne)):
    time = ((datetime.datetime.strptime(datesOne[i],
   datetimeFormat)\
    - datetime.datetime.strptime(datesTwo[i], datetimeFormat)
   )/ datetime.timedelta(milliseconds=1))
    print(time)
    total += time
    if time > highest:
        highest = time
    if time < lowest or lowest == 0:
        lowest = time

print("Avg: {}".format(total/len(datesOne)))
print("High: {}".format(highest))
print("low: {}".format(lowest))
```

Code Snippet A.13: IDPS & Testing tool log Timediff Comparison code

A.5.3 IDPS Testing Tool Help Menu

```
pi@phy-host-two:~ $ ./ARP_IDPS_test.py -h
usage: ARP_IDPS_test.py [-h]
                        [-p {1.1,1.2,1.3,2.1,2.2,2.3,2.4,2.5,
   rtt,cpu}]
                        [-m {s,c,r}] [-t TIME] [-c COUNT] [-o
    OUTPUT]

Simple tool to Test ARP IDPS

optional arguments:
  -h, --help            show this help message and exit
  -p {a1.1,a1.2,a1.3,a2.1,a2.2,a2.3,a2.4,a2.5,rtt,cpu}
                        Spoof type by ID
  -m {s,c,r}            Used for Reply attack testing, One
   device as (s)erver
```

```
13                          and other device as (c)lient, or (r)
    ply only
14  -t TIME                 Time between tests
15  -c COUNT                Times to test
16  -o OUTPUT               Extension text to output file
```

Code Snippet A.14: IDPS Testing Tool Help Menu

A.6 Full Testing Results

A.6.1 Experiment 1

```
ARP Request in 2864434397 | Host-Two -> 192.168.4.50 | Broadcast
ARPID: (1.1) ARP spoofed | MAC b8:27:eb:9c:86:b7 -> 00:00:aa:bb:cc:dd | for
 IP: 192.168.4.1
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Acting as proxy server in meantime for 192.168.4.100
```

Figure A.1: Exp 1: ARP A1.1 IDPS Detection Log

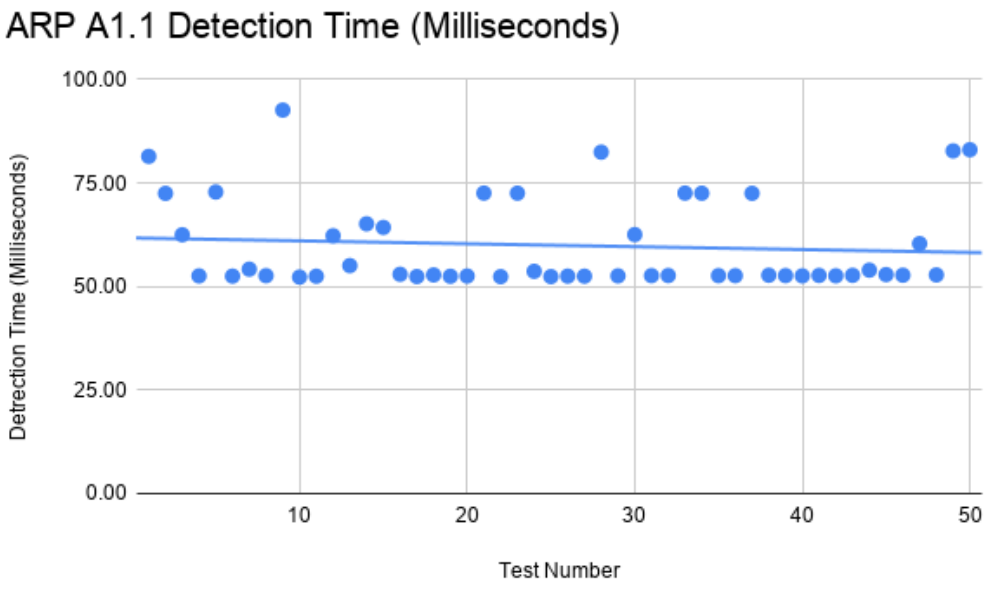

Figure A.2: Exp 1: ARP A1.1 Detection Time Results

```
ARP Request in 2864434397 | Host-Two -> 192.168.4.50 | Broadcast
ARPID: (1.2) ARP spoofed IP from unknown MAC | MAC b8:27:eb:9c:86:b7 | for
IP: 192.168.4.1
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Cannot act as proxy server for unknown MAC: b8:27:eb:9c:86:b7
```

Figure A.3: Exp 1: ARP A1.2 IDPS Detection Log

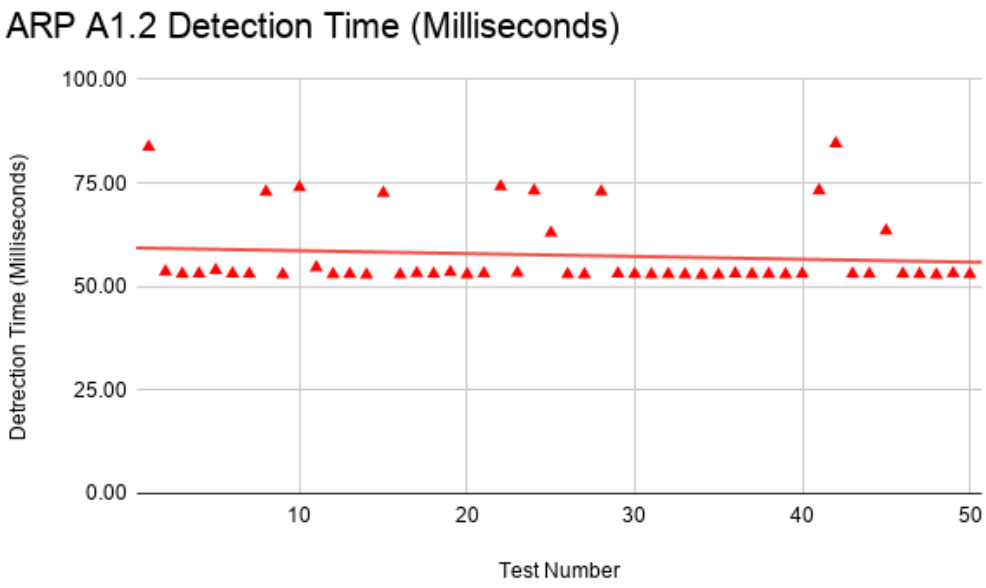

Figure A.4: Exp 1: ARP A1.2 Detection Time Results

```
ARP Request in 2864434397 | Host-Two -> 192.168.4.1 | Broadcast
ARPID: (1.3) ARP source miss-match | MAC b8:27:eb:9c:86:b7 <-> 00:00:00:01:
02:03 | from IP: 192.168.4.100
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Acting as proxy server in meantime for 192.168.4.100
```

Figure A.5: Exp 1: ARP A1.3 IDPS Detection Log

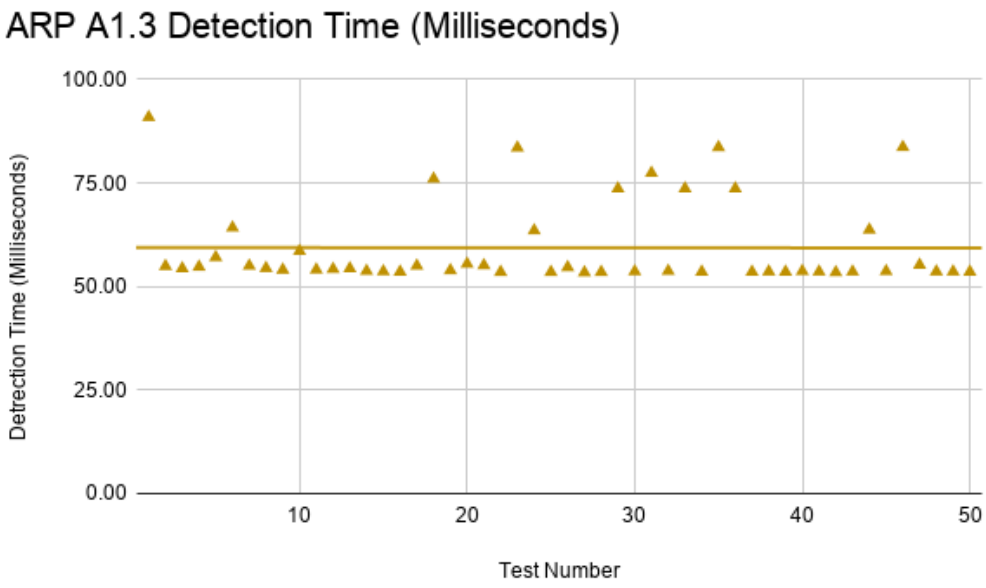

Figure A.6: Exp 1: ARP A1.3 Detection Time Results

```
ARPID: (2.1) ARP spoofed | MAC b8:27:eb:9c:86:b7 -> 00:00:aa:bb:cc:dd | for
 IP: 192.168.4.1
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Acting as proxy server in meantime for 192.168.4.100
```

Figure A.7: Exp 1: ARP A2.1 IDPS Detection Log

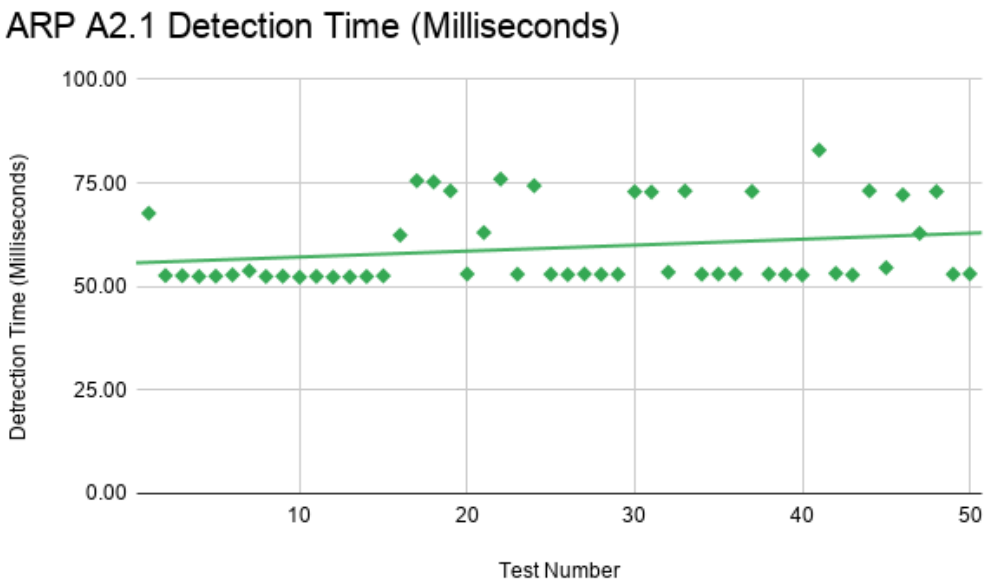

Figure A.8: Exp 1: ARP A2.1 Detection Time Results

```
ARPID: (2.2) ARP spoofed IP | MAC b8:27:eb:9c:86:b7 |  IP: 192.168.4.100
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Acting as proxy server in meantime for 192.168.4.100
```

Figure A.9: Exp 1: ARP A2.2 IDPS Detection Log

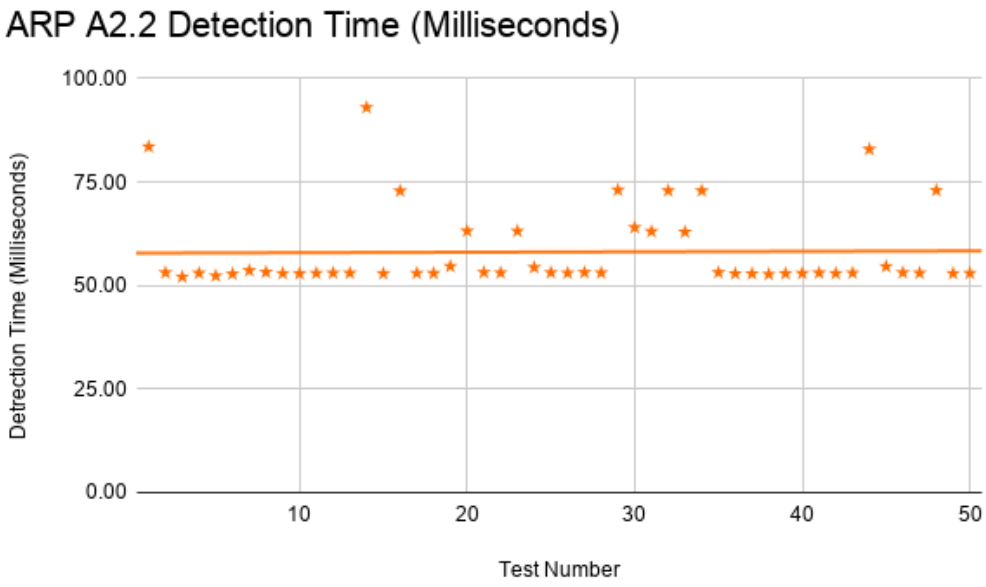

Figure A.10: Exp 1: ARP A2.2 Detection Time Results

```
ARPID: (2.3) ARP source miss-match | MAC 00:00:aa:bb:cc:ff <-> b8:27:eb:9c:
86:b7 | from IP: 192.168.4.100
Temporarily blocking MAC: 00:00:aa:bb:cc:ff, idle timeout 5s
Cannot act as proxy server for unknown MAC: 00:00:aa:bb:cc:ff
```

Figure A.11: Exp 1: ARP A2.3 IDPS Detection Log

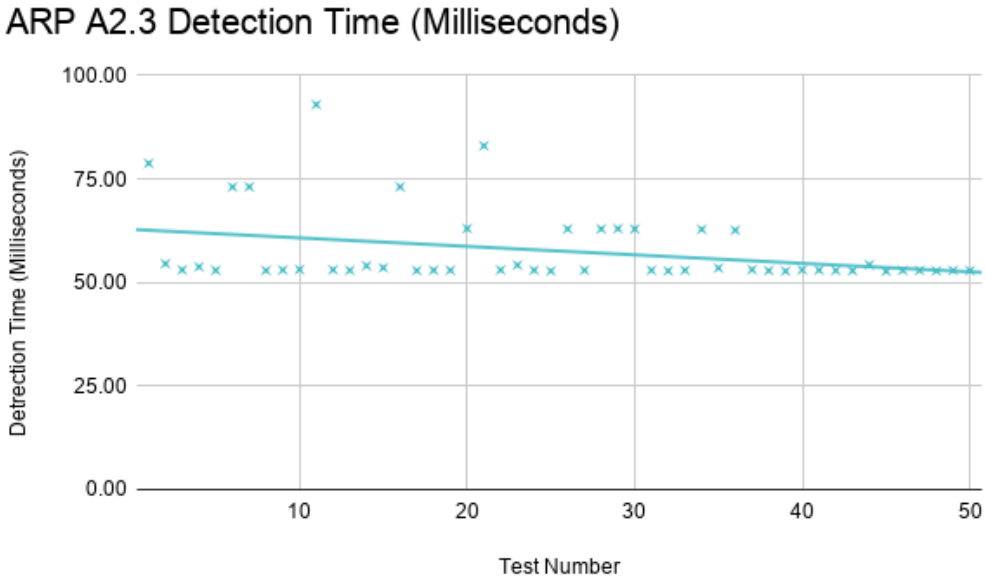

Figure A.12: Exp 1: ARP A2.3 Detection Time Results

```
ARPID: (2.4) ARP destination miss-match | MAC b8:27:eb:9c:86:b7 <-> b8:27:e
b:9c:86:b7 | from IP: 192.168.4.100
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
```

Figure A.13: Exp 1: ARP A2.4 IDPS Detection Log

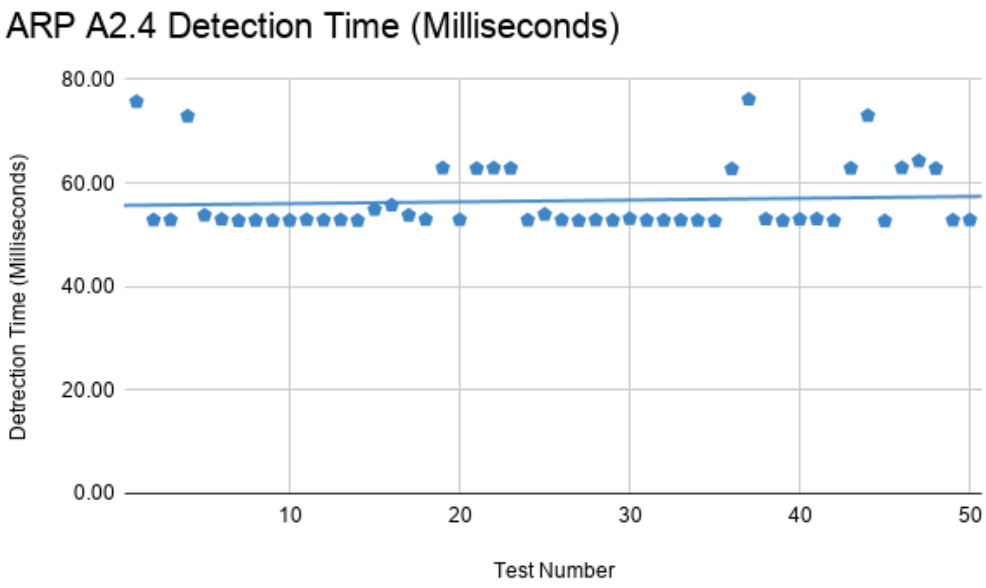

Figure A.14: Exp 1: ARP A2.4 Detection Time Results

```
ARPID: (2.5) ARP ether dst is FF:FF:FF:FF:FF:FF | MAC ff:ff:ff:ff:ff:ff <->
 ff:ff:ff:ff:ff:ff | from IP: 192.168.4.100
Temporarily blocking MAC: b8:27:eb:9c:86:b7, idle timeout 5s
Acting as proxy server in meantime for 192.168.4.100
```

Figure A.15: Exp 1: ARP A2.5 IDPS Detection Log

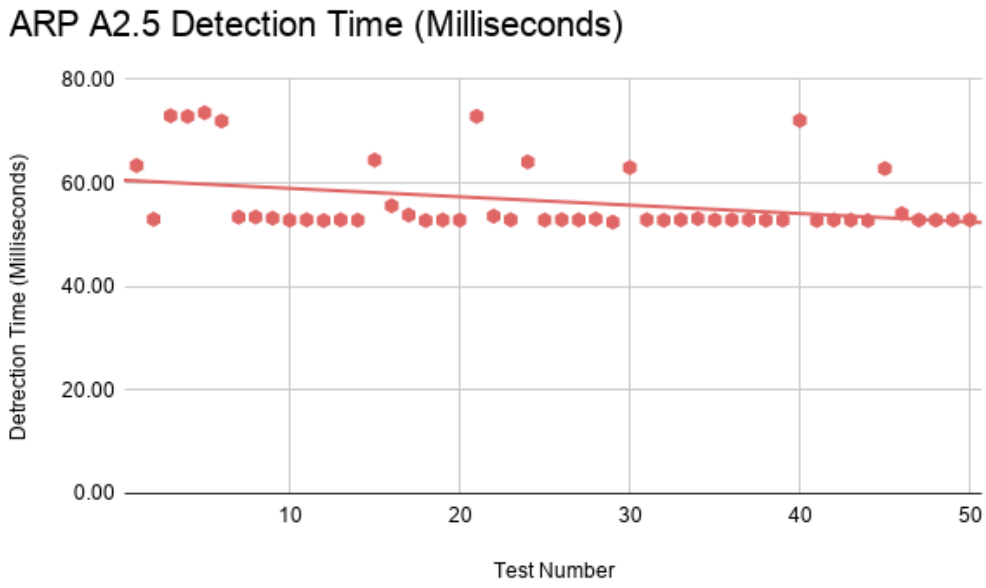

Figure A.16: Exp 1: ARP A2.5 Detection Time Results

A.6.2 Experiment 2

No.	Source	Destination	Port	Protocol	Info
1	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_ECHO_REQUEST
2	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_ECHO_REPLY
3	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=9 Ack=9 Win=1892 Len=
4	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_PACKET_IN
5	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_PACKET_OUT
6	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=111 Ack=109 Win=1892
7	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_FLOW_MOD
8	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=111 Ack=229 Win=1892
9	b8:27:eb:9c:86:b7	ff:ff:ff:ff:ff:ff		ARP	Who has 192.168.4.1? Tell 192.168.4.100
10	00:00:aa:bb:cc:dd	b8:27:eb:9c:86:b7		ARP	192.168.4.1 is at 00:00:aa:bb:cc:dd
11	192.168.4.1	192.168.4.2	6633	OpenFlow	Type: OFPT_PACKET_IN
12	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_PACKET_OUT
13	192.168.4.2	192.168.4.1	49518	OpenFlow	Type: OFPT_FLOW_MOD
14	192.168.4.1	192.168.4.2	6633	TCP	49518 → 6633 [ACK] Seq=195 Ack=431 Win=1892
15	b8:27:eb:9c:86:b7	ff:ff:ff:ff:ff:ff		ARP	Who has 192.168.4.1? Tell 192.168.4.100
16	00:00:aa:bb:cc:dd	b8:27:eb:9c:86:b7		ARP	192.168.4.1 is at 00:00:aa:bb:cc:dd

Figure A.17: Exp 2: Wireshark New and Existing ARP flow rule capture

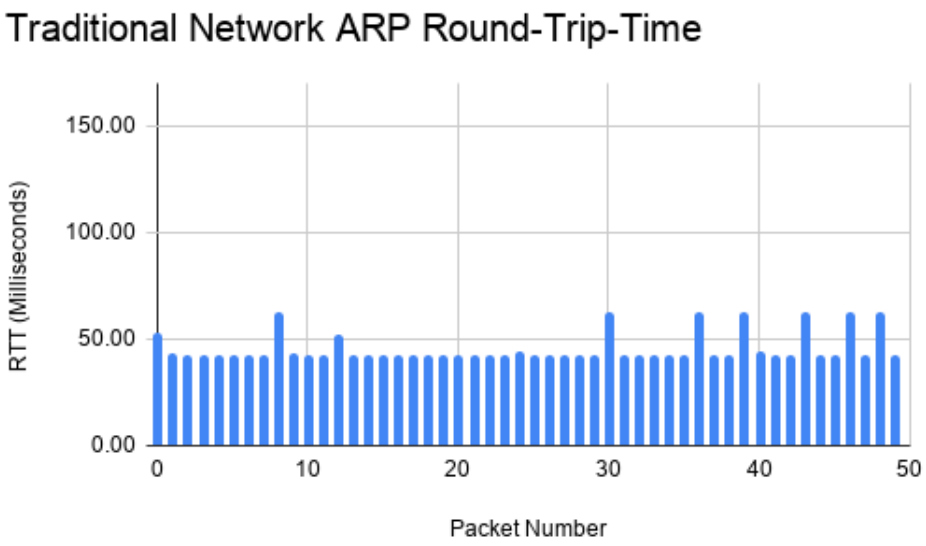

Figure A.18: Exp 2: Traditional Network ARP Round-Trip-Time Bar Chart

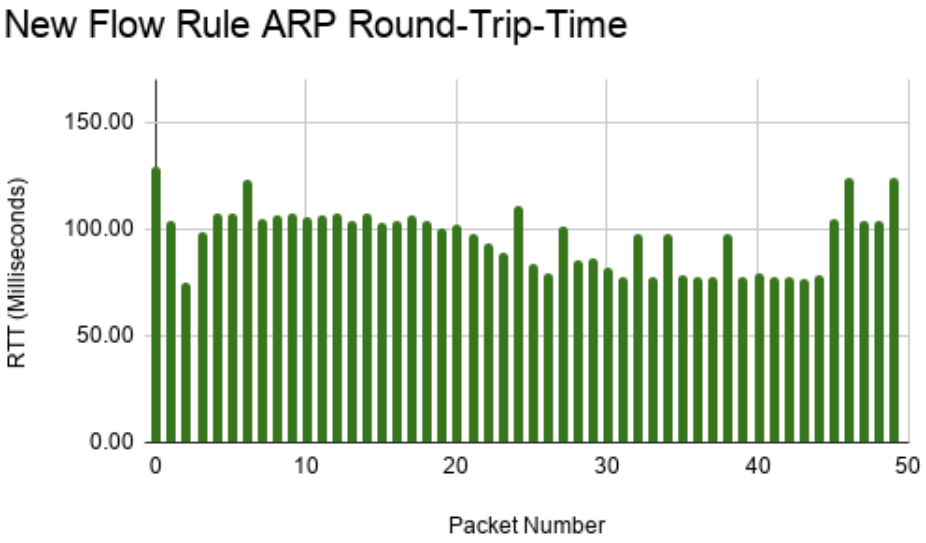

Figure A.19: Exp 2: New ARP flow rule ARP Round-Trip-Time Bar Chart

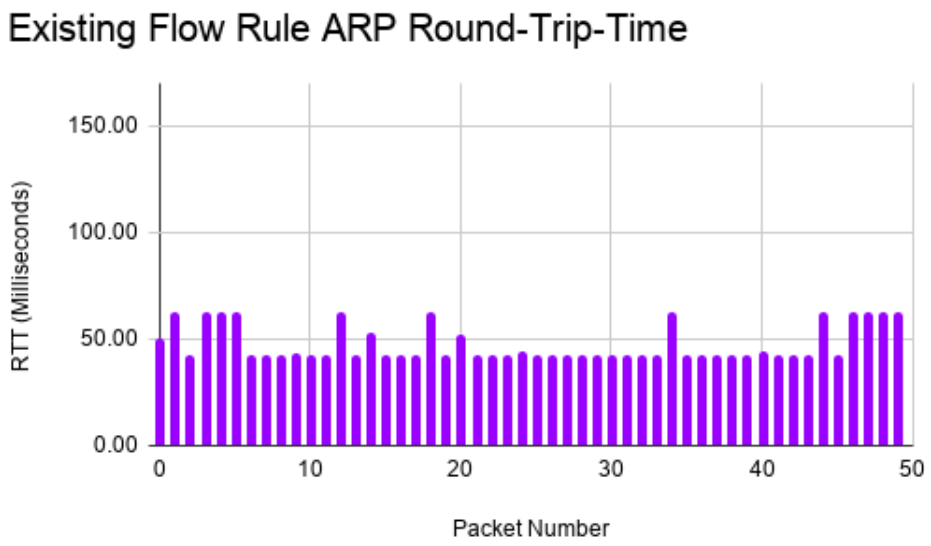

Figure A.20: Exp 2: Existing ARP flow rule ARP Round-Trip-Time Bar Chart

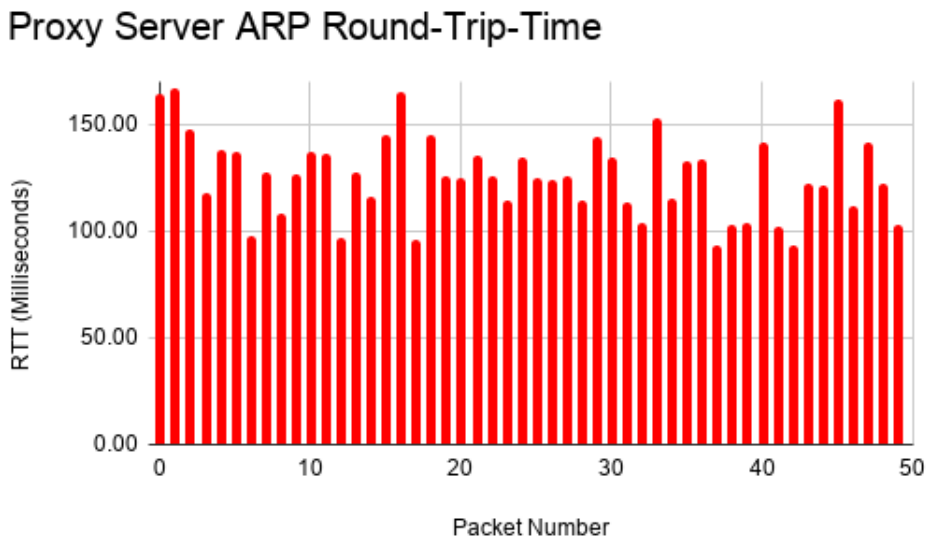

Figure A.21: Exp 2: Arp Proxy Round-Trip-Time Bar Chart

A.6.3 Experiment 3

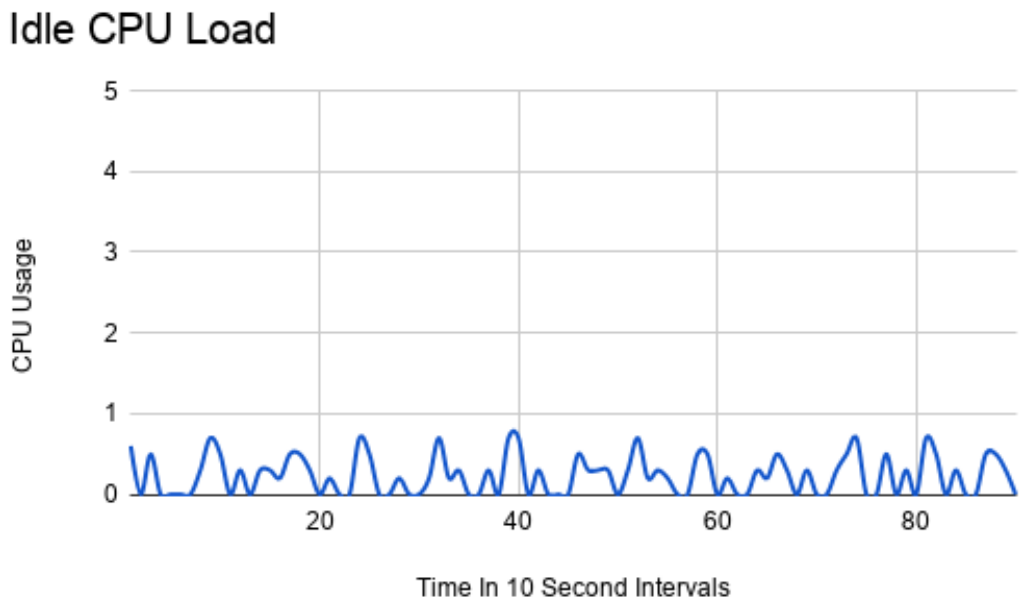

Figure A.22: Exp 3: Idle CPU Usage Line Chart

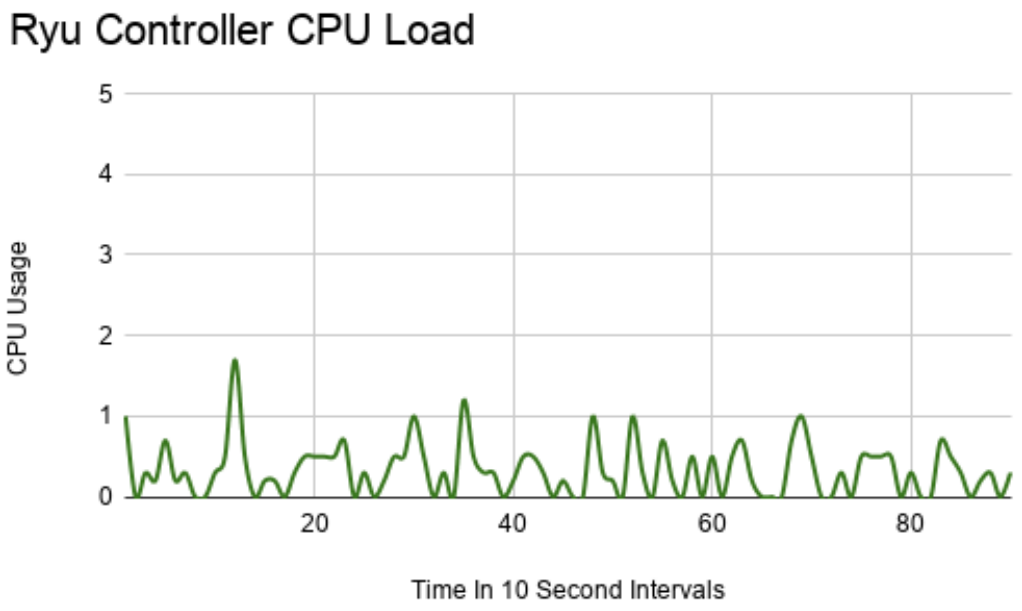

Figure A.23: Exp 3: Ryu Controller Usual CPU Usage Line Chart

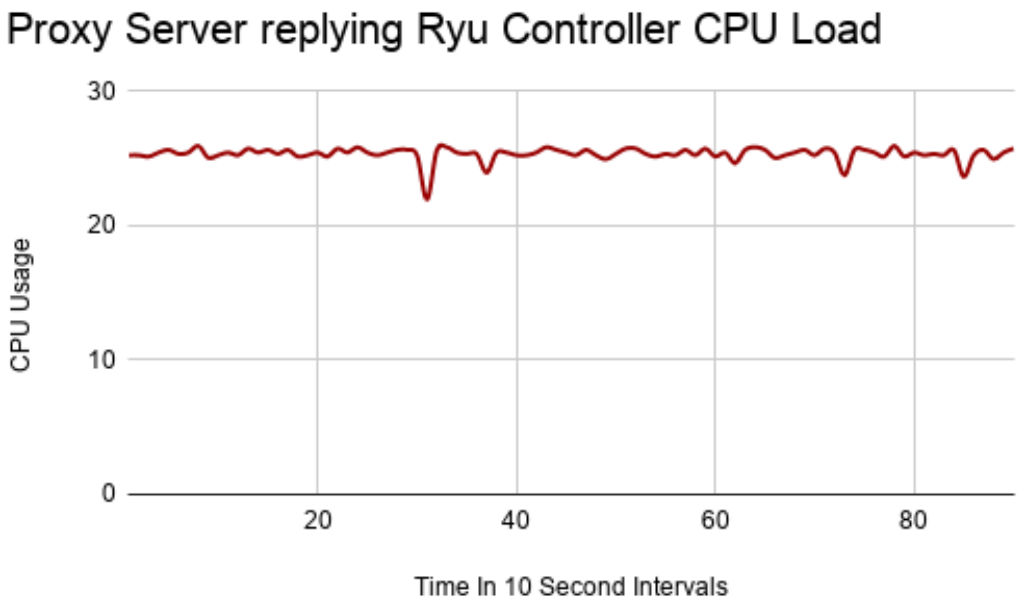

Figure A.24: Exp 3: ARP Proxy Replying CPU Usage Line Chart