



SAPIENZA
UNIVERSITÀ DI ROMA

Exploiting network programmability to mitigate DHCP starvation attacks

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea Magistrale in Cybersecurity

Candidate

Aurora Polifemo

ID number 1802485

Thesis Advisor

Prof. Marco Polverini

Academic Year 2022/2023

Contents

1	Introduction	1
1.1	Motivation & Objective	3
2	Background	5
2.1	Dynamic Host Configuration Protocol	5
2.1.1	DHCP starvation attack	7
2.2	State-of-the-Art	7
2.2.1	Mitigation of security attacks in the SDN data plane using P4-enabled switches	8
2.2.2	Detection of DHCP Starvation Attacks in Software Defined Networks: A Case Study	11
3	Technologies	14
3.1	P4	14
3.1.1	P4Runtime	15
3.2	Mininet	16
3.3	Scapy	17
4	Methodology	20
4.1	Wired Networks	20
4.1.1	Idea	20
4.1.2	Implementation	22

<i>CONTENTS</i>	3
4.2 Wireless Networks	30
4.2.1 Idea	30
5 Proof of Concept	32
5.1 Prerequisites	32
5.2 Main Verification	37
5.2.1 Execution	37
5.3 Additional Verification	42
5.3.1 Scripts' adjustments	42
5.3.2 Execution	44
6 Conclusions	46
Bibliography	47

Chapter 1

Introduction

Software Defined Networking (SDN) is an approach to network administration that permits flexible, programmatically effective network configuration to enhance its monitoring and performance.

Instead of using SDN, conventional networks are based on Internet Protocol Routing. In IP Routing whenever a packet arrives on a router, it is forwarded based on the destination of the packets itself, which means that packets coming from different sources but with the same destination have to go through the same network path. Not only is it destination-based, but it's also a hop-to-hop routing, implying that the routers do not have the whole view of the network and so the decisions are made by knowing just the next router that the packet could go to. As a result, the path is determined using the Least Cost Path (if all the links have the same cost, the Shortest Path). Following this strategy causes network traffic to be unstable because instead of using 100% of the network's capacity, only a portion of it is used.

The main innovation that SDN introduces is that it separates the data plane from the control plane so that the latter handles routing and the former handles forwarding. By removing the burden of packet directing from the control part and focusing it solely on routing choices, the network becomes programmable.

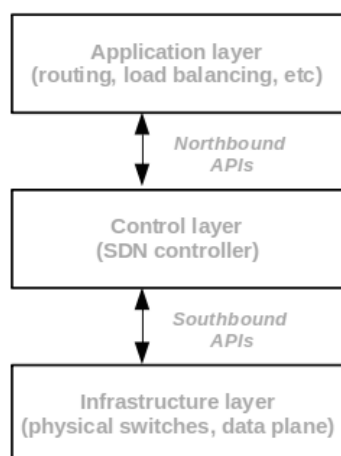
There is a logical separation in traditional networks; data traffic moves inside

the router where both functions are implemented, but from a logical standpoint, the packet is handled only by the data plane.

SDN centralises control, there is a physical separation between the two parts, and data traffic is only permitted to flow through data plane devices.

The decoupling enables the switches to perform a *Generalised Forwarding*. This implies that an SDN switch can forward packets based on a set of fields rather than just one, as in destination-based routing. Additionally, since it became a centrally managed network, controllers have a complete view of it.

A typical SDN architecture consists of three components, each of which may be situated in a distinct physical location:



- *Applications*: transmit queries to the controllers for resources or data about the network as a whole.
- *Controllers*: choose the best path for a data packet based on the information from applications.
- *Network devices*: are connected to a network, these are informed by the controller where to transfer the data.

Infrastructure is another factor that distinguishes SDN from conventional networking: traditional networking relies on hardware, whereas SDN relies on software. Exactly because the control layer is software-based, SDN provides more flexibility. In essence, the network can be managed without the need for extra hardware. [1]

The research that underpins this thesis demonstrates how a particular attack can be mitigated in the setting of software-defined networks.

Chapter 2 presents the background and the state-of-art.

The primary tools for achieving the goals stated are described in *Chapter 3*.

Chapter 4 discusses a method to mitigate the effects of common attacks in SDN.

In *Chapter 5*, the methodologies outlined in *Chapter 4* are proven and verified.

The results of the study are summarised in *Chapter 6*.

1.1 Motivation & Objective

To connect with one another, hosts must have an Internet Protocol (IP) address. IP addresses are temporary network identifiers required when a host wants to establish a connection with another host on a different network, meaning that the connection happens between several connected LANs (Local Area Networks), or in other words on a WAN (Wide Area Network); otherwise, MAC addresses are used with no requirement for IP addresses when communication occurs only within a LAN.

IP addresses can be assigned statically, which implies that the network administrator must configure each one individually. This technique is feasible for small networks. When dealing with large networks, the best option is to allocate IP addresses dynamically via DHCP.

DHCP is a crucial service that facilitates network management. Due to its widespread adoption, it has become a target for attackers attempting to breach the network, which makes it vulnerable to common network attacks, such as Denial-of-Service and Man-in-the-Middle attacks.

The thesis proposes a mitigating strategy for Denial-of-Service attacks, often known as DHCP starvation attacks. The technique discussed here is based on programmable network principles.

This method makes use of P4 (p. 14), a cutting-edge tool for Software Defined Networks that defines a table-based mechanism.

The proposed solution employs this tool to communicate and operate at the control level, resulting in a dynamic solution to the problem.

Chapter 2

Background

2.1 Dynamic Host Configuration Protocol

Dynamic Host Configuration Protocol (DHCP) is a client/server protocol that instantly assigns an IP address to an Internet Protocol (IP) host along with other relevant setup data like the subnet mask and default gateway.

Every device on a TCP/IP-based network must have a unique unicast IP address to access the network and its resources. Without DHCP, IP addresses for new computers must be configured manually. With DHCP, this entire process is automated and managed centrally. When a DHCP-enabled client connects to the network for the first time, the DHCP server assigns an address from a pool of IP addresses it manages. Rather than being static, IP addresses are dynamic, so those that are no longer in use are immediately returned to the pool for redistribution.

DHCP distributes IP addresses by means of DORA packets, which stand for Discover, Offer, Request, and Acknowledgement. The process is shown in *Figure 2.1*. [2]:

1. A DHCP client sends a DHCP Discover message in the discovery phase to look for a DHCP server. Since the client doesn't know the IP addresses of

DHCP servers, it sends the DHCP Discover packet over the network.

2. When a DHCP server receives a DHCP Discover packet from a DHCP client, it chooses an appropriate IP address from its address pool and sends a DHCP Offer packet to the DHCP client with the chosen IP address, the duration of the lease and other configuration information.
3. The DHCP client learns the addresses of the network's DHCP servers by getting a response packet from each available DHCP server. If a DHCP client gets DHCP Offer packets from multiple DHCP servers, the client only responds with a request message to the DHCP Offer packet that arrives first. Other DHCP servers can reclaim their pre-allocated IP addresses.
4. After getting a DHCP Request packet from a DHCP client, the DHCP server searches for the matching lease record using the MAC address contained in the DHCP Request packet. The DHCP server transmits a DHCP ACK packet if a record is discovered. This last message ends the process.

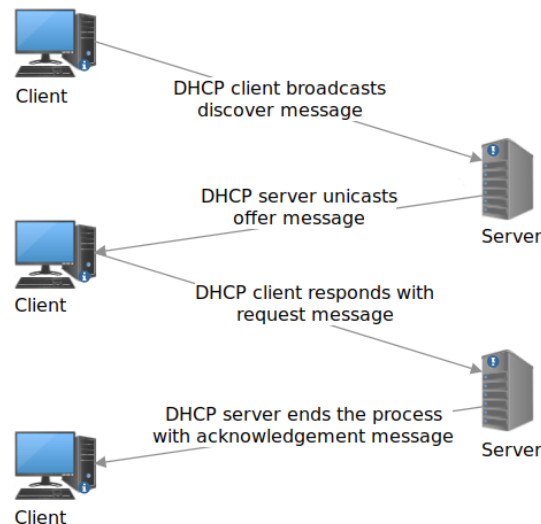


Figure 2.1. Exchanging of DORA packets between DHCP client and DHCP server

2.1.1 DHCP starvation attack

The attacker launches this assault by sending a large number of bogus DHCP Discover messages with fake source MAC addresses. The DHCP server attempts to respond to each of these false messages, which reduces the pool of IP addresses that it is able to use. A genuine user will therefore be unable to obtain an IP address via DHCP. A Denial-of-Service attack comes from this. *Figure 2.2.* illustrates the attack. [3]

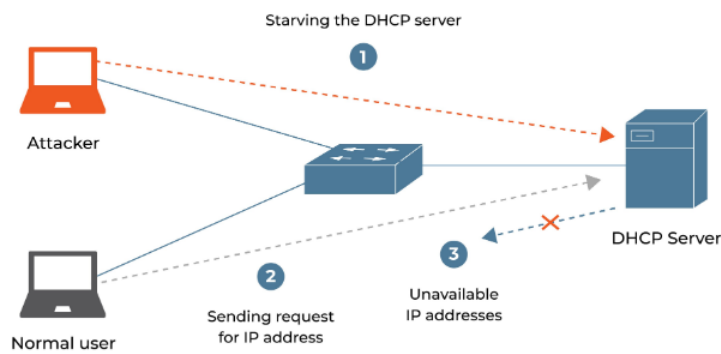


Figure 2.2. DHCP starvation attack in action

Port security can be used to limit DHCP starvation attacks in traditional networks. With port security, the amount of MAC addresses that the port can learn can be restricted. The switch would therefore pass on messages with known MAC addresses while rejecting others. By doing this, fake packets wouldn't be able to contact the DHCP server.

2.2 State-of-the-Art

The most recent stages in the development of solutions for the DHCP starvation attack in software-defined networks are described in the following two papers. These are crucial components that served to carry out the analysis in this thesis.

2.2.1 Mitigation of security attacks in the SDN data plane using P4-enabled switches

This paper [4] uses the data plane programming language P4 to examine and demonstrate some of the frequently encountered internal security attacks and associated defences. The main attacks analysed are: the DHCP starvation/rogue attack, the IP-Address spoofing attack and the ARP spoofing/poison routing attack. There is a P4 implementation/program developed for each one of those attacks.

For the purpose of this thesis, only the description of how DHCP starvation is dealt with seems significant to be explained.

A switch using DHCP snooping determines which DHCP packets are genuine in the data plane traffic and only forwards those that meet the criteria stated in the P4 code.

The network topology used consists of one P4 switch and three hosts of which one is the trusted host, one is the attacker and one is the DHCP server, as shown in *Figure 2.3.*:

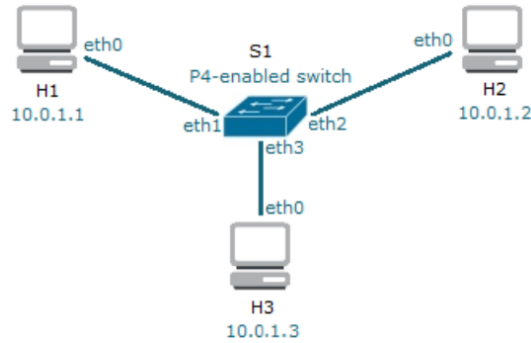


Figure 2.3. Topology of three hosts and one switch

In P4 (p. 14), tables are a core component of the match-action chain. It has one or more rules, each of which defines the key that a packet must match as well as the appropriate action to be performed when this occurs.

There are two tables defined in the P4 program *dhcp-snoop.p4*: *trusted_dhcp_client*

and *trusted_dhcp_server*.

For DHCP clients the exact match has to be on the Ethernet header source address.

```
table trusted_dhcp_client {
    key = { hdr.ethernet.srcAddr: exact; }
    actions = { pkt_fwd; drop; NoAction; }
}
```

The actions, that are allowed to be performed in case of a match, are three: *NoAction* is normally the default action, *drop* makes the packet be dropped, *pkt_fwd* is defined as follows:

```
action pkt_fwd(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

The action forwards the packet on the port specified by *port* to the destination Ethernet address specified in *dstAddr*.

DHCP servers require a longest prefix match (lpm) on the Internet Protocol (IP) source address. The actions that are allowed are the same as for the DHCP clients.

```
table trusted_dhcp_server {
    key = { hdr.ipv4.srcAddr: lpm; }
    actions = { pkt_fwd; drop; NoAction; }
}
```

The entries of the tables are set manually by defining for each switch a runtime JSON file. The entries specified in *s1-runtime.json* are:

- for the DHCP server table

```

"table": "IngressProcess.trusted_dhcp_server",
"match": {"hdr.ipv4.srcAddr": ["10.0.1.2", 32]},
"action_name": "IngressProcess.pkt_fwd",
"action_params": {
    "dstAddr": "00:00:00:00:01:01",
    "port": 1 }

```

The host with IP address *10.0.1.2* is the *trusted server* and the action that follows the match is the forwarding of the packet to the host with MAC address *00:00:00:00:01:01* on port *1* of the switch *s1*.

- for the DHCP client table

```

"table": "IngressProcess.trusted_dhcp_client",
"match": {"hdr.ethernet.srcAddr": ["00:00:00:00:01:01", 48]},
"action_name": "IngressProcess.pkt_fwd",
"action_params": {
    "dstAddr": "00:00:00:00:01:02",
    "port": 2 }

```

The host with MAC address *00:00:00:00:01:01* is the *trusted client* and the action that follows the match is the forwarding of the packet to the host with MAC address *00:00:00:00:01:02* on port *2* of the switch *s1*.

Three Python scripts are employed to mimic network activity.

Firstly, *dhcp-server.py* is run on host *H2*, then *malicious-host.py* on *H3* and *dhcp-client.py* on *H1*.

The first thing a switch does after receiving a DHCP request from a client is to see if the source MAC address is on the list of recognised clients. When the opCode field in the DHCP packet matches 1 (signifying it is a client request), the trusted

client table is used:

```
if (hdr.dhcp.opCode == 1)
    trusted_dhcp_client.apply();
```

If the MAC address is a match on the keys of this table, then the *pkt_fwd* action is performed, as seen above only packets coming from *00:00:00:00:01:01* are forwarded, and all other packets are dropped.

When the opCode field in the DHCP packet matches 2 (signifying it is a server offer), the trusted server table is used:

```
if (hdr.dhcp.opCode == 2)
    trusted_dhcp_server.apply();
```

Only packets coming from *10.0.1.2* are forwarded.

The malicious host script which is run on H3 simulates first a DHCP starvation attack by sending a DHCP discovery packet in broadcast on the network and then a DHCP rogue attack by sending a DHCP offer packet to H1. Both packets are dropped since neither H3's MAC nor the IP are in either of those tables. [5]

2.2.2 Detection of DHCP Starvation Attacks in Software Defined Networks: A Case Study

This paper [6] provides a case study to demonstrate the advantages of SDN architecture in detecting security threats. A demonstration is given in particular by creating a new application running on the controller to detect DHCP starvation attacks and maintain network lifetime by blocking the attacker's connection point.

Because of its generic interfaces, an ONOS controller is used to create the security application. Although starvation attacks are typically generated by sending DHCP

packets with proper header fields, the paper’s primary emphasis is on detecting incorrect DHCP packets.

DHCP starvation attacks are classified into two groups:

- TYPE-1: defines multiple DHCP discover messages with the same random MAC address in the CHADDR field of the DHCP packet and in the Ethernet header;
- TYPE-2: defines multiple DHCP discover messages with random MAC address in the CHADDR field of the DHCP packet but real MAC address in the Ethernet header.

A port security feature is used to limit the amount of MAC addresses permitted per network switch port in order to detect TYPE-1 attacks. When this limit is met, the port is disabled automatically. The pseudo-algorithm in *Figure 2.4.* describes this aspect.

```

1:  $MAX\_HOST \leftarrow 100$  ▷ configured by admin
2: procedure HOSTOBSERVERTIMER
3:   while true do ▷ Timer task for every 2 seconds
4:      $connPoints \leftarrow \emptyset$ 
5:      $hosts \leftarrow \emptyset$ 
6:      $connPoints \leftarrow CPConfigMap.keys()$ 
7:     for all  $cp \in connPoints$  do
8:        $hosts \leftarrow HostService.getHosts()$ 
9:       if  $hosts.size \geq MAX\_HOST$  then
10:         alert: attack detected
11:         send feedback to listeners
12:       end if
13:     end for
14:   end while
15: end procedure

```

Figure 2.4. Port Security: Limiting the number of permissible MAC addresses per port

This feature cannot be used for TYPE-2 packets, since port security cannot analyse the application layer protocols header, which means it doesn’t see the different MACs in the CHADDR field but just the same one in the Ethernet header. So if a specific MAC address is seen as trusted, all packets with that MAC in the Ethernet header, and different MAC in the CHADDR field, are accepted.

The approach for TYPE-2 packets measures the request rate, which is the number of request messages started within a given time frame. To accomplish this, the application (*Figure 2.5.*) counts the number of DHCP request messages that arrive at each port of the switch within a predefined time frame and occasionally resets all counters.

```

1:  $MAX\_DHCP\_PKT \leftarrow 100$     ▷ configured by admin
2:  $store \leftarrow ObserverStore()$ 
3: procedure PROCESSDHCPPACKET( $CP$ )
4:   if not  $observerStore.contains(CP)$  then
5:      $newObserve \leftarrow Observer(CP)$ 
6:   else
7:      $newObserver \leftarrow store.get(CP)$ 
8:   end if
9:    $newObserver.count \leftarrow newObserver.count + 1$ 
10:   $newObserver.time \leftarrow currentTime$ 
11:  if  $newObserver.count \geq MAX\_DHCP\_PKT$ 
12:    then
13:      alert: attack detected
14:      send event to listeners
15:    end if
16: end procedure

```

Figure 2.5. DHCP Request Rate Measurement, Attack Detection and Alerting

To identify DHCP starvation attacks, DHCP packets must be routed through the controller as packet-in messages. For implementing the rules through the switches, the OpenFlow protocol is used. Switches forward all DHCP messages received to the controller.

When an attack event is received, the Connection Point (CP) must be blocked because the alert shows that a DHCP starvation attack is being launched from that CP. The drop rule will be sent from the controller to the switch. No packets are routed through the controller anymore since the switch discards attack packets coming from hosts connected to that port.

Chapter 3

Technologies

This chapter presents the primary tools used to analyse the environment and to design ways for minimising attacks in software-defined networks.

3.1 P4

P4 is a programming language that explains how the data plane of a programmable forwarding element, such as a hardware or software switch, network interface card, router, or network appliance, processes packets. P4 is an acronym that stands for "Programming Protocol-independent Packet Processors."

This language is intended to specify exclusively the target's data plane functionality. A P4 program also establishes the interface through which the control plane and data plane communicate, it cannot be used to describe the target's control-plane abilities though. [7]

However, the compilation of a P4 program produces two results, one is a unique configuration that is transmitted to the switch to make it learn specific network protocols, and the other one is the Runtime API.

3.1.1 P4Runtime

"The P4Runtime API is a control plane specification for controlling the data plane elements of a device defined or described by a P4 program." [8]

A P4Runtime controller selects an appropriate configuration for a specific target and installs it using the *SetForwardingPipelineConfig* RPC. A controller can query the *ForwardingPipelineConfig* from the target via the *GetForwardingPipelineRequest* RPC.

As previously stated, the compiled P4 source program generates a P4 device configuration file and a *P4Info* metadata file. The latter is significant because it not only serves as the sole element required for communication between the P4 device and the controller, but it also contains all of the information necessary to specify the P4Runtime API messages required to configure the controller itself, which can thus be run without a P4 source program.

Even though a P4 program provides a detailed explanation of data plane behaviour, in certain circumstances having the control plane API and adequate documentation of its behaviour is sufficient for the development of the control part.

The P4 Language Consortium's repository on GitHub explains the usage of P4 by introducing a P4 Tutorial [9]. Some of these activities require the P4Runtime API, which employs Python libraries contained in the */utils/p4runtime_lib* directory:

- *helper.py*: This package contains the *P4InfoHelper* class, which is used to parse p4info files.
- *switch.py*: The *SwitchConnection* class is responsible for grabbing the gRPC client stub and connecting to the switches. Helper methods for constructing P4Runtime protocol buffer messages and making P4Runtime gRPC service calls are provided.

- *bmv2.py*: It includes the *Bmv2SwitchConnection* class and the BMv2-specific device payload employed to load the P4 program.
- *convert.py*: It provides techniques for converting friendly texts and numbers to the byte strings necessary for protocol buffer messages.

The P4Runtime API is used in this study to transmit flow entries to the switch rather than using the switch's CLI (Command-Line Interface). A P4Info helper is used to translate table, action, and key names into the IDs required by P4Runtime.

As a result, this API establishes a link between the P4 switches and the controller, allowing them to communicate dynamically.

The Virtual Machine, *vm-ubuntu-20.04*, given in the repository above has been installed and utilised for the purpose of the thesis.

3.2 Mininet

"Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking." [10]

Mininet makes it simple to test system behaviour and performance as well as experiment with topologies. Its networks run real code, which includes typical Unix/Linux network programs and also the genuine Linux kernel and network stack. Mininet is able to generate OpenFlow switches in the kernel or user space, controllers to control the switches, and hosts to communicate across the simulated network.

Mininet has no specific user interface because everything on it is controlled via Python scripts or commands. However, when building and visualising the topology in the graphic environment, Mininet can be controlled via a graphical user interface.

A custom Mininet topology can be defined by using a:

- Python file: the import of a Python library specific for Mininet topologies is required to establish a topology class in which hosts, switches, and links are defined as objects. This file is given as input when running the `mn` command with the `-custom` option.
- JSON file: simply defines the host, switches and links as in a Python dictionary. This file can be read when the P4 compiler is run.

3.3 Scapy

"Scapy is a powerful interactive packet manipulation library written in Python. Scapy is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more." [11]

Scapy is an advanced packet manipulation tool with a straightforward syntax. Python allows Scapy to manage all layers of data packet handling and it is used for network status verification and similar activities.

Scapy is also a utility for creating packets to send out in the network. These can be built by specifying and stacking the layers of the TCP/IP stack. For each one of those layers, there are some options that can be employed, such as the source and destination address or the port that needs to be used.

Let's see some simple examples of packet composition:

```
pkt = Ether(dst="ff:ff:ff:ff:ff:ff")
```

Creates a packet that sends it in broadcast on the second layer.

```
pkt = IP(src="10.0.0.1", dst="10.0.0.2")
```

Creates a packet with the source and destination IP addresses specified.

```
pkt = Ether(dst="a0:1a:93:8d:67:10")/IP(dst="10.0.0.1")
```

Creates a packet with the destination MAC and IP specified.

Once the packets have been generated, they are ready to be forwarded over the network. There are various ways to accomplish this:

- The *send()* function will deliver packets at the third layer. It will be in charge of routing.
- The *sendp()* method sends packets at layer 2.
- The *sr()* function is also used on layer 3 for sending packets and receiving answers. It returns a couple of packets, answers and unanswered packets.
- The *sr1()* function is similar to the previous one but it returns only one packet that answered.
- The *srp()* function does the same as *sr1()* for layer 2 packets.

Another powerful tool is the *sniff()* function. This function allows the capture of packets over the network by specifying the interface of the device where the packet is to be captured. In order to do so, a value for the *iface* option must be given.

When a packet is sniffed, an action should be executed; this action can be defined by using the *prn* option followed by the function to be performed. It is generally called using a lambda expression with the packet received on the interface as the argument, as shown by this example:

```
sniff(iface = eth0, prn = lambda x: function(x))
```

Other options, such as *filter* and *lfilter*, are used to filter the packets so that only the necessary ones are selected. The former is normally used to sort out packets depending on the internet protocol or on the port and it can be done by simply writing its name or number, whilst the latter can be used for more complicated queries and as a matter of fact it can employ a lambda expression similar to the one mentioned above for the *prn* option.

Chapter 4

Methodology

This is the thesis's main chapter. The idea for solving the DHCP starvation attack in SDN and the solution's implementation are presented.

Both the idea and the implementation are intended for wired networks. Nonetheless, in the concluding part of this chapter, a solution for wireless environments is offered but not implemented.

4.1 Wired Networks

The major idea and implementation are set in a wired environment, which means that each device in the network is linked to a network device through network cables.

4.1.1 Idea

The idea is formed by reading, comprehending, and analysing the two papers provided in Chapter 2.

The initial step in solving the problem is to study and run the implementation described in [4], which proposes a solution for DHCP starvation and rogue attacks. As previously stated, the P4 program of that paper employs two tables: one for the

trusted server, where host h2 is added via the *s1-runtime.json* file, and one for the trusted client, where host h1 is added via the same file as h2.

By defining the tables in this manner, only h1 is authorised to send DHCP requests; any other host is not permitted to send anything at all. Unless the host is present in the table, everyone is essentially blocked. Needless to say that this is limiting because the goal is to prevent DHCP attacks rather than to prevent a potential attacker from doing anything on the network. Furthermore, the trusted host is established statically, which implies that a host's legitimacy is evaluated in advance.

Consequently, one of the aspects that need to be changed from the approach above is to make the process dynamic, which implies that a host is seen as legitimate until it starts to act like an attacker.

The other aspect is to accept other kinds of packets as the purpose is to only block DHCP packets sent by untrusted hosts.

Inaccurate DHCP packets—of which two can be differentiated and that are the source of different DHCP attacks—are the main topic in [5]. The suggested algorithms provide a defence against both of those attacks.

One algorithm takes advantage of port security features. In essence, it counts how many hosts with different MAC addresses send a DHCP request from the same port. If this value exceeds a limit set before by the administrator, then an alert is sent.

The other algorithm counts how many DHCP requests are sent by the same MAC address in a certain time slot, and it sends an alert if this number goes beyond another threshold also established by the administrator.

Both algorithms from this second publication are taken into account for the proposed solution. In point of fact, the concept is to first determine how many

devices are permitted on a specific port, then if more than that number of DHCP packets arrive on that port in a short period of time, the port is temporarily blocked.

The premise is that if a host requests an IP address once via a DHCP discover message, sending more ones quickly afterwards on the same port is interpreted as an attempt to attack the DHCP server because it's unlikely that the same host (or even a different one) would want it again in such a brief amount of time.

Recap: The primary concept that emerged from these two publications is that all hosts in the network should be treated equally, they are neither attackers nor trusted hosts until one of them launches an attack. Second, an attack is acknowledged as such only when there are more DHCP requests on a port than there are permitted.

4.1.2 Implementation

The method relies on the control plane of the network. The switches that make up the network are P4 switches, designed and compiled using P4Runtime (para. 3.1.1), even though P4 tables aren't used.

Moving from a static to a dynamic technique is the first thing that needs to be done. As seen in the previous chapters, one tactic is to use the P4Runtime API rather than manually generating a runtime JSON file.

The P4 Tutorial repository provides the necessary libraries in the *utils* folder; crucial libraries include the mininet library and the p4runtime library.

The *run_exercise.py* file, an essential Python script, executes several of the project's key functions, including building the mininet topology and configuring switches and hosts as P4 elements.

Figure 4.1. depicts the topology that is adopted: compared to the one in the first publication [5], it is a more complicated topology. There are six hosts and two

switches, s1 and s2, with h2 serving as the DHCP server. The first three hosts are connected to the first switch while the last three are connected to the second switch. Then, each switch is logically linked to a central controller, which is specified by a different Python script that will be discussed later.

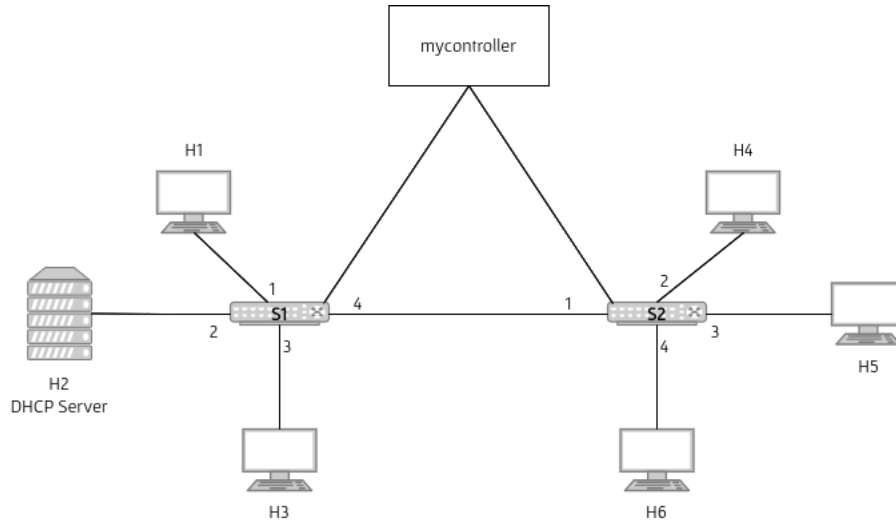


Figure 4.1. The network is composed of two P4 switches and six P4 hosts

A JSON file with explicit declarations for each element is used to construct the topology, as shown below:

```

"hosts": { "h1": {"ip": "ip not defined", "mac": "00:00:00:00:01:01"},
           "h2": {"ip": "10.0.1.2/24", "mac": "00:00:00:00:01:02"},
           "h3": {"ip": "ip not defined", "mac": "00:00:00:00:01:03"},
           "h4": {"ip": "ip not defined", "mac": "00:00:00:00:01:04"},
           "h5": {"ip": "ip not defined", "mac": "00:00:00:00:01:05"},
           "h6": {"ip": "ip not defined", "mac": "00:00:00:00:01:06"} },
"switches": { "s1": {}, "s2": {} },
"links": [ ["h1", "s1-p1"], ["h2", "s1-p2"], ["h3", "s1-p3"],
           ["s1-p4", "s2-p1"], ["h4", "s2-p2"], ["h5", "s2-p3"],
           ["h6", "s2-p4"] ]

```

Only the DHCP server is trusted and it's the only one with an IP.

The following commands are defined in the *Makefile* in the *utils* folder. They execute the *run_exercise.py* script and the API, which are used to build and compile everything. The switches are created by using the Behavioural Model-based compiler p4c and the *simple_switch_grpc*.

```
$ p4c-bm2-ss --p4v 16 --p4runtime-files build/dhcp-mitigate.p4.p4info.txt  
    -o build/dhcp-mitigate.json dhcp-mitigate.p4  
$ sudo python3 utils/run_exercise.py -t topology.json -j  
    build/dhcp-mitigate.json -b simple_switch_grpc
```

The network is operational after these procedures, and the hosts and switches are up. If a P4 program has been defined, it is compiled during this stage. The data plane is essentially in place.

The control plane serves as the main area of interest in this thesis because it is where all features are implemented and where the most important defences against DHCP attacks are employed. The control plane comprises a Python script called *mycontroller.py* that uses the P4Runtime API to emulate a real controller, similar to other exercises proposed in [9].

The *mycontroller.py* script's functionality may be broken down into three different sections for easier comprehension: creating switch connections, generating dictionaries, and sniffing packets on the switch's ports.

Setting up gRPC connections

The initial step is to connect the switches and configure their IP address and port for communication with the controller.

In the *main()* function, a P4Runtime helper (described in para. 3.1.1) is instanti-

ated. Then a switch connection object for all the switches in the topology is created, this connection is backed by a P4Runtime gRPC connection. Finally, if a P4 program has been defined, it is installed on the switches by using *SetForwardingPipelineConfig*.

```
p4info_helper = p4runtime_lib.helper.P4InfoHelper(p4info_file_path)

for s in switches:
    switch = p4runtime_lib.bmv2.Bmv2SwitchConnection(
        name=s,
        address='127.0.0.1:5005' + s[1],
        device_id=int(s[1]) - 1,
        switch.MasterArbitrationUpdate()
        switch.SetForwardingPipelineConfig(p4info=p4info_helper.p4info,
            bmv2_json_file_path=bmv2_file_path)
```

Definition of dictionaries

The second step is to generate two dictionaries: *port_mac* and *port_ndevices*.

port_mac simulates the well-known *MAC Address Table*, a table switches in real environments typically feature. Unlike *port_ndevices*, which holds the match between the switch's port and the maximum number of devices that can connect to it, this dictionary contains the match between the switch's port and the MAC address of the host connected to that port.

To direct the packet to the right device, the *port_mac* dictionary is utilised to determine which host is currently logged in to a certain port.

The number of DHCP packets that are likely to be received from a port is determined using the *port_ndevices* dictionary.

The interfaces of each switch are required in order to build these dictionaries.

They are taken from the `/sys/class/net/` directory and saved in a list named *ifaces*.

```
ifaces = list(filter(lambda i: '-eth' in i, os.listdir('/sys/class/net/')))
```

The *port_mac* dictionary is created following the controller's initial startup because each host's MAC address is added to it after the host sent a DHCP packet over the network. Its application is therefore discussed in the next section.

On the other hand, the *main()* function immediately constructs the *port_ndevices* dictionary. The *topology.json* file serves as its basis.

When a port in a cabled network has a host at the other end of the connection, only one device should be expected to be connected to it as the cable only has two ends. This changes if there is a second switch at the other end, which means that depending on how many ports the second switch has and even if it is connected to a third or fourth switch, more devices may be able to send packets from that end.

The maximum number of devices anticipated when a host is on the opposite end of the network is therefore 1. On the other hand, when there is a switch, its links must be taken into account, so it is once more determined whether there is a host or a switch.

```
def get_devices(ifaces):
    sw = {}
    for iface in ifaces:
        value = links[iface]
        if value[0] == "h": port_ndevices[iface] = 1
        else:
            sw[iface] = []
            for k,v in links.items():
                if k.startswith(value[:2]) and k != value:
                    if v.startswith("h"): sw[iface].append(v)
                    else: sw[iface].append(str(k))
```

```
for k, v in sw.items():
    x = 0
    for i in v:
        if i.startswith("h"): x += 1
        else:
            value = sw[i].copy()
            while value != []:
                temp = value.copy()
                for y in temp:
                    if y.startswith("h"):
                        x += 1
                        value.remove(y)
                    else: value = sw[y].copy()
    port_ndevices[k] = x
```

In the first `for`: if a host is connected to a certain interface, then the number of devices is set to 1. Instead, if it's a switch, a dictionary is created which includes a list of all the devices attached to that switch as its value and the interface to which the switch is linked as its key.

The previously constructed dictionary's keys and values are used in the second `for`: another `for` loop starts on the values of the switch, if the value is a host, then 1 is added to the maximum number of hosts of that switch. If the value is again a switch, a `while` loop starts until the list of this switch's connected devices is empty because if another host is found it is counted and then removed, if another switch is found, that list is replaced with the one of the new switch.

As a case in point, the following is the dictionary that is produced using the chosen topology: { 's1-eth1': 1, 's1-eth2': 1, 's1-eth3': 1, 's1-eth4': 3, 's2-eth1': 3, 's2-eth2': 1, 's2-eth3': 1, 's2-eth4': 1 }.

Sniffing of packets

The most crucial step comes last. It is where Scapy's *sniff()* function is utilised.

```
sniff(iface=ifaces, filter = "inbound and udp and (port 67 or 68)",
prn = lambda x: handle_pkt(x, switch, p4info_helper, port_ndevices,
port_ndevices_max))
```

The controller monitors all of the switches' interfaces (*ifaces*) for incoming DHCP packets as specified by *filter*. When a packet of this type is delivered from a host to one of the switches, the controller intercepts it and calls the *handle_pkt* function, passing the packet, the switch it is sent on, the helper, and the *port_ndevices* dictionary as input.

```
def handle_pkt(pkt, s1, p4info_helper, port_ndevices, port_ndevices_max):
    if BOOTP in pkt and pkt[BOOTP].op == 1:
        port_pkt = pkt.sniffed_on
        client_hw_addr = pkt[Ether].src
        host_connected = len(port_mac[port_pkt])
        if port_ndevices[port_pkt] > 0:
            if port_ndevices_max[port_pkt] == host_connected:
                port_mac[port_pkt][0] = client_hw_addr
            elif port_ndevices_max[port_pkt] > host_connected:
                port_mac[port_pkt].append(client_hw_addr)
            dhcp_packet(pkt, s1, p4info_helper, client_hw_addr)
            port_ndevices[port_pkt] -= 1
        else:
            if port_ndevices[port_pkt] == 0:
                port_ndevices[port_pkt] = -1
                t = threading.Timer(10, free_port, [port_ndevices,
                    port_ndevices_max, port_pkt])
                t.start()
```

By examining the packet's opcode, which must equal 1, the function determines whether the incoming DHCP packet is a discover packet. The following action is to get the port that the packet is sniffed on, the maximum number of devices that can be connected to that port and the host's MAC address.

There are two possibilities here:

- If the number of devices on that port is greater than 0, then there is still the chance to get/send a DHCP packet from that end.

When the condition is met, the *port_mac* dictionary is filled: if the number of devices connected to that port is the maximum there can be then the MAC address associated with that interface is replaced, because it means that there is a new device connected. If no host is yet connected or more than one host can be linked to that port then the MAC address is added to the table and not replaced.

After that, the *dhcp_packet* function is called.

```
def dhcp_packet(pkt, s1, p4info_helper, client_hw_addr):  
    if IP in pkt and pkt[IP].src == "0.0.0.0":  
        sendp(pkt, iface="s1-eth2", verbose = False)
```

This function only determines whether the sender of the packet does not already have an IP address before sending the packet to the DHCP server, which should assign an IP address to that host.

- If the number of devices in that port is less than or equal to 0, the port is blocked, because more packets than allowed have been delivered.

Therefore, after reducing the maximum number of devices to -1, the Python Threading module starts a timer during which the DHCP packets are dropped.

Following the timer's expiration, the *free_port* function is invoked, which resets the number of devices to its initial value.

```
def free_port(port_ndevices, port_ndevices_max, port_pkt):
    port_ndevices[port_pkt] = port_ndevices_max[port_pkt]
```

4.2 Wireless Networks

The above-mentioned proposal isn't ideal in a wireless environment. If a switch has an access point associated with it, then the number of devices that can send packets from that point is obviously unknown, so it's not possible to establish beforehand the number of devices allowed to be connected to that specific port, which means that there is no chance to block the port after that number is surpassed.

Hence, a different solution is suggested.

4.2.1 Idea

The main concept used for this idea is the use of a *challenge* the host needs to pass in order to get an IP address.

The topology of the network is the same as for the wired environment, but there is an additional wireless host and an access point connected to one of the switches (*Figure 4.2*).

Let's consider the switch s1 which has an access point associated with its port 5. The controller is supposed to operate as in the wired network. So if for example, a host h7 connects through a wireless connection to the access point and then sends a DHCP discover request in broadcast, this packet would be captured by the controller.

The first thing the controller does is to check whether the value of the Ethernet header address is the same as the one in the CHADDR field of the DHCP packet, as seen in [6] there could be two types of attack based on that. So by making this

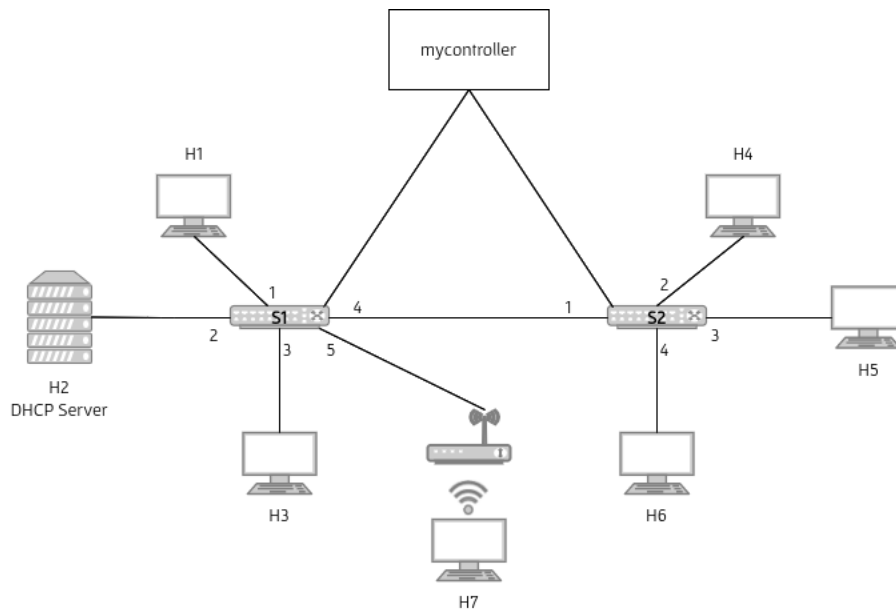


Figure 4.2. The network topology with additional wireless host and access point.

control packets which differ in these two values are directly dropped. If the values are the same the challenge is sent to the host.

The challenge consists in answering the controller with the right MAC address in the packet after a precise amount of time.

When the first DHCP packet from the host reaches the controller, the controller begins a random timeout. When this timeout expires, the controller sends a message back to the host. This message includes a second random timeout that begins as soon as the packet is received. After which, the host has to send another DHCP packet with the exact same MAC address as in the initial packet.

If the host is successful, it is considered as trustworthy and it is allowed to have an IP address; otherwise, the packet is dropped.

To ensure that each DHCP client is prepared to take a challenge and respond to it appropriately, it is advised that they all be re-implemented.

Chapter 5

Proof of Concept

The topic presented is the validation of the approach stated in the previous chapter. A brief explanation of the elements employed is given in the first section. The actual execution is displayed in the second part.

5.1 Prerequisites

There is a newly created directory on the Ubuntu virtual machine. From [9], the *utils* folder has been copied and pasted. Even though the P4 program file is not specifically stated for the study's objectives, it must be run along with the *mycontroller.py* script.

There are two basic commands, each of which is executed on a distinct terminal: the commands seen in section 4.1.2 are built and executed by `sudo make`, which is run on one terminal. The outcomes are displayed in *Figure 5.2*; the controller is launched on a different terminal and it begins scanning for packets (*Figure 5.1*).

```
p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~
```

Figure 5.1. After running `sudo ./mycontroller.py`

```

p4@p4:~/dhcp-complex$ sudo make
mkdir -p build pcaps logs
p4c-bm2-ss --p4v 16 --p4runtime-files build/dhcp-complex.p4.p4info.txt -o build/dhcp-complex.json dhcp-complex
.p4
sudo python3 utils/run_exercise.py -t topology.json -j build/dhcp-complex.json -b simple_switch_grpc
Reading topology file.
Building mininet topology.
s1 -> gRPC port: 50051
s2 -> gRPC port: 50052
*****
h1
default interface: eth0 ip not defined 00:00:00:00:01:01
*****
*****
h2
default interface: eth0 10.0.1.2 00:00:00:00:01:02
*****
*****
h3
default interface: eth0 ip not defined 00:00:00:00:01:03
*****
*****
h4
default interface: eth0 ip not defined 00:00:00:00:01:04
*****
*****
h5
default interface: eth0 ip not defined 00:00:00:00:01:05
*****
*****
h6
default interface: eth0 ip not defined 00:00:00:00:01:06
*****
Starting mininet CLI

=====
Welcome to the BMV2 Mininet CLI!
=====
Your P4 program is installed into the BMV2 software switch
and your initial runtime configuration is loaded. You can interact
with the network using the mininet CLI below.

To inspect or change the switch configuration, connect to
its CLI from your host operating system using this command:
    simple_switch_CLI --thrift-port <switch thrift port>

To view a switch log, run this command from your host OS:
    tail -f /home/p4/dhcp-complex/logs/<switchname>.log

To view the switch output pcap, check the pcap files in /home/p4/dhcp-complex/pcaps:
for example run: sudo tcpdump -xxx -r s1-eth1.pcap

To view the P4Runtime requests sent to the switch, check the
corresponding txt file in /home/p4/dhcp-complex/logs:
for example run: cat /home/p4/dhcp-complex/logs/s1-p4runtime-requests.txt

```

Figure 5.2. Result of running `sudo make`

Three distinct scripts (like those used by [4] in [5]) have been defined to show how the methodology operates. Each script is designed to act respectively as a DHCP server, a legitimate host, or a malicious attacker.

- *server-response.py* is the script run on h2, the host which simulates the DHCP server. It's listening for packets on its *eth0* interface. When a packet is captured the *handle_pkt* function is called, which checks if it's a DHCP Discover packet (opcode = 1) and if the host doesn't have an IP address.

If all the conditions are met, the packet is received by the server.

```
#!/usr/bin/env python

import sys, os

from scapy.all import *

def handle_pkt(pkt):
    if UDP in pkt and pkt[UDP].dport == 67:
        if BOOTP in pkt and pkt[BOOTP].op == 1:
            if IP in pkt and pkt[IP].src == "0.0.0.0":
                print("Got a DHCP packet from " + str(pkt[Ether].src))

def main():
    ifaces = list(filter(lambda i: 'eth' in i,
                        os.listdir('/sys/class/net/')))
    iface = ifaces[0]
    print("sniffing on %s" % iface)
    sys.stdout.flush()
    sniff(iface = iface, filter = "inbound", prn = lambda x:
        handle_pkt(x))

if __name__ == '__main__':
    main()
```

- *dhcp-packet.py* is the script run on an honest host. The host is first checked to see if it already has an IP address; if not, the *send_dhcp* function is called, which generates a packet using Scapy. Given that a DHCP Discover packet has to be built, the packet's destination is the broadcast address for both the MAC address and IP address. Additionally, opcode 1 and the UDP destination port 67 need to be specified. After its creation, the packet is sent out on the *eth0* interface.

```
#!/usr/bin/env python

import os

from scapy.all import *

def send_dhcp(iface):
    fam, hw = get_if_raw_hwaddr(iface)
    print("Sending a DHCP packet on interface %s" % (iface))
    pkt = Ether(src = get_if_hwaddr(iface), dst = 'ff:ff:ff:ff:ff:ff')
    pkt = pkt / IP(src = get_if_addr(iface), dst='255.255.255.255') /
        UDP(dport=67, sport=68) / BOOTP(op = 1, chaddr = hw) /
        DHCP(options = [('message-type', 'request'), ('end')]))
    sendp(pkt, iface = iface, verbose = False)
    print("Packet was sent")

def main():
    ifaces = list(filter(lambda i: 'eth' in i,
        os.listdir('/sys/class/net/')))
    iface = ifaces[0]
    if get_if_addr(iface) == "0.0.0.0":
        send_dhcp(iface)

if __name__ == '__main__':
    main()
```

- *malicious-packet.py* is the script run on the host that impersonates the attacker. Approximately fifty packets are created each one with a spoofed client hardware address in the DHCP packet field. All the other parameters are identical to the legitimate host; in fact, packets are sent on port 67 with opcode 1 to the broadcast address.

```
#!/usr/bin/env python

import os

from scapy.all import *

def main():

    ifaces = list(filter(lambda i: 'eth' in i,
                        os.listdir('/sys/class/net/')))

    iface = ifaces[0]

    fam, hw = get_if_raw_hwaddr(iface)

    print("Sending DHCP packet with spoofed MAC address")

    for i in range(3,50):

        if len(str(i)) == 1: i = "0" + str(i)

        spoofed_hw = (hw[:len(hw) - 1].decode() + r"\x" +
                      str(i)).encode()

        pkt = Ether(src = get_if_hwaddr(iface), dst =
                    'ff:ff:ff:ff:ff:ff') / IP(dst = '255.255.255.255') /
        UDP(dport = 67, sport = 68) / BOOTP(op = 1, chaddr =
        spoofed_hw) / DHCP(options = [('message-type','request'),
        ('end')])

        sendp(pkt, iface = iface, verbose = False)

if __name__ == '__main__':

    main()
```

On the first terminal (*Figure 5.2.*) the topology has been created and a mininet console has been opened. To demonstrate the reliability of the method, the hosts' terminals are needed. For the purpose of the description, a host from the first half of the network and a host from the second half (*Figure 4.1.*) are used, besides the host for the DHCP server.

To open such terminals, the `xterm` command must be used:

```
mininet> xterm h1 h2 h4
```

5.2 Main Verification

To better understand how the method provided in Chapter 4 operates and how the packets are dropped, the controller's functionality will be shown in both cases where the procedure is used and when it is not. To do that, the hosts must run the scripts that were previously mentioned.

5.2.1 Execution

The sequence in which everything must be carried out is as follows:

1. Run `sudo make` on one terminal (*Figure 5.2*). A Mininet console is opened.
2. The previously mentioned command `xterm h1 h2 h4` is executed on that console. This opens up three terminals, *dhcp-packet.py* is run on h1, *server-response.py* is run on h2 and *malicious-packet.py* on h4.
3. Run `sudo ./mycontroller.py` on another terminal (*Figure 5.1*). The controller is currently listening for DHCP packets on all the switches' interfaces.
4. On h2 run `python3 server-response.py`. It makes the server listen on interface *eth0*.
5. On h1 run `python3 dhcp-packet.py`.

A DHCP packet is sent from the host h1 to the host h2, but it is first intercepted by the controller, which determines the packet's course.


```

"Node: h1"
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
root@p4:/home/p4/dhcp-mitigate#

"Node: h2"
root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:00:01:01

p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~

We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server

```

If the mitigation strategy is NOT used:

- 6a. If `python3 dhcp-packet.py` is run on h1 right after the first time, the packet goes through and is delivered to the server.

```

"Node: h1"
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
root@p4:/home/p4/dhcp-mitigate#

"Node: h2"
root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:00:01:01
Got a DHCP packet from 00:00:00:00:01:01

p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~

We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server

```

- 7a. On h4 `python3 malicious-packet.py` is run. It sends numerous DHCP packets with spoofed CHADDR and they all reach the server.

```

"Node: h4"
root@p4:/home/p4/dhcp-mitigate# python3 malicious-packet.py
Sending DHCP packet with spoofed MAC address
b'\x00\x00\x00\x00\x01\x03'
WARNING: No route found (no default route?)
WARNING: No route found (no default route?)
b'\x00\x00\x00\x00\x01\x04'
WARNING: more No route found (no default route?)
b'\x00\x00\x00\x00\x01\x05'
b'\x00\x00\x00\x00\x01\x06'
b'\x00\x00\x00\x00\x01\x07'
b'\x00\x00\x00\x00\x01\x08'
[]

"Node: h2"
root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:00:01:01
Got a DHCP packet from 00:00:00:00:01:01
Got a DHCP packet from 00:00:00:00:01:04
Got a DHCP packet from 00:00:00:00:01:04
Got a DHCP packet from 00:00:00:00:01:04
Got a DHCP packet from 00:00:00:00:01:04
Got a DHCP packet from 00:00:00:00:01:04
Got a DHCP packet from 00:00:00:00:01:04
[]

p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~

We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s2-eth2
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s2-eth2
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s2-eth2
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s2-eth2
Sending the DHCP packet to the DHCP server
We received a DHCP packet on port s2-eth2
Sending the DHCP packet to the DHCP server
[]

```

If the mitigation strategy is used:

- 6b. If the command on h1 is immediately followed by another execution, then the port is blocked and the packet doesn't reach the server.

```

"Node: h1"
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
root@p4:/home/p4/dhcp-mitigate# []

"Node: h2"
root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:00:01:01
[]

p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~

We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
Port s1-eth1 is blocked
[]

```

- 7b. When `python3 malicious-packet.py` is run on h4, the controller captures and sends the first packet, then the port is blocked and the server doesn't receive any more of those packets.

```

"Node: h4"
root@p4:/home/p4/dhcp-mitigate# python3 malicious-packet.py
Sending DHCP packet with spoofed MAC address
b'\x00\x00\x00\x00\x01\x03'
WARNING: No route found (no default route?)
WARNING: No route found (no default route?)
b'\x00\x00\x00\x00\x01\x04'
WARNING: more No route found (no default route?)
b'\x00\x00\x00\x00\x01\x05'
b'\x00\x00\x00\x00\x01\x06'
b'\x00\x00\x00\x00\x01\x07'
[]

"Node: h2"
root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:01:01
Got a DHCP packet from 00:00:00:01:04
[]

p4p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~

We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
Port s1-eth1 is blocked
We received a DHCP packet on port s2-eth2
Sending the DHCP packet to the DHCP server
Port s2-eth2 is blocked
[]

```

Extra execution

An additional execution has been carried out to show how the approach functions in situations where more than one DHCP packet is delivered and it is permitted to do so. The execution takes into account that interface *s2-eth1* is designed to send out a maximum of three packets, so the mechanism is shown by utilising only the right side of the topology.

From the aforementioned sequence in para. 5.2.1, steps from 1 to 4 remain the same, just the host to use are h4, h5, h6 and obviously the DHCP server on h2.

The fifth step is carried out on all hosts (apart from h2).

So the *dhcp-packet.py* script is executed and since the interface allows for 3 packets to pass without problems, neither of the ports is blocked.

<pre> Node: h4 root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# Node: h5 root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# Node: h6 root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# Node: h2 root@p4:/home/p4/dhcp-mitigate# python3 server-response.py sniffing on eth0 Got a DHCP packet from 00:00:00:00:01:04 Got a DHCP packet from 00:00:00:00:01:05 Got a DHCP packet from 00:00:00:00:01:06 </pre>	<pre> p4@p4:~/dhcp-mitigate\$ sudo ./mycontroller.py Installed P4 Program using SetForwardingPipelineConfig on s1 Installed P4 Program using SetForwardingPipelineConfig on s2 Sniffing on: s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~ s2-eth3 ~ s2-eth4 ~ We received a DHCP packet on port s2-eth2 Sending the DHCP packet to the DHCP server We received a DHCP packet on port s2-eth3 Sending the DHCP packet to the DHCP server We received a DHCP packet on port s2-eth4 Sending the DHCP packet to the DHCP server </pre>
---	---

Then step 6a is carried out on h6. Since the packet is sent immediately after, the port linked to h6 is blocked.

<pre> Node: h4 root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# Node: h5 root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# Node: h6 root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py Sending a DHCP packet on interface eth0 Packet was sent root@p4:/home/p4/dhcp-mitigate# Node: h2 root@p4:/home/p4/dhcp-mitigate# python3 server-response.py sniffing on eth0 Got a DHCP packet from 00:00:00:00:01:04 Got a DHCP packet from 00:00:00:00:01:05 Got a DHCP packet from 00:00:00:00:01:06 </pre>	<pre> p4@p4:~/dhcp-mitigate\$ sudo ./mycontroller.py Installed P4 Program using SetForwardingPipelineConfig on s1 Installed P4 Program using SetForwardingPipelineConfig on s2 Sniffing on: s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~ s2-eth3 ~ s2-eth4 ~ We received a DHCP packet on port s2-eth2 Sending the DHCP packet to the DHCP server We received a DHCP packet on port s2-eth3 Sending the DHCP packet to the DHCP server We received a DHCP packet on port s2-eth4 Sending the DHCP packet to the DHCP server Port s2-eth4 is blocked </pre>
--	---

The example only serves to demonstrate that three packets can flow when they originate from three different hosts; if a third packet had originated from the same hosts, the port would have been blocked and the packet would have been discarded.

5.3 Additional Verification

Since the proof's sole intent is to convey the host's request, an additional component might be shown in which the server's answer is also displayed.

The server's reply is sent via the *port_mac* dictionary. Some functions are added to the controller's, the legitimate host's, and the server's scripts so that the dictionary can be used to direct the server's response to the correct host when the host requests an IP address.

5.3.1 Scripts' adjustments

The *dhcp_server* function has been added to the *mycontroller.py* script:

```
def dhcp_server(pkt, s1, p4info_helper, port_ndevices_max):
    iface = str([i for i in port_mac.keys() if pkt[Ether].dst in
                  port_mac[i]][0])
    print("Transferring from the DHCP server to " + str(pkt[Ether].dst))
    sendp(pkt, iface=iface, verbose=False)
```

The function just takes the packet sent by the server and delivers it to the host specified in the destination field.

To ensure that the previously mentioned procedure is invoked whenever a DHCP packet with opcode 2 arrives on an interface, a condition must be stated in the *handle_pkt* function.

```
elif BOOTP in pkt and pkt[BOOTP].op == 2:
    print("Servers response")
    dhcp_server(pkt, s1, p4info_helper, port_ndevices_max)
```

The *port_mac* dictionary is used in the first line of the *dhcp_server* function.

In the *handle_pkt* function of the *server-response.py* script the *assign_ip* function

is called after the packet is received by the server:

```
def handle_pkt(pkt):
    if UDP in pkt and pkt[UDP].dport == 67:
        if BOOTP in pkt and pkt[BOOTP].op == 1:
            if IP in pkt and pkt[IP].src == "0.0.0.0":
                print("Got a DHCP packet from " + str(pkt[Ether].src))
                assign_ip(pkt)

def assign_ip(pkt):
    iface = list(filter(lambda i: 'eth' in i,
                        os.listdir('/sys/class/net/')))[0]
    fam, hw = get_if_raw_hwaddr(iface)

    rpkt = Ether(src = get_if_hwaddr(iface), dst = pkt[Ether].src)
    rpkt = rpkt / UDP(dport = 68, sport = 67) / BOOTP(op = 2, chaddr = hw)
    rpkt = rpkt / DHCP(options = [('message-type', 'ack'), ('end')])
    rpkt = rpkt / Raw(load="DHCP Offer packet")
    sendp(rpkt, iface=iface, verbose=False)
```

The new function is only used to display how a message travels from the server to the host. A basic DHCP packet with opcode 2 and a message is generated and sent.

After the packet is transmitted, the *sniff()* function in the *dhcp_packet.py* script is used to monitor the server's response. When a packet is captured, the *handle_pkt* function is invoked, which just prints the packet's contents.

```
def main():
    iface = list(filter(lambda i: 'eth' in i,
                        os.listdir('/sys/class/net/')))[0]
    if get_if_addr(iface) == "0.0.0.0":
        send_dhcp(iface)
```

```

print("Listening for a response from the DHCP server on %s" % iface)

sniff(iface = iface, filter = "inbound", prn = lambda x:
      handle_pkt(x))

def handle_pkt(pkt):
    if IP in pkt and pkt[IP].src == "10.0.1.2":
        if "DHCP options" in pkt:
            print(pkt["DHCP options"].options[-1].decode())

exit(1)

```

5.3.2 Execution

Run `python3 dhcp-packet.py` on host h1.

The screenshot shows three terminal windows. The top-left window, titled "Node: h1", shows the execution of `python3 dhcp-packet.py` on host p4. The output indicates that a DHCP packet was sent on interface eth0 and that the program is now listening for a response. The top-right window, titled "p4@p4:~/dhcp-mitigate", shows the execution of `sudo ./mycontroller.py`. The output indicates that the P4 program was installed on s1 and s2, and that it is sniffing on s1-eth1 through s2-eth4. The bottom window, titled "Node: h2", shows the execution of `python3 server-response.py` on host p4. The output indicates that the program is sniffing on eth0 and has received a DHCP packet from 00:00:00:01:01.

```

"Node: h1"
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
Listening for a response from the DHCP server on eth0
DHCP Offer packet.
root@p4:/home/p4/dhcp-mitigate#

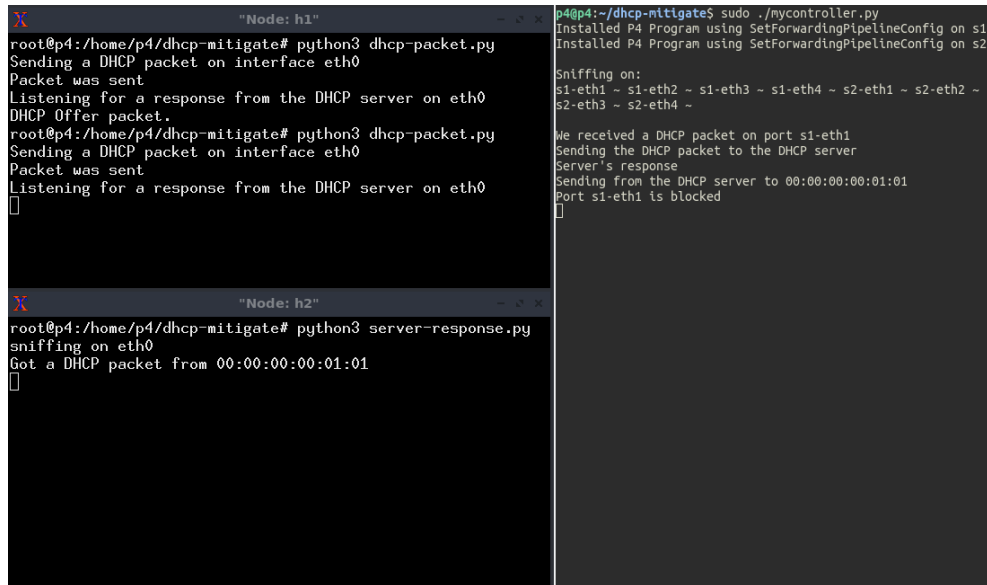
p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2
Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~
We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
Server's response
Sending from the DHCP server to 00:00:00:01:01

"Node: h2"
root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:01:01

```

The server responds with a message that travels through the controller before reaching h1, which prints the message.

Run `python3 dhcp-packet.py` a second time on h1.



The screenshot displays three terminal windows. The top-left window, titled '"Node: h1"', shows the execution of `python3 dhcp-packet.py` twice. The first run sends a DHCP packet on `eth0` and listens for a response. The second run repeats the same process. The top-right window, titled '"Node: h2"', shows the execution of `python3 server-response.py`, which sniffs on `eth0` and reports 'Got a DHCP packet from 00:00:00:00:01:01'. The bottom window, titled '"Node: h1"', shows the output of `sudo ./mycontroller.py`, which reports 'Installed P4 Program using SetForwardingPipelineConfig on s1' and 'Installed P4 Program using SetForwardingPipelineConfig on s2'. It also shows the sniffing configuration: 'Sniffing on: s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~ s2-eth3 ~ s2-eth4 ~'. The output continues with 'We received a DHCP packet on port s1-eth1', 'Sending the DHCP packet to the DHCP server', 'Server's response', 'Sending from the DHCP server to 00:00:00:00:01:01', and finally 'Port s1-eth1 is blocked'.

```
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
Listening for a response from the DHCP server on eth0
DHCP Offer packet.
root@p4:/home/p4/dhcp-mitigate# python3 dhcp-packet.py
Sending a DHCP packet on interface eth0
Packet was sent
Listening for a response from the DHCP server on eth0
[]

root@p4:/home/p4/dhcp-mitigate# python3 server-response.py
sniffing on eth0
Got a DHCP packet from 00:00:00:00:01:01
[]

p4@p4:~/dhcp-mitigate$ sudo ./mycontroller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2

Sniffing on:
s1-eth1 ~ s1-eth2 ~ s1-eth3 ~ s1-eth4 ~ s2-eth1 ~ s2-eth2 ~
s2-eth3 ~ s2-eth4 ~

We received a DHCP packet on port s1-eth1
Sending the DHCP packet to the DHCP server
Server's response
Sending from the DHCP server to 00:00:00:00:01:01
Port s1-eth1 is blocked
[]
```

Since the controller didn't send the DHCP discover packet to the server but instead blocked the port, there is no response from the server.

Chapter 6

Conclusions

To sum up everything that has been stated previously, the main problem software-defined networks encountered when using the DHCP technique has been solved.

After a description of how SDN functions and an exposé of various practitioners tackling the same problem, the approach and its verification have been provided.

The problem has been addressed by utilising the network's programmability, and even though the solution dealt with the control plane rather than the data plane, it nevertheless remained practicable to implement. Since the solution is given in a simulated scenario, it should be adjusted for a real setting.

All of the choices made during the research's creation were believed to be the most pertinent, but this does not preclude the possibility that other, possibly superior choices could have been made.

In point of fact, as stated at the thesis's beginning, the technique is only applicable to wired networks. Therefore, if the objective is to work on wireless networks, the given idea for a wireless environment can be applied and appropriately modified.

Bibliography

- [1] Understanding Software-Defined Networking (SDN), Andrew Magnusson, August 2022, <https://www.strongdm.com/blog/software-defined-networking>
- [2] What is DHCP: Interaction Process and Packet Structure, VSOL, April 2022, <https://www.vsolcn.com/blog/what-is-dhcp.html>
- [3] The Ultimate Guide to DHCP Spoofing and Starvation Attacks, PivIT Global, May 2022, <https://info.pivitglobal.com/resources/dhcp-spoofing-and-starvation-attacks>
- [4] N. Narayanan, G. C. Sankaran and K. M. Sivalingam, "Mitigation of security attacks in the SDN data plane using P4-enabled switches," 2019 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), Goa, India, 2019, pp. 1-6, doi: 10.1109/ANTS47819.2019.9118071.
- [5] "Data Plane Security with P4 (DPS-P4)," <https://github.com/nniranjhana/dps-p4/tree/master/src/sw/dhcp-snooping>, Oct. 2019.
- [6] C. Toprak, C. Turker and A. T. Erman, "Detection of DHCP Starvation Attacks in Software Defined Networks: A Case Study", 2018 3rd International Conference on Computer Science and Engineering (UBMK), Sarajevo, Bosnia and Herzegovina, 2018, pp. 636-641, doi: 10.1109/UBMK.2018.8566268.

-
- [7] P416 Language Specification, The P4 Language Consortium, May 2017, <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>
 - [8] P4Runtime Specification, The P4.org API Working Group, July 2021<https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html>
 - [9] P4 Tutorial, <https://github.com/p4lang/tutorials>
 - [10] Mininet Overview, <http://mininet.org/overview/>
 - [11] Scapy Documentation, <https://scapy.readthedocs.io/en/latest/>