UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

Programmable dataplanes emulation with P4 language in mininet environment

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Via Vetoio, 67100 L'Aquila
http://disim.univaq.it

Università degli Studi dell'Aquila
Palazzo Camponeschi
Piazza Santa Margherita 2, 67100 L'Aquila
http://www.univaq.it

Università degli Studi dell'Aquila

Stefano Hinic - stefano.hinic@student.univaq.it
Umberto Impicciatore - umberto.impicciatore@student.univaq.it

Contents

List of Figures

CHAPTER 1

Introduction

1.0.1 Project Proposal

Programmable dataplanes emulation with P4 language in mininet environment.

1.0.2 Developed Project

The goal of this project is to show the flexibility and the power of using P4 to program switches and P4Runtime to modify their behaviour at runtime. In order to show the network behaviour composed by P4 programmed switches, Mininet was used. A network controller was implemented and used to control at runtime the switches defining the packet forwarding rules and implementing security checks in the network. In order to emulate network traffic scapy and iperf3 tools were used. Moreover the GeoIP2 database was installed and used to localize IPV4 addresses and to guarantee security against spoofing on custom defined header. This project will give an overview of the following topics: Mininet, DoS and DDoS attacks, Scapy, GeoIP2 database, P4 language, P4Runtime. The design and implementation of the network and the results and analysis of the experiments conducted on it will be described.

1.0.3 Implemented Functions

Main functions that have been implemented and that will be presented in this report are:

- Constructing a network in Mininet Environment
- Programming switches with a P4 custom code
- Running a controller that uses P4runtime to manage operations and write forwarding rules on P4 programmed switches
- Detecting spoofing of a custom header by using a real database available online
- Detecting DDoS and Dos (traffic congestion in general) attacks against an host of the network

Next chapters will detail these features.

CHAPTER 2

Mininet

2.1 What is mininet?

Mininet is a network emulation orchestration system. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel[3]. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine; you can ssh into it and run arbitrary programs (including anything that is installed on the underlying Linux system). The programs you run can send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch, router, or middlebox, with a given amount of queueing. In short, Mininet's virtual hosts, switches, links, and controllers are the real thing – they are just created using software rather than hardware – and for the most part their behavior is similar to discrete hardware elements.

2.2 Mininet features

Mininet is also a great way to develop, share, and experiment with Software-Defined Networking (SDN) systems using OpenFlow and P4. Mininet offers the following features:

- starting up a simple network taking just a few seconds
- creating custom topologies: a single switch, larger Internet-like topologies, a data center, or anything else
- running real programs: anything that runs on Linux is available for you to run, from web servers to TCP window monitoring tools to Wireshark
- customizing packet forwarding: Mininet's switches are programmable using the OpenFlow protocol and P4
- running Mininet on your laptop, on a server, in a VM, on a native Linux box or in the cloud
- sharing and replicating results: anyone with a computer can run your code once you've packaged it up

- using it easily: you can create and run Mininet experiments by writing simple (or complex if necessary) Python scripts
- fast prototyping for new networking protocols
- simplified testing for complex topologies without the need of buying expensive hardware

Notice that running on a single system is convenient, but it imposes resource limits: if your server has 3 GHz of CPU and can switch about 10 Gbps of simulated traffic, those resources will need to be balanced and shared among your virtual hosts and switches.

2.3 A quick tutorial

2.3.1 Mininet use with the default topology

In this section, the "hello world" of mininet is reported. By running the command reported in the screenshot 2.1

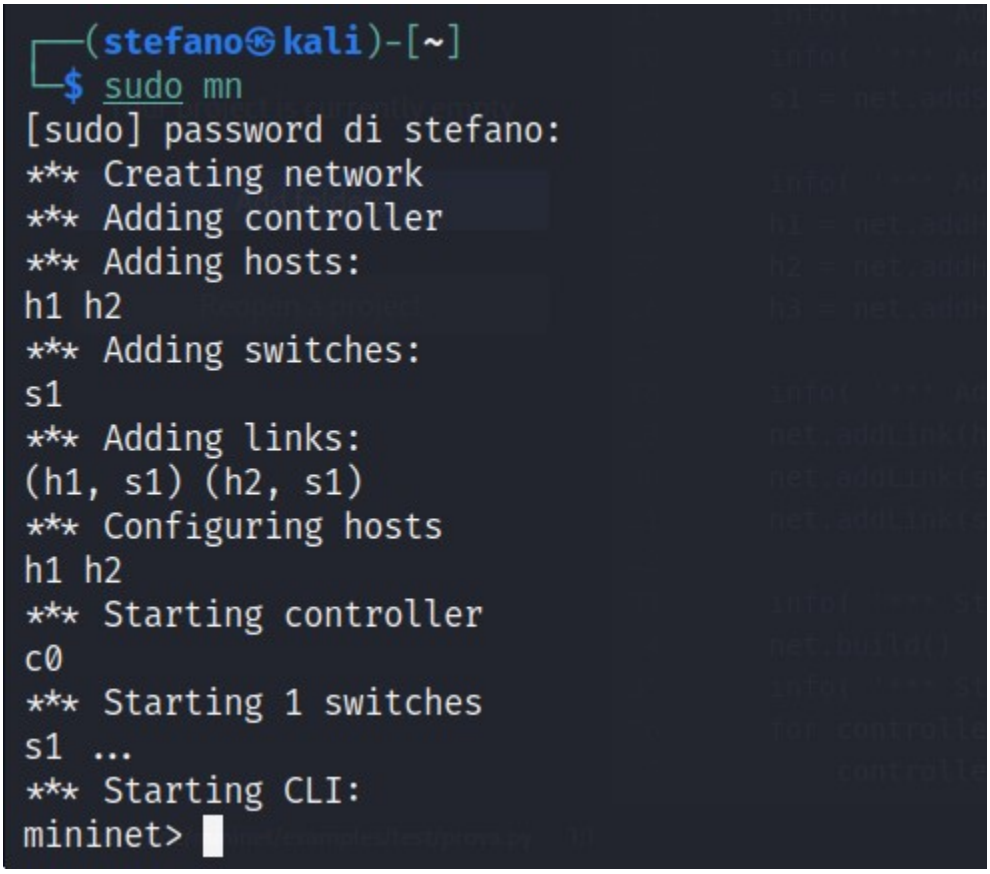


Figure 2.1: Starting Mininet using the CLI

from linux terminal, Mininet initializes the topology and launches its command line interface. The mn utility creates Mininet network from the command line. It can create parametrized topologies, invoke the Mininet CLI, and run tests. Without specifying the network topology, mininet will choose the default one, which consists of a switch connected to two hosts as shown in the figure 2.2.

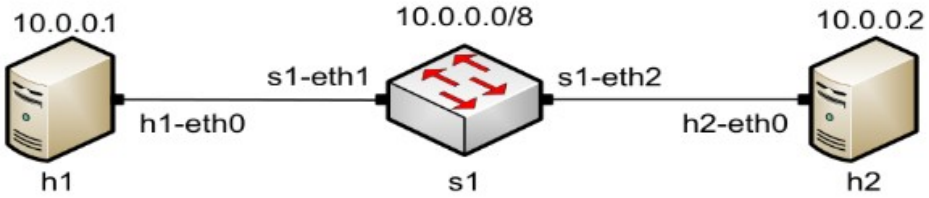


Figure 2.2: Mininet default topology

Once you enter the CLI, by running the help command you can see the list of Mininet CLI commands and examples on their usage. The figure 2.3

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
mininet> nodes
available nodes are:
c0 h1 h2 s1
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet>
```

Figure 2.3: Mininet default topology

shows the creation of the default network and some commands to understand the network topology and test connectivity between hosts through pings. Mininet allows you to execute commands on a specific device by specifying the device first, followed by the command. Mininet also allows you to open the terminal of a network node and give commands directly from there. These two equivalent ways of executing commands on a specific node in the network are shown in figure 2.4.

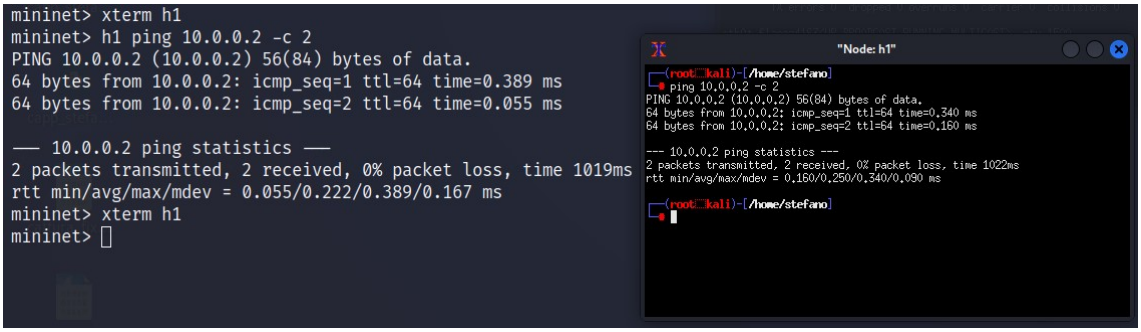


Figure 2.4: Starting Mininet using the CLI

It is possible to stop the emulation by typing the ”exit” command, and if Mininet were to crash for any reason, the ”sudo mn – c command” can be utilized to clean a previous instance (directly from the linux terminal).

2.3.2 Build and emulate a network in Mininet using the GUI

Mininet offers a simple GUI network editor called MiniEdit to build and later emulate a network in a very intuitive way. To start it, simply run the python script ”miniedit.py” located in the ”examples” folder which is in turn located in the ”mininet” folder. The GUI, shown in 2.5,

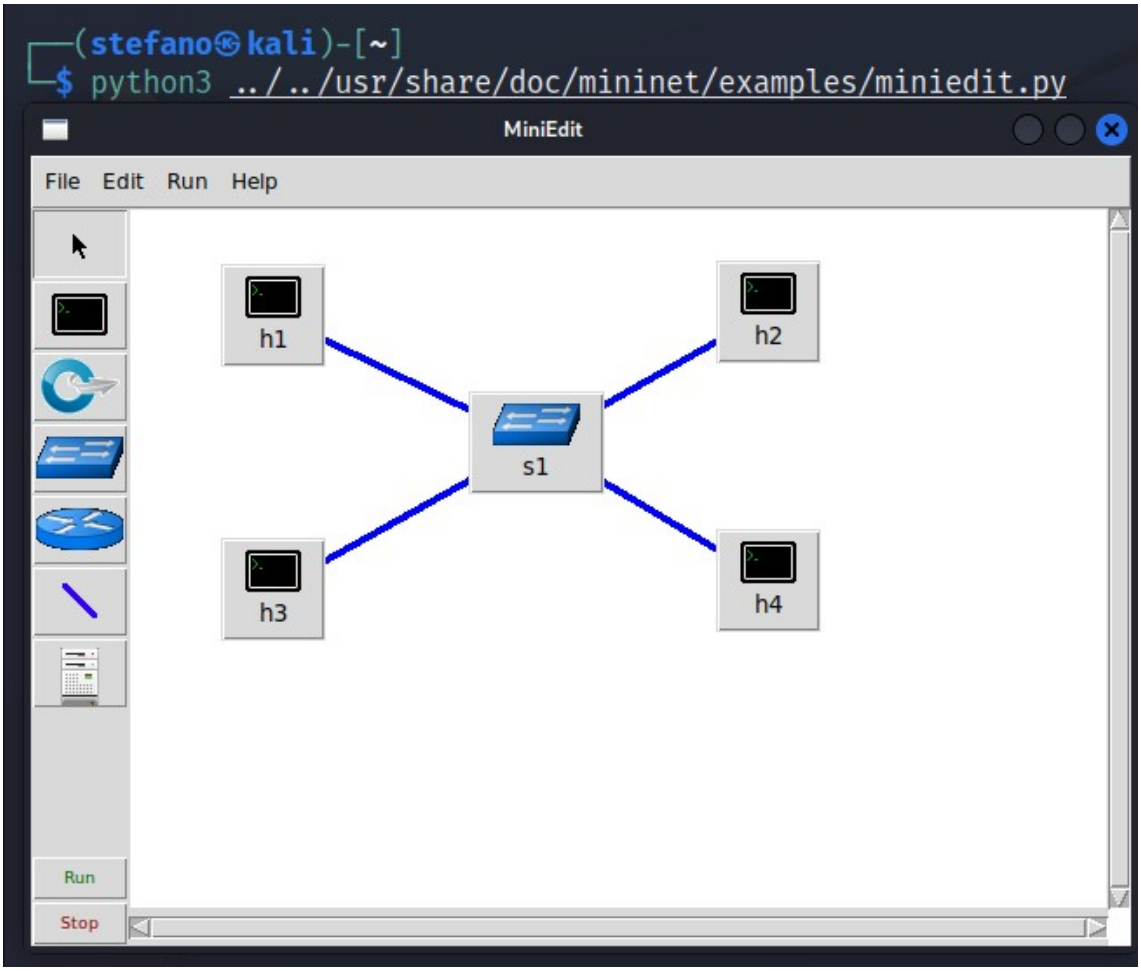


Figure 2.5: MiniEdit, the Mininet GUI

allows you to build the network by adding hosts, routers, regular or openflow-supporting switches, controllers, and connecting them through links to achieve the desired topology. Right-clicking on the host to be configured, and then selecting ”properties,” will open a window from which certain host parameters can be configured, as in figure 2.6.

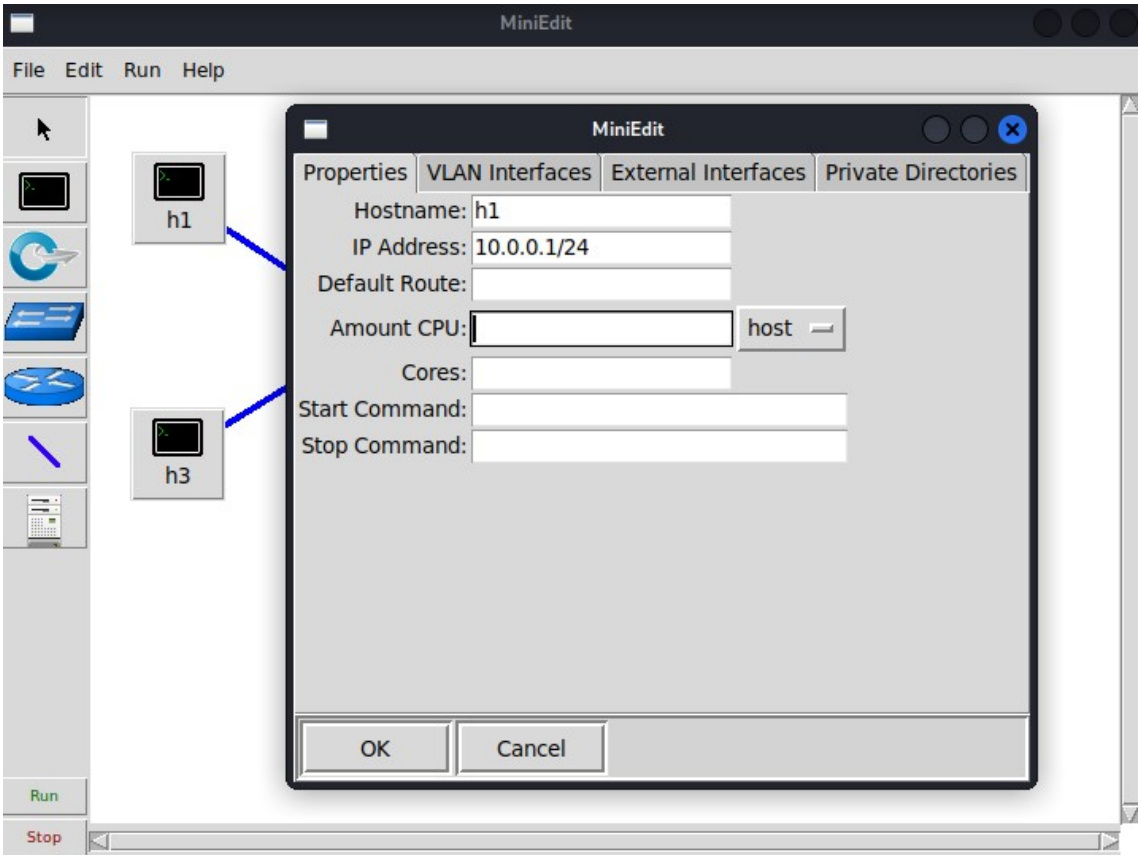


Figure 2.6: Host configuration window in MiniEdit

The run and stop buttons that are in the lower left corner of the GUI allow you to start and stop the emulation. After pressing the ”run” command, you can open a terminal by right-clicking on the host icon and select ”Terminal”. This opens a terminal on the selected host and allows the execution of commands on it. It is often useful to save the designed network topology, in particular when its complexity increases, for this reason MiniEdit enables you to save the topology to a file. This way you can reopen it and continue to edit and work on it through MiniEdit without having to redesign it from scratch. In the MiniEdit application you can save the designed topology by clicking ”File” and then selecting ”Save”. To load the topology in the MiniEdit application, first click on ”File” and then on ”Open”. MiniEdit also allows you to save the topology in a python script, which when executed will start Mininet with the previously saved topology. In order to do that, after completing the network design in MiniEdit, simply click on ”File” and then on ”Export Layer 2 Script”. To start it, simply run the python script that will instantiate the network in Mininet with the previously exported topology and it will allow you to interact with the network from the Mininet CLI as usual, as shown in 2.7.

```
┌──(stefano㉿kali)-[/usr/…/doc/mininet/examples/test]
└─$ sudo python3 custom_topo.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h1 h2 h3
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0
mininet>
```

Figure 2.7: Starting exported custom topology

2.3.3 Build and emulate a custom network in Mininet using the Python API

Custom topologies can be easily defined as well, by using a simple Python API, and an example is provided in 2.1. This example connects two switches directly, with a single host off each switch.

```
1
2  """Custom topology example
3
4  Two directly connected switches plus a host for each switch:
5
6     host --- switch --- switch --- host
7
8  Adding the 'topos' dict with a key/value pair to generate our newly defined
9  topology enables one to pass in '--topo=mytopo' from the command line.
10 """
11
12 from mininet.topo import Topo
13
14 class MyTopo( Topo ):
15     "Simple topology example."
16
17     def build( self ):
18         "Create custom topo."
19
20         # Add hosts and switches
```

```
21          leftHost = self.addHost( 'h1' )
22          rightHost = self.addHost( 'h2' )
23          leftSwitch = self.addSwitch( 's3' )
24          rightSwitch = self.addSwitch( 's4' )
25
26          # Add links
27          self.addLink( leftHost, leftSwitch )
28          self.addLink( leftSwitch, rightSwitch )
29          self.addLink( rightSwitch, rightHost )
30
31
32  topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Listing 2.1: MyTopo() class

When a custom Mininet file is provided, it can add new topologies, switch types, and tests to the command-line. Mininet's API allows you to create custom networks with a few lines of Python. For example, the following script 2.2:

```
1  from mininet.net import Mininet
2  from mininet.topolib import TreeTopo
3  tree4 = TreeTopo(depth=2,fanout=2)
4  net = Mininet(topo=tree4)
5  net.start()
6  h1, h4  = net.hosts[0], net.hosts[3]
7  print h1.cmd('ping -c1 %s' % h4.IP())
8  net.stop()
```

Listing 2.2: Python API to customize the Network

creates a small network (4 hosts, 3 switches), and performs pings between them.

CHAPTER 3

DoS attack

3.1 Denial-of-Service attack (DoS attack)

A denial-of-service (DoS) attack occurs when legitimate users are unable to access information systems, devices, or other network resources due to the actions of a malicious cyber threat actor [5]. Services affected may include email, websites, online accounts (e.g., banking), or other services that rely on the affected computer or network. A denial-of-service condition is accomplished by flooding the targeted host or network with traffic until the target cannot respond or simply crashes, preventing access for legitimate users. DoS attacks can cost an organization both time and money while their resources and services are inaccessible. There are many different methods for carrying out a DoS attack. The most common method of attack occurs when an attacker floods a network server with traffic. In this type of DoS attack, the attacker sends several requests to the target server, overloading it with traffic. These service requests are illegitimate and have fabricated return addresses, which mislead the server when it tries to authenticate the requestor. As the junk requests are processed constantly, the server is overwhelmed, which causes a DoS condition to legitimate requestors. There are two general forms of DoS attacks: those that crash services and those that flood services. The most serious attacks are distributed.

3.1.1 Distributed Denial-of-Service attack (DDoS attack)

A distributed denial-of-service (DDoS) attack occurs when multiple machines are operating together to attack one target. DDoS attackers often leverage the use of a botnet—a group of hijacked internet-connected devices to carry out large scale attacks. Attackers take advantage of security vulnerabilities or device weaknesses to control numerous devices using command and control software. Once in control, an attacker can command their botnet to conduct DDoS on a target. In this case, the infected devices are also victims of the attack. Botnets—made up of compromised devices—may also be rented out to other potential attackers. Often the botnet is made available to "attack-for-hire" services, which allow unskilled users to launch DDoS attacks. DDoS allows for exponentially more requests to be sent to the target, therefore increasing the attack power. It also increases the difficulty of attribution, as the true source of the attack is harder to identify. More sophisticated strategies are required to mitigate this type of attack,

as simply attempting to block a single source is insufficient because there are multiple sources. Since the incoming traffic flooding the victim originates from different sources, it may be impossible to stop the attack simply by using ingress filtering. It also makes it difficult to distinguish legitimate user traffic from attack traffic when spread across multiple points of origin. As an alternative or augmentation of a DDoS, attacks may involve forging of IP sender addresses (IP address spoofing) further complicating identifying and defeating the attack. A DoS or DDoS attack is analogous to a group of people crowding the entry door of a shop, making it hard for legitimate customers to enter, thus disrupting trade. Criminal perpetrators of DoS attacks often target sites or services hosted on high-profile web servers such as banks or credit card payment gateways. Revenge, blackmail and hacktivism can motivate these attacks.

3.1.2 How to know if an attack is happening

Symptoms of a DoS attack can resemble non-malicious availability issues, such as technical problems with a particular network or a system administrator performing maintenance. However, the following symptoms could indicate a DoS or DDoS attack:

- Unusually slow network performance
- Unavailability of a particular website
- An inability to access any website

The best way to detect and identify a DoS attack would be via network traffic monitoring and analysis. Network traffic can be monitored via a firewall or intrusion detection system. An administrator may even set up rules that create an alert upon the detection of an anomalous traffic load and identify the source of the traffic or drops network packets that meet a certain criteria.

3.1.3 Methods of attack

The simplest DoS attack relies primarily on brute force, flooding the target with an overwhelming flux of packets, oversaturating its connection bandwidth or depleting the target's system resources. Bandwidth-saturating floods rely on the attacker's ability to generate the overwhelming flux of packets. A common way of achieving this today is via distributed denial-of-service, employing a botnet. An application layer DDoS attack is done mainly for specific targeted purposes, including disrupting transactions and access to databases. It requires fewer resources than network layer attacks but often accompanies them. An attack may be disguised to look like legitimate traffic, except it targets specific application packets or functions. The attack on the application layer can disrupt services such as the retrieval of information or search functions on a website.

3.1.4 Simulating network traffic using iPerf3

The iPerf3 tool was used in this work to generate a significant amount of traffic within the network in order to simulate DoS or DDoS attacks. iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks[6]. It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test it reports the bandwidth, loss, and other parameters. The tool is simple to use: a single executable runs on both the client and the server. Command-line parameters indicate which system will take on the role of the server – the target –and which will be the client. It makes no difference which is which. In the example shown in the figure 3.1, host h2 works in iperf3 -s mode (server mode), listening on the default port 5201, on which h1 transmits, which works in iperf3 -c mode (client mode). Through the -p command, executed in iperf3 -s mode, it is possible to specify the port on which to listen, and in iperf3 -c mode, the server port on which to transmit, previously enabled for listening. In iperf3 -c mode, it is possible to set some session parameters, such as bandwidth through the -b tag, or transmission duration with the -t tag (it can be terminated earlier by typing Ctrl+C). The figure shows the commands executed to instantiate the iperf3 session between h1 and h2, with duration of 3 seconds and bandwidth of 5Gbps.

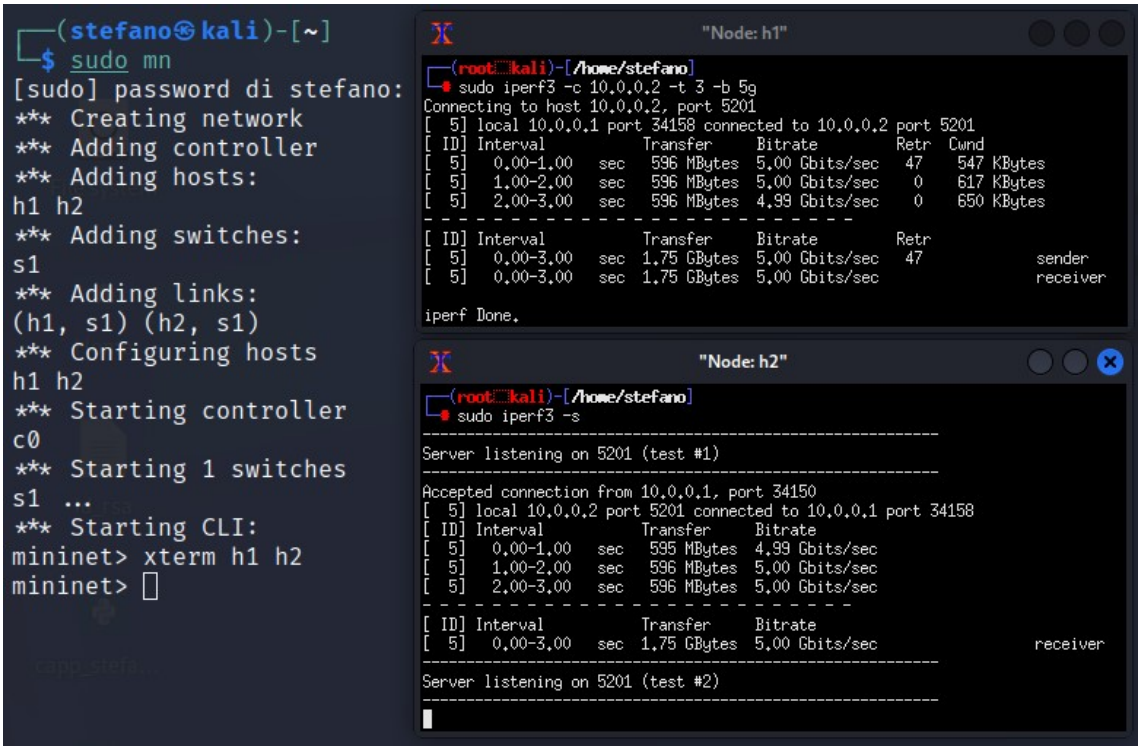


Figure 3.1: iPerf3 usage example

Another usage example of iperf3 is shown in 3.2, in which host h2 (in server mode) supports two iperf3 sessions simultaneously with hosts h1 and h3 (in client mode), while

listening on two different ports. This mode of using iPerf3 was used to simulate the DDoS attack to be detected.

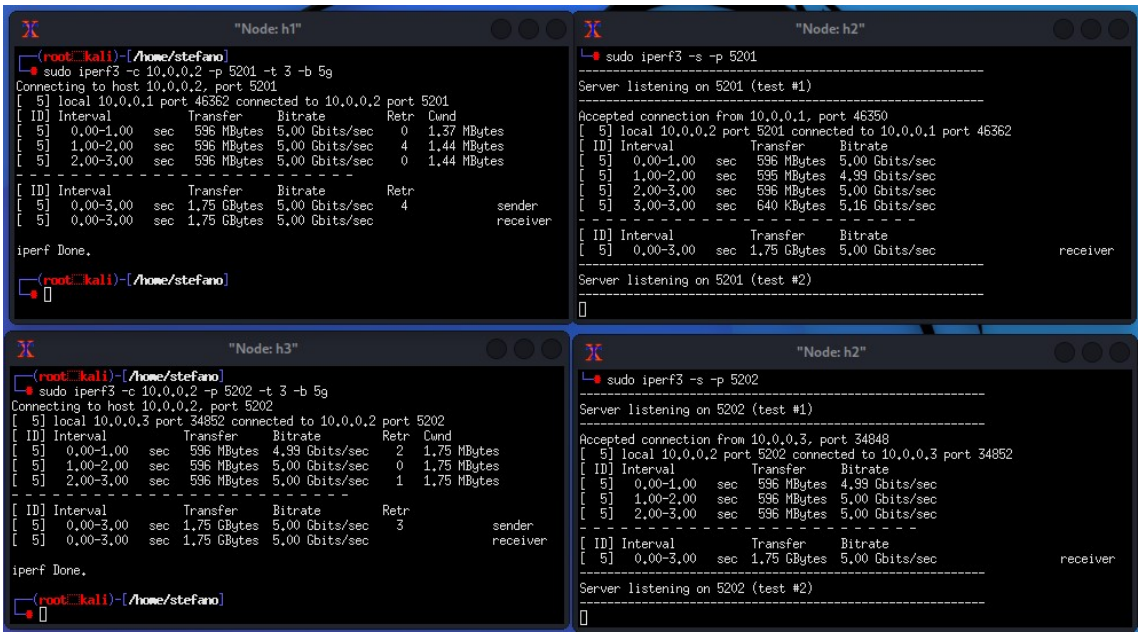


Figure 3.2: iPerf3 simultaneous sessions

CHAPTER 4

GeoIP

4.1 GeoIP

GeoIP refers to the process of finding a computer terminal's geographical location by determining the terminal's IP address. Even though it can identify the location of a terminal in a city, it requires utilization of GeoIP database and an understanding of APIs to execute in a right manner[7].

4.2 Download the Database

In order to download the Database the Maxmind service is used. MaxMind is a site that provides IP intelligence through the GeoIP brand. MaxMind's GeoIP2 Enterprise database provides contextual data for a comprehensive profile of IP addresses, including:

- geolocation data such as country, region, state, city, ZIP/postal code
- additional intelligence such as confidence factors, ISP, domain, and connection type

Before downloading the DB it is required to register to the Maxmind page with an email and a password. In our project we used the 'GeoLite2 City' database, in order to obtain it, after registration, visit the 'Download GeoIP Databases' section and download the 'GeoLite2 City' zip file; then extract it in a folder, this process will store the file with the '.mmDB' extension in the same folder. After this process, the Reader object of the DB can locate and read the mmDB file.

4.3 IP Geolocation Usage

IP geolocation is inherently imprecise. Locations are often near the center of the population. Any location provided by a GeoIP2 database or web service should not be used to identify a particular address or household.

4.4 Installation

In order to install the geoip2 module for interact with the DB using Python, type:

```
$ pip3 install geoip2
```

If you are not able to use pip, you may also use from the source directory:

```
$ easy_install .
```

4.5 Database Usage

To use the database API, you first construct a geoip2.database.Reader using the path to the file as the first argument. After doing this, you may call the method corresponding to database type (e.g., city or country), passing it the IP address you want to look up.

- If the lookup succeeds, the method call will return a model class for the database method you called. This model in turn contains multiple record classes, each of which represents part of the data for the record
- If the request fails, the reader class throws an exception

4.6 Database Read Implementation

The code we developed 4.1 to read the DB and scan cities names and geonameIDs by passing the IP addresses we want to locate as parameter is shown below:

```
#!/bin/env python3
from geoip2 import *
import geoip2.database
reader=geoip2.database.Reader("/home/p4/Desktop/Originale/GeoLite2-City_20221004/GeoLite2-City.mmDB")
response1=reader.city("89.46.106.33") #Arezzo

response2=reader.city("31.28.27.50") #St. Petersburg
response3=reader.city("8.27.67.188") #Shangai
response4=reader.city("95.110.235.107") #Arezzo (sito comune di sulmona)
print(response1.city.name)

print(response1.country.name)
print(response1.city.geoname_id)
print(response2.city.geoname_id)
print(response3.city.geoname_id)
print(response4.city.geoname_id)
#print(response.country.geoname_id)
reader.close()
```

Listing 4.1: Python API to interact with GeoIP2 DB

The output will be as follows:

```
Arezzo
Italy
3182884
498817
1796236
3182884
```

4.7 GeoIP Database Reader

The class geoip2.database.Reader is the GeoIP2 database Reader object. Instances of this class provide a reader for the GeoIP2 database format. IP addresses can be looked up using the country and city methods. The basic API for this class is the same for every database. First, you create a reader object, specifying a file name or file descriptor. You then call the method corresponding to the specific database, passing it the IP address you want to look up. If the request succeeds, the method call will return a model class for the method you called. This model contains multiple record classes, each of which represents part of the data returned by the database. If the database does not contain the requested information, the attributes on the record class will have a None value. If the address is not in the database, an geoip2.errors.AddressNotFoundError exception will be thrown. If the database is corrupt or invalid, a maxmindDB.InvalidDatabaseError will be thrown. Main attributes and methods related to this class we used are:

- city(ip address: ipaddress.IPv6Address or ipaddress.IPv4Address]) → geoip2.models.City

  Get the City object from the IP address. Parameters: ip address – IPv4 or IPv6 address as a string. Returns: geoip2.models.City objects

- close() → None

  Closes the GeoIP2 database.

- country(ip address: ipaddress.IPv6Address or ipaddress.IPv4Address]) → geoip2.models.Country

  Get the Country object fromr the IP address. Parameters: ip address – IPv4 or IPv6 address as a string. Returns: geoip2.models.Country object

Many of the records returned by the GeoIP web services and databases include a geonameid field. This is the ID of a geographical feature (city, region, country, etc.) in the GeoNames database. In our project we used the Reader object to retrieve the geonameId and the city name from the IP address to check a potential spoofing attack.

CHAPTER 5

Scapy

5.1 Introduction to Scapy

Scapy is a Python program that enables the user to send, sniff and dissect and forge network packets[9]. This capability allows construction of tools that can probe, scan or attack networks. In other words, Scapy is a powerful interactive packet manipulation program. It 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. Scapy can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery. It can replace hping, arpspoof, arp-sk, arping, p0f and even some parts of Nmap, tcpdump, and tshark. Scapy also performs very well a lot of other specific tasks that most other tools can't handle, like sending invalid frames, injecting your own 802.11 frames, combining techniques (VLAN hopping+ARP cache poisoning, VOIP decoding on WEP encrypted channel, …), and many others. Scapy mainly does two things:

- sending packets
- receiving answers

You define a set of packets, it sends them, receives answers, matches requests with answers and returns a list of packet couples (request, answer) and a list of unmatched packets. This has the big advantage over tools like Nmap or hping that an answer is not reduced to (open/closed/filtered), but is the whole packet.

5.2 Fast packet design

Other tools stick to the program-that-you-run-from-a-shell paradigm. The result is an awful syntax to describe a packet. For these tools, the solution adopted uses a higher but less powerful description, in the form of scenarios imagined by the tool's author. As an example, only the IP address must be given to a port scanner to trigger the port scanning scenario. Even if the scenario is tweaked a bit, you still are stuck to a port scan. Scapy's paradigm is to propose a Domain Specific Language (DSL) that enables a powerful and fast description of any kind of packet. Using the Python syntax and a Python interpreter as the DSL syntax and interpreter has many advantages: there is no need to write a separate interpreter, users don't need to learn yet another language and

they benefit from a complete, concise and very powerful language. Scapy enables the user to describe a packet or set of packets as layers that are stacked one upon another. Fields of each layer have useful default values that can be overloaded. Scapy does not force the user to use predetermined methods or templates. This alleviates the requirement of writing a new tool each time a different scenario is required. In C, it may take an average of 60 lines to describe a packet. With Scapy, the packets to be sent may be described in only a single line with another line to print the result. 90% of the network probing tools can be rewritten in 2 lines of Scapy.

5.3 Installing Scapy

The following steps describe how to install (or update) Scapy itself. Dependent on your platform, some additional libraries might have to be installed to make it actually work. So please also have a look at the platform specific to understand how to install those requirements. The following steps apply to Unix-like operating systems (Linux, BSD, Mac OS X) use pip:

```
$ pip3 install scapy
```

For Windows operating systems refer to the path installation described on this link.

5.4 Starting Scapy

Scapy's interactive shell is run in a terminal session. Root privileges are needed to send the packets, so we're using sudo here:

```
$ sudo scapy -H
Welcome to Scapy (2.4.0)
>>>
```

On Windows, open a command prompt (cmd.exe) and make sure that you have administrator privileges:

```
C:\>scapy
Welcome to Scapy (2.4.0)
>>>
```

If you do not have all optional packages installed, Scapy will inform you that some features will not be available:

```
INFO: Can't import python matplotlib wrapper. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
```

The basic features of sending and receiving packets should still work, though.

5.5 Using scapy in tools

You can easily use Scapy in your own tools. Just import what you need and do it.

```
1     from scapy.all import *
```

5.6 Using Scapy in a Python Script

For this example, a new file example.py is created with the following lines:

```
1 #! /usr/bin/env python
2 from scapy.all import *
3 ip_layer = IP(dst="172.16.27.135")
4 icmp_layer = ICMP(seq=9999)
5 packet = ip_layer / icmp_layer
6 send(packet)
```

The first line specifies the use of the Python interpreter. The second line is used to import all Scapy packages. In this example will be shown the concatenation of only two layers to define a packet.The first is the IP layer where the destination IP will be listed; the second defined is the ICMP layer with specified sequence number of value 9999. In order to combine both layers, the / character can be used. Finally, the packet is sent:

Finally, it is time to run the Python script using:

```
1 sudo python3 example.py
```

5.6.1 Verify ICMP Echo Request

Wireshark tool can help verify that the sequence number 9999 is actually set in the ICMP echo request. Prior to running example.py, Wireshark was started and it captured the following. The sequence number can be seen in figure 5.1

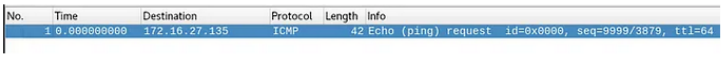


Figure 5.1: Wireshark Displaying ICMP Echo Request

The sequence number can also be verified by expanding the Internet Control Message Protocol section as seen in figure 5.2.

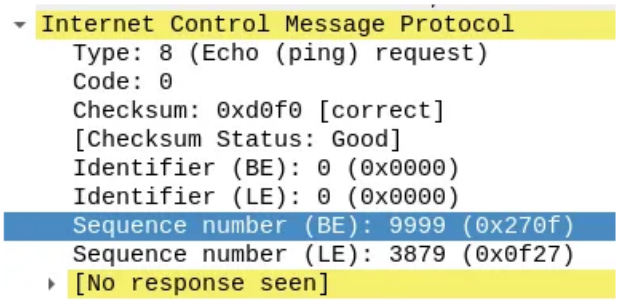


Figure 5.2: Sequence Number 9999 Set

CHAPTER 6

P4

6.1 Introduction

P4, which stands for Programming Protocol-independent Packet Processors, is a language for programming the data plane of network devices[2]. It is a language for expressing how packets are processed by the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance. In a traditional switch the manufacturer defines the data-plane functionality while the control-plane controls the data plane by managing entries in tables (e.g. routing tables), configuring specialized objects (e.g.meters), processing control-packets (e.g. routing protocol packets) and so on. A P4-programmable switch differs from a traditional switch in two essential ways:

- The data plane functionality is not fixed in advance but is defined by the a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program.
- The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. Once the P4 program has been compiled, the P4 compiler generates the API that the control plane uses to communicate with the data plane.

P4 can be said to be protocol independent, but it enables programmers to express a rich set of protocols and other data plane behaviors, just by writing their own P4 programs. P4 programs also partially define the interface by which the control plane and the data plane communicate. P4 cannot be used to describe the control-plane functionality of the target, being designed to specify only its data plane functionality. The two types of switches are compared in figure 6.1. In the rest of this document we will use the term target to refer to a generic packet-processing system capable of executing a P4 program.

6.2 Motivation

Since the explosive growth of the Internet in the 1990s, the networking industry has been dominated by closed and proprietary hardware and software. The progressive reduction

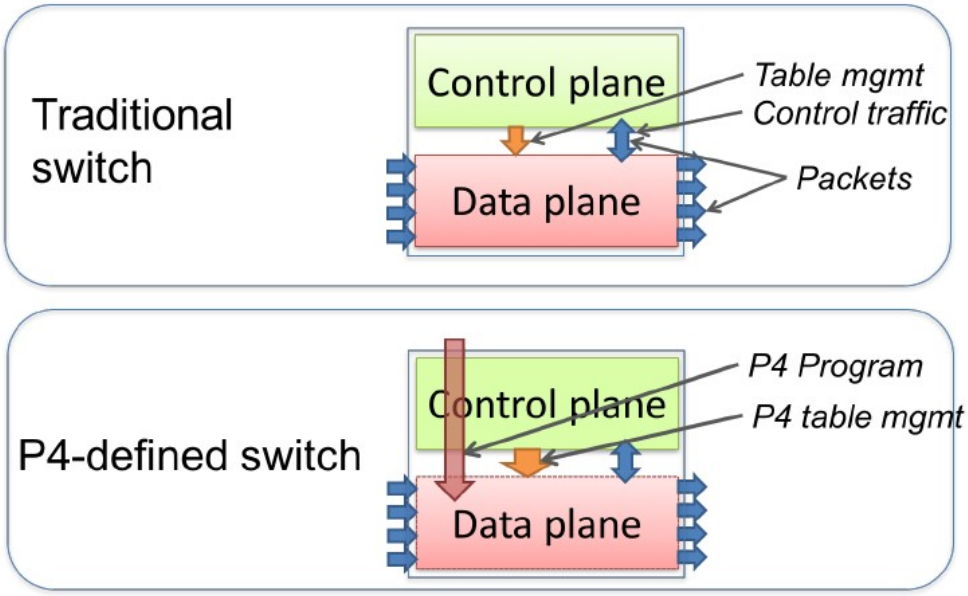


Figure 6.1: Traditional switches vs. programmable switches

in the flexibility of protocol design caused by standardized requirements, which cannot be easily removed to enable protocol changes, has led to the current situation. This protocol ossification has been characterized by a slow innovation pace at the hand of few network vendors. For this reason, the Application Specific Integrated Circuit (ASIC) implementation of a new protocol that has just been conceived, can take several years, instead of a few weeks if it were implemented via software. The design cycle of switch ASICs has been characterized by a long, closed, and proprietary process that usually takes years, in contrast to the the agility of the software industry. The programmable data-plane can be viewed as an evolution of Software-Defined Networking (SDN), where the software that describes the behavior of how packets are processed, can be conceived, tested, and deployed in a much shorter time span by operators, engineers, researchers, and practitioners in general. The de-facto standard for defining the forwarding behavior is the P4 language. We could say that P4 programmable switches have removed the entry barrier to network design, previously reserved to network vendors.

6.3 Programming a target with P4

In the P4 scenario, manufacturers provide the hardware or software implementation framework, an architecture definition, and a P4 compiler for that target. On the other hand, P4 programmers write programs for a specific architecture, which defines a set of P4- programmable components on the target as well as their external data plane interfaces. The compiler maps the target-independent P4 source code (P4 program) to the specific platform and after a P4 program has been compiled, it produces two artifacts:

- A data plane configuration that implements the forwarding logic described in the input program
- An API for managing the state of the data plane objects from the control plane

The data plane configuration (Data plane runtime) implements the forwarding logic specified in the P4 program given as input to the compiler, and it includes the instructions and resource mappings for the target. On the other hand, APIs generated by the compiler contain the information needed by the control plane to manage tables and objects in the data plane, such as the identifiers of the tables, fields used for matches, keys, action parameters, and others. The figure 6.2 shows the workflow of a P4 program.

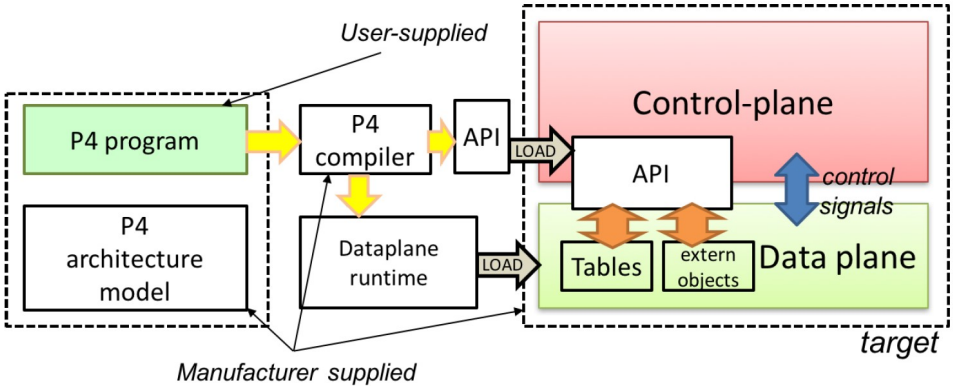


Figure 6.2: Workflow of a P4 program

6.4 Architecture Model

The P4 architecture identifies the P4-programmable blocks (e.g., parser, ingress control flow, egress control flow, deparser, etc.) and their data plane interfaces. The P4 architecture can be thought as a contract between the program and the target and each manufacturer must provide both a P4 compiler as well as an accompanying architecture definition for their target. Figure 6.3 illustrates the data plane interfaces between P4-programmable blocks and shows a target that has two programmable blocks (#1 and #2). Each block is programmed through a separate fragment of P4 code. The target interfaces with the P4 program through a set of control registers or signals. Input controls provide information to P4 programs (e.g., the input port that a packet was received from), while output controls can be written by P4 programs to influence the target behavior (e.g., the output port where a packet has to be directed). Control registers/signals are represented in P4 as intrinsic metadata. P4 programs can also store and manipulate data for each packet as user-defined metadata.

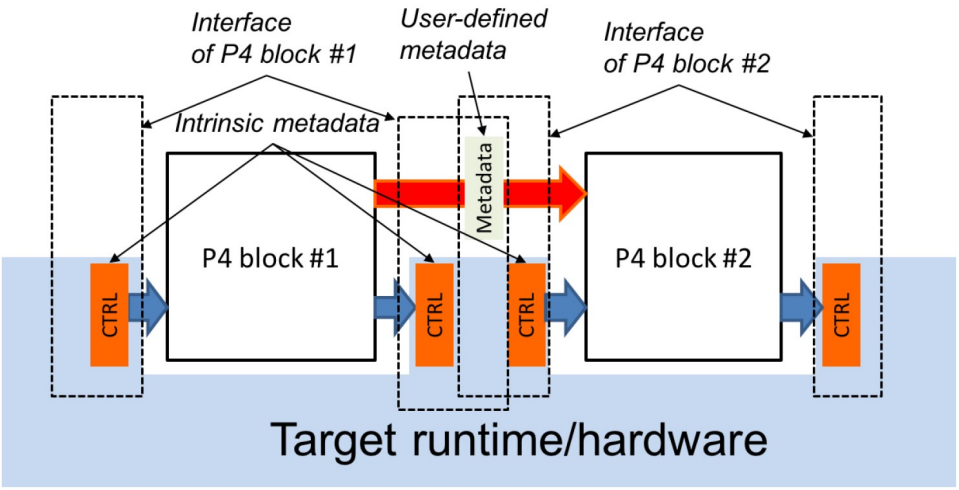


Figure 6.3: P4 program interfaces

The behavior of a P4 program can be fully described in terms of transformations that map vectors of bits to vectors of bits. In order to process a packet, the architecture model interprets the bits that the P4 program writes to intrinsic metadata. In general, P4 programs are not expected to be portable across different architectures, however, those that are written for a given architecture should be portable across all targets that faithfully implement the corresponding model, provided there are sufficient resources.

6.5 P4 language definition

In $P4_{16}$, the latest version of P4, a large number of language features have been eliminated from the language and moved into libraries including counters, checksum units, meters, etc. The language has been transformed from a complex language (more than 70 keywords) into a relatively small core language (less than 40 keywords) accompanied by a library of fundamental constructs that are needed for writing most P4. In this report we will only focus on $P4_{16}$ since it is the newer version and is currently being supported by major programming ASIC manufacturers. The P4 language can be viewed as having several distinct components, which we describe separately:

1. The core language, comprising of types, variables, scoping, declarations, statements, expressions, etc
2. A sub-language for expressing parsers, based on state machines
3. A sub-language for expressing computations using match-action units, based on traditional imperative controlflow
4. A sub-language for describing architectures

All these components will be briefly investigated in the tutorial section 6.7, in which a simple P4 program will be composed and explained step by step.

6.6 Benefits of P4

Compared to traditional packet-processing systems, P4 provides a number of significant advantages, such as:

- Flexibility: P4 makes many packet-forwarding policies expressible as programs, in contrast to traditional switches, which expose fixed-function forwarding engines to their users.
- Expressiveness: P4 can express sophisticated, hardware-independent packet processing algorithms using solely general-purpose operations and table look-ups. Such programs are portable across hardware targets that implement the same architectures.
- Software engineering: P4 programs provide important benefits such as type checking, information hiding, and software reuse.
- Component libraries: Component libraries supplied by manufacturers can be used to wrap hardware-specific functions into portable high-level P4 constructs.
- Decoupling hardware and software evolution: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing

6.7 A quick tutorial

The focus of this tutorial is to provide an overview of the general lifecycle of programming, compiling, and running a P4 program on a software switch. The following topics will be discussed in this demo:

1. The building blocks and general structure of a P4 program
2. The implementation of a simple parser that parses the defined headers
3. The match-action tables and how to define them in a P4 program
4. The checksum recalculation of a header and the deparsing process

In order to redo the tutorial, you can download all necessary software at the following link. The P4 program will be written for the V1Model architecture implemented on P4.org's bmv2 software switch. Visual Studio Code (VS Code) will be used as text editor to write the P4 program and the p4c compiler to compile the supplied P4 program (calc.p4).

6.7.1 Problem definition

The objective of this tutorial is to implement a basic calculator using a custom protocol header written in P4. The header will contain an operation to perform and two operands. When a switch receives a calculator packet header, it will execute the operation on the operands, and return the result to the sender.

6.7.1.1 Calculator implementation

To implement the calculator, the following steps have to be performed:

- Defining a custom calculator header
- Implementing the switch logic to parse header
- Performing the requested operation
- Returning the packet to the sender

The custom header is defined according to the scheme shown in figure 6.4.

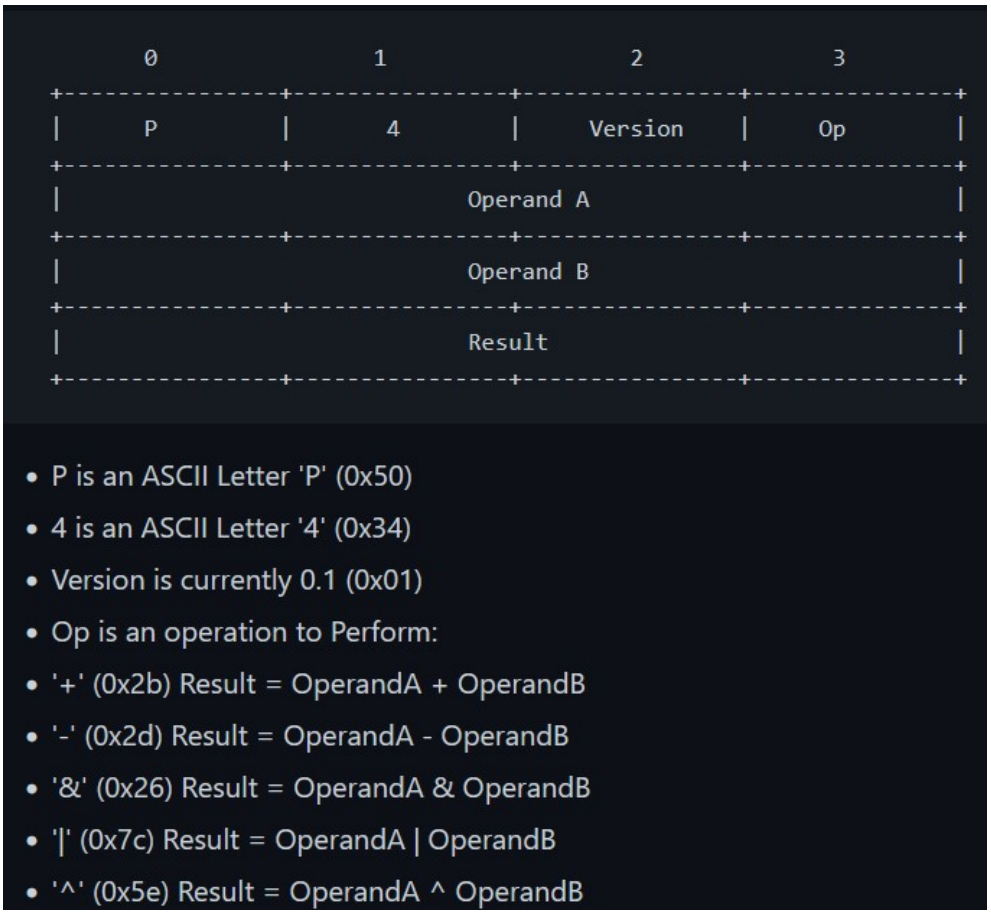


Figure 6.4: Custom calculation header

Operands A and B are positive numbers that can be written with 32 bits. Having 8 bits available for the operation identifier of the operations to be performed, we could have performed 2^8 different operations between the two operands. The reason why we allocated 8 bits, instead of just 3 bits, to the "Op" field is that BMv2 target only supports headers with fields totaling a multiple of 8 bits. The tasks to be performed by the P4 switch once a packet is received, are the following:

1. Performing the requested operation specified in the "Op" field of the Calculation header
2. Filling in the "Result" field with the result of the required operation
3. Sending the packet back out of the same port it came in on, by swapping the source and destination addresses

If an unknown operation is specified or the header is not valid, the packet will be dropped by the switch. As already said, Visual Studio Code has been used as a text editor. By importing the P4 extension, language keywords are highlighted and auto-fill with tab key is enabled, making writing and debugging code easier.

Import and custom header definition The code in figure 6.5, starts by including the core.p4 file (line 2) which defines some common types and variables declarations that are useful to most programs. The core library includes several common programming constructs and all P4 programs must include it. For example, the packet_in and packet_out extern types which represent incoming and outgoing packets, respectively, are declared in core.p4, used in parsers and deparsers to access packet data. Next, the v1model.p4 file is included (line 3) to define the V1Model architecture and all its externs used when writing P4 programs. Then, the program defines the Ethernet header (lines 10-14), depicted in figure 6.6

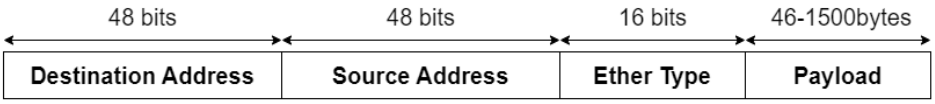


Figure 6.6: Ethernet header

and declares the 16-bit P4CALC_ETYPE constant, that will be used to fill in the etherType. This way Ethernet determines that the next header is "Calculation" if the value of EtherType is 0x1234. An equivalent method of defining the header is shown in figure 6.7 , by using the typedef keyword. The typedef declaration (line 1) is used to assign alternative names to types. In lines 29-37, the custom Calculation header previously shown in figure 6.4, is defined. Eight 8-bit constants are declared (lines 20-27), which will be used to fill in the Calculation header fields. After having defined the custom headers to be used in the program, they need to be assembled into a single struct, as depicted in figure 6.8.

calc.p4 M

home > p4 > Desktop > tutorials > exercises > calc > calc.p4

```
1  /* -*- P4_16 -*- */
2  #include <core.p4>
3  #include <v1model.p4>
4  /*
5   * Define the headers the program will recognize
6   */
7  /*
8   * Standard ethernet header
9   */
10 header ethernet_t {
11     bit<48> dstAddr;
12     bit<48> srcAddr;
13     bit<16> etherType;
14 }
15 /*
16  * This is a custom protocol header for the calculator. We'll use
17  * ethertype 0x1234 for is (see parser)
18  */
19 const bit<16> P4CALC_ETYPE = 0x1234;
20 const bit<8>  P4CALC_P     = 0x50;   // 'P'
21 const bit<8>  P4CALC_4     = 0x34;   // '4'
22 const bit<8>  P4CALC_VER   = 0x01;   // v0.1
23 const bit<8>  P4CALC_PLUS  = 0x2b;   // '+'
24 const bit<8>  P4CALC_MINUS = 0x2d;   // '-'
25 const bit<8>  P4CALC_AND   = 0x26;   // '&'
26 const bit<8>  P4CALC_OR    = 0x7c;   // '|'
27 const bit<8>  P4CALC_CARET = 0x5e;   // '^'
28
29 header p4calc_t {
30     bit<8>  p;
31     bit<8>  four;
32     bit<8>  ver;
33     bit<8>  op;
34     bit<32> operand_a;
35     bit<32> operand_b;
36     bit<32> res;
37 }
```

Figure 6.5: Import and custom headers definition

```
39
40 //All headers, used in the program needs to be assembed into a single struct.
41 struct headers {
42     ethernet_t   ethernet;
43     p4calc_t     p4calc;
44 }
45 struct metadata {
46     /* In our case it is empty */
47 }
```

Figure 6.8: Struct definition

The headers name will be used throughout the program when referring to the headers. Lines 45-47 show how to declare user-defined metadata, which are passed from one block

```
1  typedef bit<48> macAddress_t;
2  // Standard ethernet header
3   header ethernet_t {
4      macAddress_t dstAddr;
5      macAddress_t srcAddr;
6      bit<16> etherType;
7  }
```

Figure 6.7: Alternative header definition

to another as the packet propagates through the architecture. It is left blank, since this program does not require any user metadata.

Parser implementation A P4 parser describes a state machine with one start state and two final states, as depicted in figure 6.9. The start state is always named start while the two final states are named accept (indicating successful parsing) and reject (indicating a parsing failure). The programmable parser permits the programmer to describe how the switch will process the packet. It de-encapsulates the headers, converting the original packet into a parsed representation of the packet. A parser starts execution in the start state and ends execution when one of the reject or accept states has been reached. Each state has a name and a body. The second one consists of a sequence of statements that describes the processing performed when the parser transitions to that state, including transitions to other states, functions and methods invocation. Figure 6.10

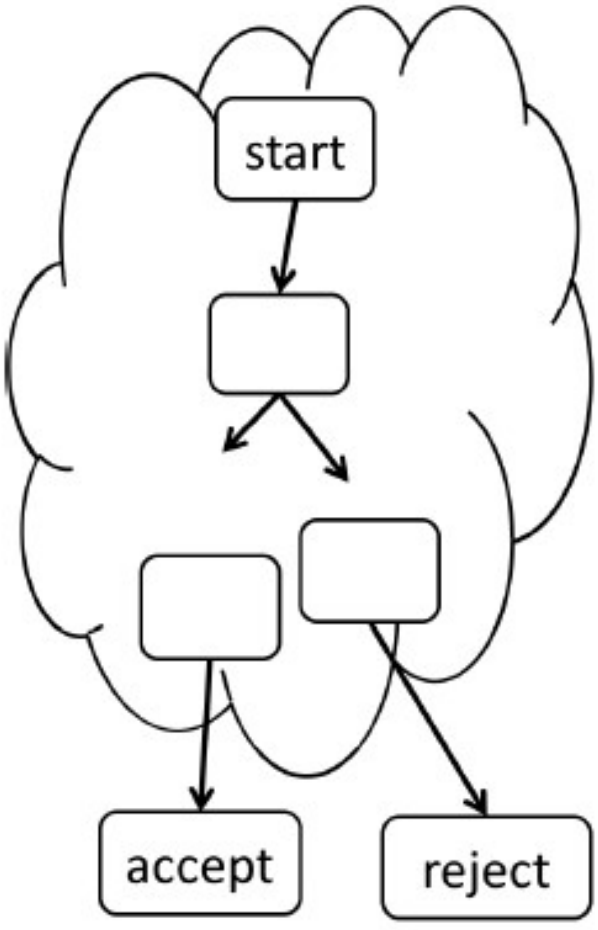


Figure 6.9: Parser FSM structure

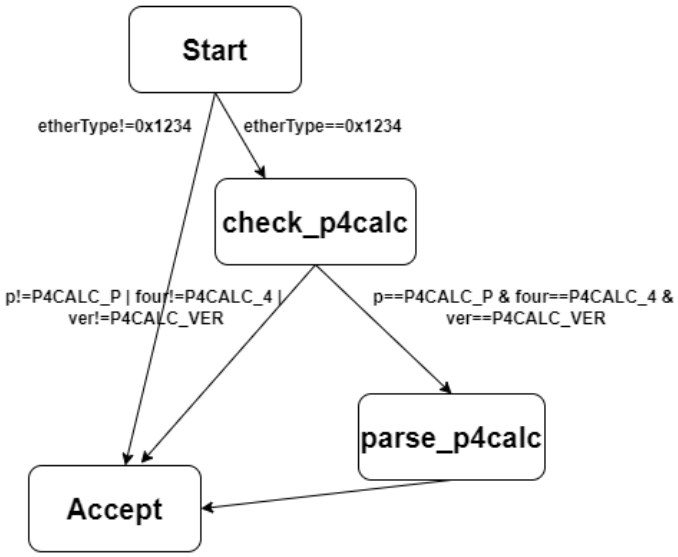


Figure 6.10: Graphical representation of the parser

depicts the graphical representation of the parser while figure 6.11

```
/*************************************************************************
**********************  P A R S E R  ******************************
*************************************************************************/
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            P4CALC_ETYPE : check_p4calc;
            default      : accept;
        }
    }

    state check_p4calc {
        transition select(packet.lookahead<p4calc_t>().p,
        packet.lookahead<p4calc_t>().four,
        packet.lookahead<p4calc_t>().ver) {
            (P4CALC_P, P4CALC_4, P4CALC_VER) : parse_p4calc;
            default                          : accept;
        }
    }

    state parse_p4calc {
        packet.extract(hdr.p4calc);
        transition accept;
    }
}
```

Figure 6.11: Parser implementation

shows the parser implementation for the basic calculator. In the start state, after extracting the Ethernet header through the extract() method, some conditions to direct the parser are created, until the accept state is reached. The extract() method associated with the packet extracts N bits, where N is the total number of bits defined in the corresponding header (for example, 112 bits for Ethernet). Afterwards, the etherType field of the Ethernet header is examined using the select statement, and the program branches to the state check_p4calc if the etherType field corresponds to P4CALC_ETYPE. The state transitions to accept if there is not a Calculation header. It may sound strange, but the reason we go to accept and not to reject is because explicit transition to reject is not supported on the target we are programming. Dropping the packet that does not contain Calculation header will be handled in a following block and not in the parser. Once check_p4calc state is reached, p, four and ver fields of the Calculation header are compared to P4CALC_P, P4CALC_4 AND P4CALC_VER respectively and simultaneously through the lookahead() method and the AND logical operator. Using the select statement, the program branches to parse_p4calc state if each of the three header Calculation fields previously compared matches the corresponding constant. The state transitions to the accept if the check fails i.e., when one or more header Calculation fields among p, four and ver, is not valid. Finally, in parse_p4calc state, the Calculation header is extracted, and the program unconditionally transitions to the accept state. It is important to specify that a header also contains a hidden Boolean “validity” field, set to false automatically when the header is created. When the “validity” bit is true we say that the “header is valid” and this bit can be manipulated by using the header

methods isValid(), setValid(), and setInvalid(). It is also assigned the value ”true” on a successful extract() method call in the parser. This validity bit will be used to determine when to drop a packet that does not contain the Calculation header in the next block, implementing essentially the same principle as the reject state that is not available for this target.

MyVerifyChecksum() and MyComputeChecksum() control blocks Scrolling down the script, two empty control blocks 6.12 are encountered:

- MyVerifyChecksum()
- MyComputeChecksum()

```
80  /*************************************************************************
81   ************   C H E C K S U M    V E R I F I C A T I O N   ************
82   *************************************************************************/
83  control MyVerifyChecksum(inout headers hdr,
84                           inout metadata meta) {
85      apply {}
86  }
87
88  /*************************************************************************…
91  control MyIngress(inout headers hdr,…
164 }
165
166 /*************************************************************************…
169 control MyEgress(inout headers hdr,…
173 }
174
175 /*************************************************************************
176  *************   C H E C K S U M    C O M P U T A T I O N   *************
177  *************************************************************************/
178
179 control MyComputeChecksum(inout headers hdr, inout metadata meta) {
180     apply { }
181 }
```

Figure 6.12: MyVerifyChecksum() and MyComputeChecksum()

The first control block can be used to verifiy the checksum for the packet by applying the verify_checksum() extern. It verifies the checksum of the supplied data and if detects that the checksum is not correct, then the value of the standard_metadata checksum_error field will be equal to 1 when the packet begins ingress processing. The second control block can be used to update the checksum of the packet, by applying the update_checksum() extern. In this tutorial, they are left empty for simplicity.

MyIngress() control block In myIngress() control block, packet processing is actually done. Before explaining in detail how a packet is processed in this control block, it is useful to first introduce the concepts of:

- Control Block: previously extracted headers (and other metadata) can be manipulated and transformed within control blocks. The body of a control block

resembles a traditional imperative program. Within the body of a control block, match-action units can be invoked to perform data transformations. Match-action units are represented in P4 by constructs called tables.

- Action: actions are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control-plane at runtime. Actions may contain data values that can be written by the control plane and read by the data plane. For this reason they are the main construct by which the control plane can influence dynamically the behavior of the data plane, as depicted in figure 6.13.

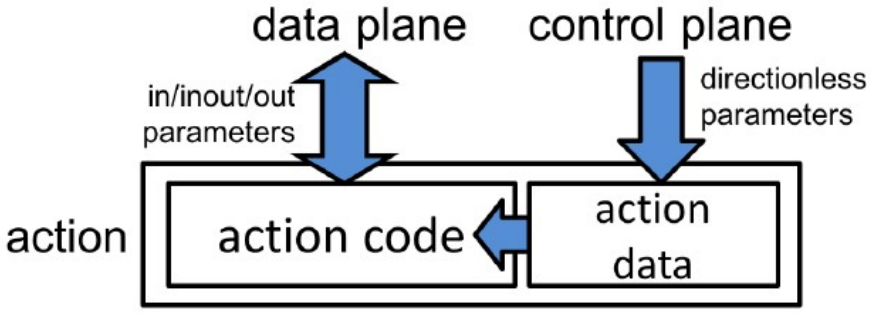


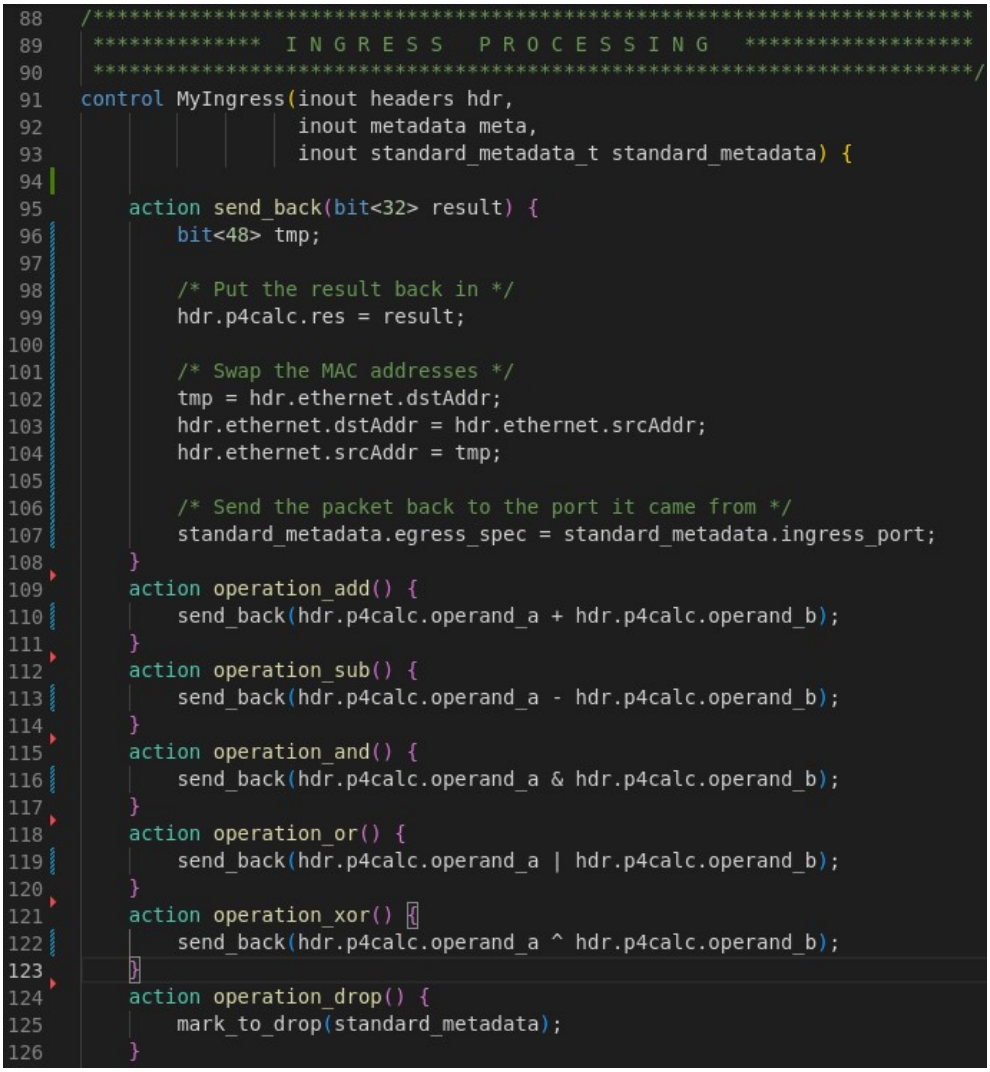
Figure 6.13: Action Working Principle

  They resemble functions with no return value and can be viewed as simple operations such as modify a header field, forward the packet to an egress port, and so on.

- Table: P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types. They are read-only for the data plane, but their entries can be modified by the control-plane through a control-plane API. A table has one or more entries all containing a key, an action, and action data.

  1. Key: it specifies the data plane values that should be used to look up an entry. A key is a list of pairs of the form (e : m), where e is an expression that describes the data to be matched in the table, and m is a match_kind constant that describes the algorithm used to perform the lookup.
  2. Action: once a match occurs, the action specified in the entry is performed by the arithmetic logic unit (ALU).
  3. Action data: it can be considered as parameter/s used by the action. For instance, the action data may represent the destination MAC address the switch must use to forward the packet.

After having introduced these concepts, let us see their implementation in the exercise we started with. As shown in figure 6.14,



```
88  /*************************************************************************
89   *************  I N G R E S S   P R O C E S S I N G   ********************
90   *************************************************************************/
91  control MyIngress(inout headers hdr,
92                    inout metadata meta,
93                    inout standard_metadata_t standard_metadata) {
94
95      action send_back(bit<32> result) {
96          bit<48> tmp;
97
98          /* Put the result back in */
99          hdr.p4calc.res = result;
100
101         /* Swap the MAC addresses */
102         tmp = hdr.ethernet.dstAddr;
103         hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
104         hdr.ethernet.srcAddr = tmp;
105
106         /* Send the packet back to the port it came from */
107         standard_metadata.egress_spec = standard_metadata.ingress_port;
108     }
109     action operation_add() {
110         send_back(hdr.p4calc.operand_a + hdr.p4calc.operand_b);
111     }
112     action operation_sub() {
113         send_back(hdr.p4calc.operand_a - hdr.p4calc.operand_b);
114     }
115     action operation_and() {
116         send_back(hdr.p4calc.operand_a & hdr.p4calc.operand_b);
117     }
118     action operation_or() {
119         send_back(hdr.p4calc.operand_a | hdr.p4calc.operand_b);
120     }
121     action operation_xor() {
122         send_back(hdr.p4calc.operand_a ^ hdr.p4calc.operand_b);
123     }
124     action operation_drop() {
125         mark_to_drop(standard_metadata);
126     }
```

Figure 6.14: MyInress() implementation, part 1

in MyIngress() control block the actions that will be called are immediately defined. The send_back() action first inserts the result passed as input into the ”res” field of the Calculation header; then swaps the source and destination MAC addresses; finally, through the instruction in line 107, it forces the packet to leave the switch from the port it came from. Then, the actions that perform the different operations between the two operands are defined. They simply call the previously defined send_back() action, passing as input the result of the corresponding arithmetic operation performed between the operands, to be entered in the res field of the Calculation header. The last defined action, operation_drop(), causes the packet to be dropped at the end of the ingress processing, by invoking the mark_to_drop primitive. In the second part of MyIngress() control block, shown in figure 6.15,

```
127     table calculate {
128         key = {
129             hdr.p4calc.op        : exact;
130         }
131         actions = {
132             operation_add;
133             operation_sub;
134             operation_and;
135             operation_or;
136             operation_xor;
137             operation_drop;
138         }
139         const default_action = operation_drop();
140         const entries = {
141             P4CALC_PLUS : operation_add();
142             P4CALC_MINUS: operation_sub();
143             P4CALC_AND  : operation_and();
144             P4CALC_OR   : operation_or();
145             P4CALC_CARET: operation_xor();
146         }
147     }
148     apply {
149         if (hdr.p4calc.isValid()) {
150             calculate.apply();
151         } else {
152             operation_drop();
153         }
154     }
155 }
```

Figure 6.15: MyIngress() implementation, part 2

table ”calculate” is defined as first step. The code inserted in lines 128-130 specifies that the ”op” field of a packet (hdr.p4calc.op) will be used as a key in the table. The match type is exact, denoting that the value of the ”op” field in the Calculation header will be matched as is against a constant among those in lines 23-27 of figure 6.5. It is worth mentioning that there exist three different match_kind types by standard (exact, Longest Prefix match, ternary), but architectures may define and implement additional ones. All the possible actions that will be used in this table are listed in lines 131-138. In line 139 operation_drop() is defined as a default action by using the default_action keyword, that specifies which default action has to be invoked whenever there is a miss. In the same line also appears the keyword const. A property marked as const cannot be changed dynamically by the control-plane. The key, actions, and size properties are always constant, so the keyword const is not needed for these. Scrolling down the script there is the entries declaration, which again uses the const keyword. While table entries are typically installed by the control plane, tables may also be initialized at compile-time with a set of entries, like in this example. This is useful in situations where tables are used to implement fixed algorithms, like in our case. Entries declared in the P4 source are installed in the table when the program is loaded onto the target and they are immutable (const) i.e., they can only be read and cannot be changed or removed by the control plane. In lines 148-154 the table calculate is called (calculate.apply()) only if Calculation header is valid (if (hdr.p4calc.isValid()), otherwise the packet is dropped. Remember that the header validity is set if the parser successfully parsed that header.

MyEgress() control block Packet processing in MyEgress() control block is not required to fulfill the purpose of this tutorial. Therefore, as seen in figure 6.16, it is left empty and no action is applied to the packet in this pipeline stage. Notice that the apply block, even though it is empty, it is required in every control block, otherwise the program will not compile.

```
166 /*************************************************************************
167  ***************  E G R E S S   P R O C E S S I N G   *******************
168  *************************************************************************/
169 control MyEgress(inout headers hdr,
170                  inout metadata meta,
171                  inout standard_metadata_t standard_metadata) {
172     apply { }
173 }
```

Figure 6.16: MyEgress() implementation

MyDeparser() control block The programmable deparser assembles the packet headers back and serializes them for transmission. The programmer specifies the headers to be emitted by the deparser, which emits the specified headers followed by the original payload of the packet when assembling the packet. We could therefore say that the parser describes how headers are emitted from the switch. P4 does not provide a separate language for packet deparsing which is done in a control block that has at least one parameter of type packet_out. The packet_out datatype defined in the P4 core library, provides a method for appending data to an output packet called emit. Figure 6.17 shows two instructions that reassemble the packet, by calling the emit() method.

```
183 /*************************************************************************
184  ***********************  D E P A R S E R  *******************************
185  *************************************************************************/
186 control MyDeparser(packet_out packet, in headers hdr) {
187     apply {
188         packet.emit(hdr.ethernet);
189         packet.emit(hdr.p4calc);
190     }
191 }
```

Figure 6.17: MyDeparser() implementation

Programming the pipeline sequence Figure 6.18 shows how to write the pipeline sequence in the calc.p4 program. In order to define the pipeline sequence according to the V1Model architecture that has been used, depicted in figure 6.19, it is sufficient to write those lines of code at the end of the P4 file. They specify the parser, the checksum verification block, the ingress and egress blocks, the checksum recomputation block and finally the deparser, previously declared in the P4 program. The traffic manager shown in figure 6.19, not mentioned so far, basically schedules packets between input ports and output ports and performs packet replication (e.g., replication of a packet for multicasting).

```
193 /*************************************************************************
194  ***********************  S W I T C H  ***********************************
195  *************************************************************************/
196
197 V1Switch(
198 MyParser(),
199 MyVerifyChecksum(),
200 MyIngress(),
201 MyEgress(),
202 MyComputeChecksum(),
203 MyDeparser()
204 ) main;
```

Figure 6.18: Writing the pipeline sequence

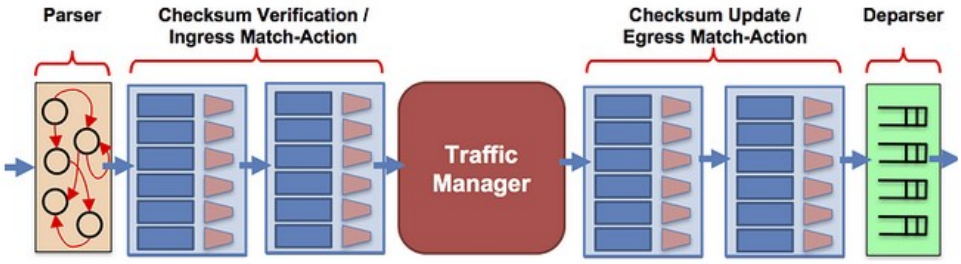


Figure 6.19: The V1Model architecture

6.7.1.2 Compiling and loading the P4 program & lab topology description

After completing the writing of the P4 program (calc.p4), typing the make command in your shell, will perform the following tasks:

1. Compile calc.p4
2. Start a Mininet instance with one switch (s1) connected to two hosts (h1, h2)
3. Configure the switch with the P4 program
4. The hosts are assigned IPs of 10.0.1.1 and 10.0.1.2

A JSON file (calc.json) and a p4info.txt file (calc.p4.p4info.txt) will be generated by the compiler. The first one will be used as the data plane program by the switch and the second one could be used to manage at runtime the behavior of the P4 target by the control plane. The network topology 6.20, including the ip configurations to be assigned to hosts, is described in the file topology.json, located in the same folder as calc.p4. As already mentioned we will assume that the calculator header is carried over Ethernet, and we will use the Ethernet type 0x1234 to indicate its presence. For this reason, the MAC addresses of hosts h1 and h2 are specified in the topology.json file as well.

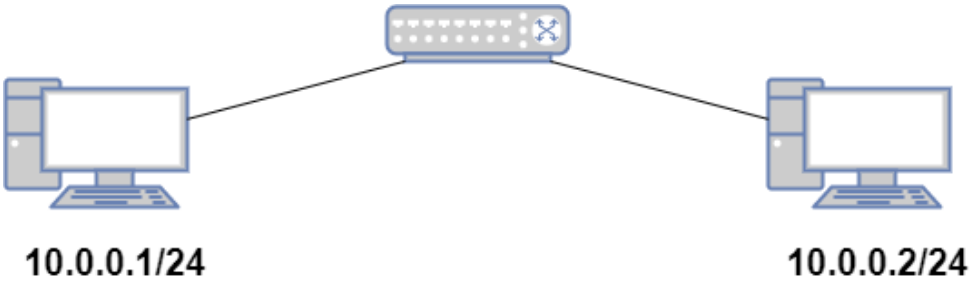


Figure 6.20: Tutorial Network Configuration

6.7.1.3 Testing and verifying the P4 program

In order to test the calculator, we used a small Python-based driver program (calc.py) located in the same folder as calc.p4 by running it directly from the Mininet command prompt, as shown in figure 6.21. Through the net command, the network is displayed.

```
p4@p4: ~/Desktop/tutorials/exercises/calc

================================================================================
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/Desktop/tutorials/exercises/calc/logs/<switchname>.log

To view the switch output pcap, check the pcap files in /home/p4/Desktop/tutorials/exercises/calc/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/Desktop/tutorials/exercises/calc/logs:
 for example run:  cat /home/p4/Desktop/tutorials/exercises/calc/logs/s1-p4runtime-requests.txt

mininet> net
h1 eth0:s1-eth1
h2 eth0:s1-eth2
s1 lo:  s1-eth1:eth0 s1-eth2:eth0
mininet> h1 python calc.py
> 123-12
123-12
111
> 10+15
10+15
25
> 10&10
10&10
10
> 10|10
10|10
10
```

Figure 6.21: Testing and verifying the P4 program

The driver program provides a new prompt, at which you can type basic expressions. It parses your expression, and prepares a packet with the corresponding operator and

operands. It sends then a packet to the switch for evaluation. When the switch returns the result of the computation, the test program prints the result, as depicted in figure 6.21. Notice that, since the switch has been programmed to process only packets that contain the Calculation header and drop all those that do not contain it (e.g. ip packets), the ping connectivity test returns a packet loss equal to 100%, as shown in figure 6.22

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
mininet>
```

Figure 6.22: Drop testing of other types of packets

6.7.1.4 Logs inspection

The folder where calc.p4 file is located contains the logs folder, where s1.log file is located. In this file the logs of switch s1 are contained. The figure 6.23 shows part of the processing logic as the packet enters switch s1. After the parsing is done, the packet is processed in the ingress and in the egress pipelines. Then, the deparser reassembles and emits the packet using port 1 (port_out: 1). This file can be very useful after writing the p4 program, to understand how a packet is processed and to perform debugging operations.

```
45  [16:27:52.778] [bmv2] [T] [thread 1754] [19.0] [cxt 0] Bytes parsed: 14
46  [16:27:52.778] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Parser 'parser' entering state 'parse_p4calc'
47  [16:27:52.778] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Extracting header 'p4calc'
48  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Parser state 'parse_p4calc' has no switch, going to default next state
49  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] Bytes parsed: 30
50  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Parser 'parser': end
51  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Pipeline 'ingress': start
52  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] calc.p4(158) Condition "hdr.p4calc.isValid()" (node_2) is true
53  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] Applying table 'MyIngress.calculate'
54  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Looking up key:
55  * hdr.p4calc.op       : 2b
56
57  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Table 'MyIngress.calculate': hit with handle 0
58  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Dumping entry 0
59  Match key:
60  * hdr.p4calc.op       : EXACT     2b
61  Action entry: MyIngress.operation_add -
62
63  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Action entry is MyIngress.operation_add -
64  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] Action MyIngress.operation_add
65  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] calc.p4(99) Primitive hdr.p4calc.res = result; ...
66  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] calc.p4(102) Primitive tmp = hdr.ethernet.dstAddr
67  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] calc.p4(103) Primitive hdr.ethernet.dstAddr = hdr.ethernet.srcAddr
68  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] calc.p4(104) Primitive hdr.ethernet.srcAddr = tmp
69  [16:27:52.779] [bmv2] [T] [thread 1754] [19.0] [cxt 0] calc.p4(107) Primitive standard_metadata.egress_spec = standard_metadata.ingress_port
70  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Pipeline 'ingress': end
71  [16:27:52.779] [bmv2] [D] [thread 1754] [19.0] [cxt 0] Egress port is 1
72  [16:27:52.779] [bmv2] [D] [thread 1756] [19.0] [cxt 0] Pipeline 'egress': start
73  [16:27:52.779] [bmv2] [D] [thread 1756] [19.0] [cxt 0] Pipeline 'egress': end
74  [16:27:52.779] [bmv2] [D] [thread 1756] [19.0] [cxt 0] Deparser 'deparser': start
75  [16:27:52.779] [bmv2] [D] [thread 1756] [19.0] [cxt 0] Deparsing header 'ethernet'
76  [16:27:52.779] [bmv2] [D] [thread 1756] [19.0] [cxt 0] Deparsing header 'p4calc'
77  [16:27:52.779] [bmv2] [D] [thread 1756] [19.0] [cxt 0] Deparser 'deparser': end
78  [16:27:52.779] [bmv2] [D] [thread 1759] [19.0] [cxt 0] Transmitting packet of size 31 out of port 1
```

Figure 6.23: Logs inspection

CHAPTER 7

P4 Runtime

7.1 Introduction

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].

7.2 Reference Architecture

Figure 7.1 represents the P4Runtime Reference Architecture.

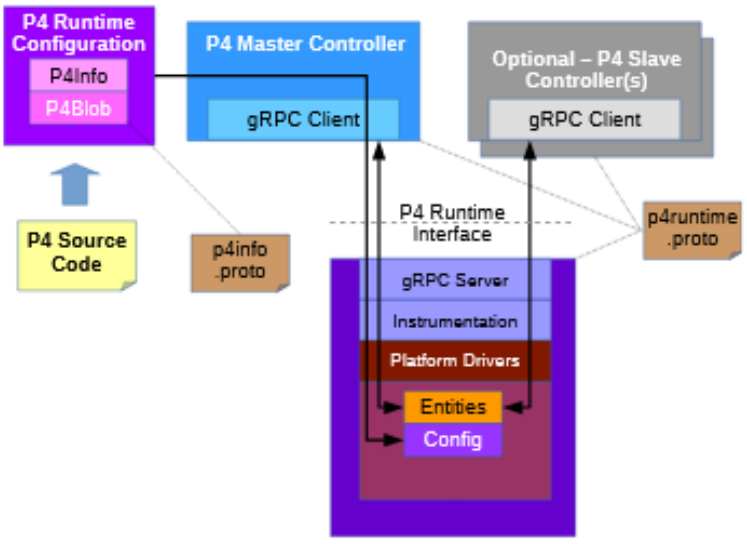


Figure 7.1: P4 Runtime Reference Architecture

The device or target to be controlled is at the bottom, and one or more controllers is shown at the top. P4Runtime only grants write access to a single primary controller for each read/write entity. A role defines a grouping of P4 entities. P4Runtime allows for a primary controller for each role, and a role-based client arbitration scheme ensures only one controller has write access to each read/write entity, or the pipeline config itself. Any controller may perform read access to any entity or the pipeline config. Later sections describe this in detail. For the sake of brevity, the term controller

may refer to one or more controllers. The P4Runtime API defines the messages and semantics of the interface between the client(s) and the server. The API is specified by the p4runtime.proto Protobuf file, which is available on GitHub as part of the standard Protobuf file. It may be compiled via protoc — the Protobuf compiler — to produce both client and server implementation stubs in a variety of languages. It is the responsibility of target implementers to instrument the server. Reference implementations of P4 targets supporting P4Runtime, as well as sample clients, may be available on the p4lang/PI GitHub repository [16]. A future goal may be to produce a reference gRPC server which can be instrumented in a generic way, e.g. via callbacks, thus reducing the burden of implementing P4Runtime. The controller can access the P4 entities which are declared in the P4Info metadata. The P4Info structure is defined by p4info.proto, another Protobuf file available as part of the standard. The controller can also set the ForwardingPipelineConfig, which amounts to installing and running the compiled P4 program output, which is included in the p4_device_config Protobuf message field, and installing the associated P4Info metadata. Furthermore, the controller can query the target for the ForwardingPipelineConfig to retrieve the device config and the P4Info.

7.3 P4Runtime Service Implementation

The P4Runtime API is implemented by a program that runs a gRPC[1] server which binds an implementation of auto-generated P4Runtime Service interface. This program is called the “P4Runtime server.” The server must listen on TCP port 9559 by default, which is the port that has been allocated by IANA for the P4Runtime service. Servers should allow users to override the default port using a configuration file or flag when starting the server. Uses of other port numbers as the default should be discontinued.

7.4 Idealized Workflow

In the idealized workflow, a P4 source program is compiled to produce both a P4 device config and P4Info metadata. These comprise the ForwardingPipelineConfig message. A P4Runtime controller chooses a configuration appropriate to a particular target and installs it via a SetForwardingPipelineConfig RPC. Metadata in the P4Info describes both the overall program itself (PkgInfo) as well as all entity instances derived from the P4 program — tables and extern instances. Each entity instance has an associated numeric ID assigned by the P4 compiler which serves as a concise “handle” used in API calls. In this workflow, P4 compiler backends are developed for each unique type of target and produce P4Info and a target-specific device config. The P4Info schema is designed to be target and architecture-independent, although the specific contents are likely to be architecture-dependent. The compiler ensures the code is compatible with the specific target and rejects code which is incompatible. In some use cases, it is expected that a controller will store a collection of multiple P4 “packages”, where

each package consists of the P4 device config and P4Info, and install them at will onto the target. A controller can also query the ForwardingPipelineConfig from the target via the GetForwardingPipelineRequest RPC. This can be useful to obtain the pipeline configuration from a running device to synchronize the controller to its current state.

7.5 Controller Use cases

P4Runtime allows for more than one controller. The mechanisms and semantics are described in section 8.3.1. Here we present a number of use-cases. Each use-case highlights a particular aspect of P4Runtime's flexibility and is not intended to be exhaustive. Real-world use-cases may combine various techniques and be more complex.

7.5.1 Single Embedded Controller

Figure 7.2 shows perhaps the simplest use-case. A device or target has an embedded controller which communicates to an on-board switch via P4Runtime. This might be appropriate for an embedded appliance which is not intended for SDN use-cases. P4Runtime was designed to be a viable embedded API. Complex controller architectures typically feature multiple processes communicating with some sort of IPC (Inter-Process Communications). P4Runtime is thus both an ideal RPC and an IPC.

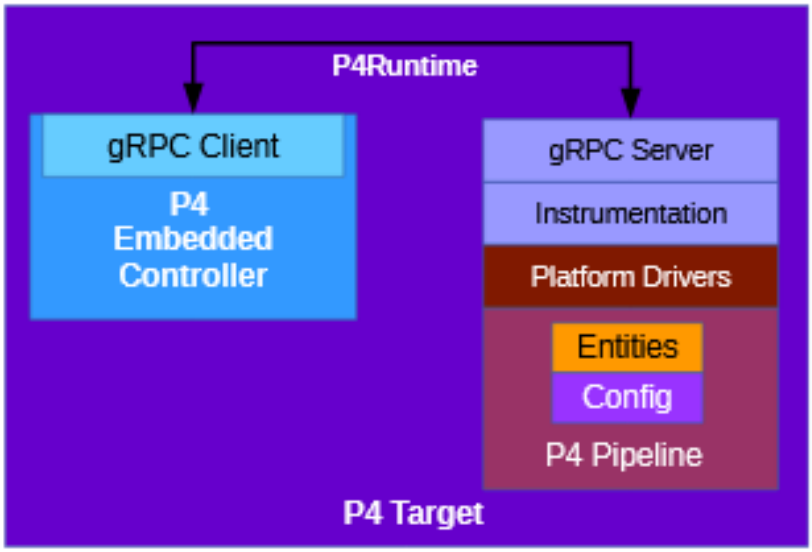


Figure 7.2: Use Case Single Embedded Controller

7.5.2 Single Remote Controller

Figure 7.3 shows a single remote Controller in charge of the P4 target. In this use-case, the device has no control of the pipeline, it just hosts the server. While this is possible, it is probably more practical to have a hybrid use-case as described in subsequent sections.

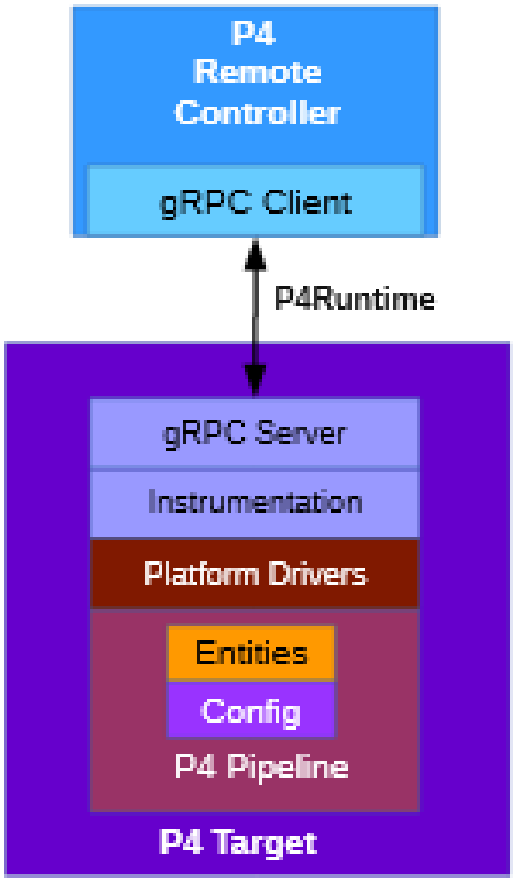


Figure 7.3: Single Remote Controller

7.5.3 Embedded plus Two Remote Controllers

Figure 7.4 illustrates the case of an embedded controller similar to the previous use-case, and two remote controllers. One of the remote controllers is responsible for some entities, e.g. routing tables, and the other remote controller is responsible for other entities, perhaps statistics tables. Role-based access divides the ownership.

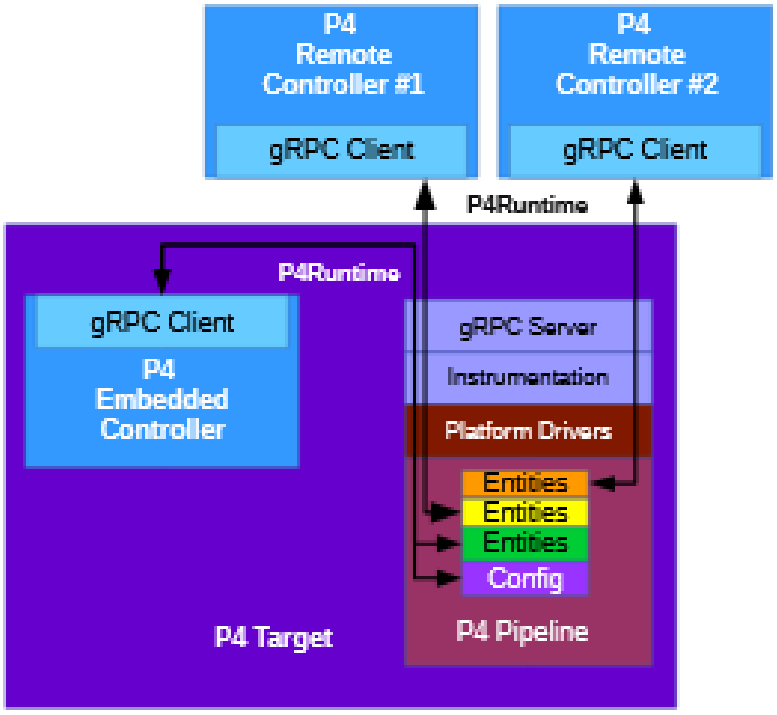


Figure 7.4: Use Case Embedded Plus Two Remote Controllers

Figure 7.5 illustrates a single embedded controller plus two remote controllers in an active-standby (i.e. primary-backup) HA (High-Availability) configuration. Controller #1 is the active controller and is in charge of some entities. If it fails, Controller #2 takes over and manages the tables formerly owned by Controller #1. The mechanics of HA architectures are beyond the scope of this report, but the P4Runtime role-based client arbitration scheme supports it.

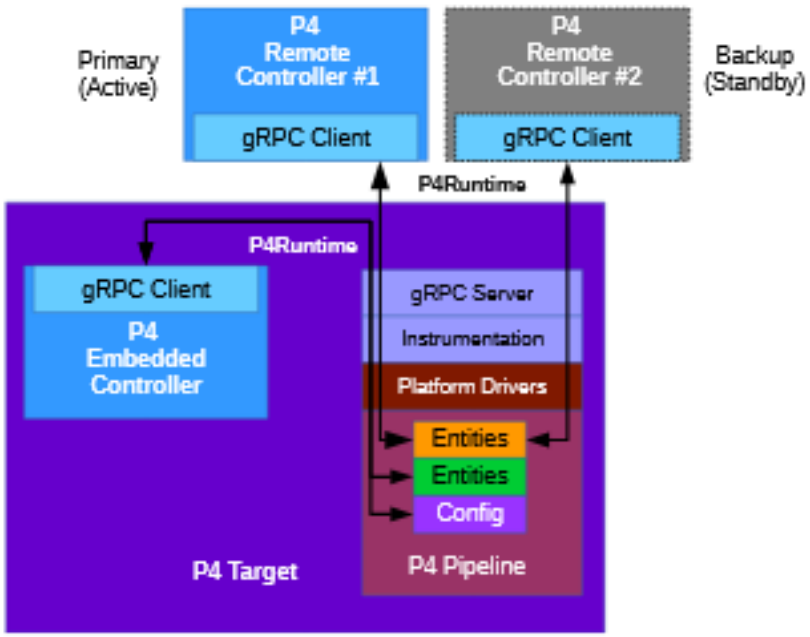


Figure 7.5: Use Case Embedded Plus Two Remote High Availability Controller

7.6 Client Arbitration and Controller Replication

The P4Runtime interface allows multiple clients (i.e. controllers) to be connected to the P4Runtime server running on the device at the same time for the following reasons:

- Partitioning of the control plane: Multiple controllers may have orthogonal, non-overlapping, “roles” (or “realms”) and should be able to push forwarding entities simultaneously. The control plane can be partitioned into multiple roles and each role will have a set of controllers, one of which is the primary and the rest are backups.
- Redundancy and fault tolerance: Supporting multiple controllers allows having one or more standby backup controllers. These can already have a connection open, which can help them become primary more quickly, especially in the case where the control-plane traffic is in-band and connection setup might be more involved.

To support multiple controllers, P4Runtime uses the streaming channel (available via StreamChannel RPC) for session management. The workflow is described as follows:

- Each controller instance (e.g. a controller process) can participate in one or more roles. For each (device_id, role), the controller receives an election_id. This

election_id can be the same for different roles and/or devices, as long as the tuple (device_id, role, election_id) is unique. For each (device_id, role) that the controller wishes to control, it establishes a StreamChannel with the P4Runtime server responsible for that device, and sends a MasterArbitrationUpdate message containing that tuple of (device_id, role, election_id) values. The P4Runtime server selects a primary independently for each (device_id, role) pair. The primary is the client that has the highest election_id that the device has ever received for the same (device_id, role) values. A connection between a controller instance and a device id, which involves a persistent StreamChannel, can be referred to as a P4Runtime client. Note that the P4Runtime server does not assign a role or election_id to any controller. It is up to an arbitration mechanism outside of the server to decide on the controller roles, and the election_id values used for each StreamChannel. The P4Runtime server only keeps track of the (device_id, role, election_id) of each StreamChannel that has sent a successful MasterArbitrationUpdate message, and maintains the invariant that all such 3-tuples are unique. A server must use all three of these values from a WriteRequest message to identify which client is making the WriteRequest, not only the election_id. This enables controllers to re-use the same numeric election_id values across different (device_id, role) pairs. P4Runtime does not require election_id values be reused across such different (device_id, role) pairs; it allows it.

- To start a controller session, a controller first opens a bidirectional stream channel to the server via the StreamChannel RPC for each device. This stream will be used for two purposes:
  - Session management: As soon as the controller opens the stream channel, it sends a StreamMessageRequest message to the switch. The controller populates the MasterArbitrationUpdate field in this message using its role and election_id, as well as the device_id of the device. Note that the status field in the MasterArbitrationUpdate is not populated by the controller. This field is populated by the P4Runtime server when it sends a response back to the client, as explained below.
  - Streaming of notifications (e.g. digests) and packet I/O: The same streaming channel will be used for streaming notifications, as well as for packet-in and packet-out messages. Note that unless specified otherwise by the role definitions, only the primary controller can participate in packet I/O.
- Note that a controller session is only required if the controller wants to do Packet I/O, or modify the forwarding state.
- Note that the stream is opened per device. In case a switching platform has multiple devices (e.g. multi-ASIC line card) which are all controlled via the same P4Runtime server, it is possible to have different primary clients for different devices. In this case, it is the responsibility of the P4Runtime server to keep track of the primary for each device (and role). More specifically, the P4Runtime server

will know which stream corresponds to the primary controller for each pair of (device_id, role) at any point of time.

- The streaming channel between the controller and the server defines the liveness of the controller session. The controller is considered "offline" or "dead" as soon as its stream channel to the switch is broken. When a primary channel gets broken: first an advisory message is sent to all other controllers for that device_id and role, as described in the following section, second the P4Runtime server will be without a primary controller, until a client sends a successful MasterArbitrationUpdate.

- The mechanism via which the controller receives the P4Runtime server details which includes the device_id, ip and port, as well as the mechanism via which it receives the Forwarding Pipeline Config, are implementation specific and beyond the scope of this report. Similarly, the mechanism via which the P4Runtime server receives its switch config (which notably includes the device_id) is beyond the scope of this report. Nevertheless, if the server details or switch config are transferred via the network, it is recommended to use TLS or similar encryption and authentication mechanisms to prevent eavesdropping attacks.

gRPC enables the server to identify which client originated each message in the StreamChannel stream. For example, the C++ gRPC library in synchronous mode enables a server process to cause a function to be called when a new client creates a StreamChannel stream. This function should not return until the stream is closed and the server has done any cleanup required when a StreamChannel is closed normally (or broken, e.g. because a client process unexpectedly terminated). Thus the server can easily associate all StreamChannel messages received from the same client, because they are processed within the context of the same function call. A P4Runtime implementation need not rely on the gRPC library providing information with unary RPC messages that identify which client they came from. Unary RPC messages include requests to write table entries in the data plane, or read state from the data plane, among others described later. P4Runtime relies on clients identifying themselves in every write request, by including the values device_id, role, and election_id in all write requests. The server trusts clients not to use a triple of values other than their own in their write requests. gRPC provides authentication methods that should be deployed to prevent untrusted clients from creating channels, and thus from making changes or even reading the state of the server.

7.7 Role Config

The role.config field in the MasterArbitrationUpdate message sent by the controller describes the role configuration, i.e. which operations are in the scope of a given role. In particular, the definition of a role may include the following:

- A list of P4 entities for which the controller may issue write updates and receive notification messages (e.g. DigestList and IdleTimeoutNotification).
- Whether the controller is able to receive PacketIn messages, along with a filtering mechanism based on the values of the PacketMetadata fields to select which PacketIn messages should be sent to the controller.
- Whether the controller is able to send PacketOut messages, along with a filtering mechanism based on the values of the PacketMetadata fields to select which PacketOut messages are allowed to be sent by the controller.

An unset role.config implies "full pipeline access". In order to support different role definition schemes, role.config is defined as an Any Protobuf message. Such schemes are out-of-scope of this document. When partitioning of the control plane is desired, the P4Runtime client(s) and server need to agree on a role definition scheme in an out-of-band fashion. It is the job of the P4Runtime server to remember the role.config for every device_id and role pair.

7.8 Rules for Handling MasterArbitrationUpdate Messages Received from Controllers

1 If the MasterArbitrationUpdate message is received for the first time on this particular channel (i.e. for a newly connected controller):

   1. If device_id does not match any of the devices known to the P4Runtime server, the server shall terminate the stream by returning a NOT_FOUND error.
   2. If the election_id is set and is already used by another controller for the same (device_id, role), the P4Runtime server shall terminate the stream by returning an INVALID_ARGUMENT error.
   3. If role.config does not match the "out-of-band" scheme previously agreed upon, the server must return an INVALID_ARGUMENT error.
   4. If the number of open streams for the given (device_id, role) exceeds the supported limit, the P4Runtime server shall terminate the stream by returning a RESOURCE_EXHAUSTED error.
   5. Otherwise, the controller is added to a list of connected controllers for the given (device_id, role) and the server remembers the controllers device_id, role and election_id for this gRPC channel. See below for the rules to determine if this controller becomes a primary or backup, and what notifications are sent as a consequence.

2. Otherwise, if the MasterArbitrationUpdate message is received from an already connected controller:

    1. If the device_id does not match the one already assigned to this stream, the P4Runtime server shall terminate the stream by returning a FAILED_PRECONDITION error.
    2. If the role does not match the current role assigned to this stream, the P4Runtime server shall terminate the stream by returning a FAILED_PRECONDITION error. If the controller wishes to change its role, it must close the current stream channel and open a new one.
    3. If role.config does not match the "out-of-band" scheme previously agreed upon, the server must return an INVALID_ARGUMENT error.
    4. If the election_id is set and is already used by another controller (excluding the controller making the request) for the same (device_id, role), the P4Runtime server shall terminate the stream by returning an INVALID_ARGUMENT error.
    5. If the election_id matches the one assigned to this stream:
        1. If the controller for this channel is the primary, then the server updates the role.config to the one specified in the MasterArbitrationUpdate. An advisory client arbitration message is sent to all controllers for this device_id and role informing them of the new role.config. Since the format of role.config is out of scope for the P4Runtime specification, the server will send the advisory message for every update, even if the primary sets the same role.config as it has before. See the following section for the format of the advisory message.
        2. If the controller is a backup, this is a no-op and the role.config is ignored. No response is sent to any controller.
    6. Otherwise, the server updates the election_id it has stored for this controller. This change might cause a change in the primary client (this controller might become primary, or the controller might have downgraded itself to a backup, see below), as well as notifications being sent to one or more controllers.

If the MasterArbitrationUpdate is accepted by either of the two steps above (cases 1.5. and 2.6. above), then the server determines if there are changes in the primary client. Let election_id_past be the highest election ID the server has ever seen for the given device_id and role (including the one of the current primary if there is one).

1. If election_id is greater than or equal to election_id_past, then the controller becomes primary. The server updates the role configuration to role.config for the given role.id. Furthermore:

1. If there was no primary for this device_id and role before and there are no Write requests still processing from a previous primary, then the server immediately sends an advisory notification to all controllers for this device_id and role. See the following section for the format of the advisory message.
2. If there was a previous primary or Write requests in flight, then the server carries out the following steps (in this order):
    1. The server stops accepting Write requests from the previous primary (if there is one). At this point, the server will reject all Write requests with PERMISSION_DENIED.
    2. The server notifies all controllers other than the new primary client of the change by sending the advisory notification described in the following section.
    3. The server will finish processing any Write requests that have already started. If there are errors, they are reported as usual to the previous primary. If the previous primary has already disconnected, any possible errors are dropped and not reported.
    4. The server now accepts the current controller as the new primary, thus accepting Write requests from this controller. The server updates the highest election ID (i.e. election_id_past) it has seen for this device_id and role to election_id.
    5. The server notifies the new primary by sending the advisory message described in the following section.

2. Otherwise, the controller becomes a backup. If the controller was previously a primary (and downgraded itself), then an advisory message is sent to all controllers for this device_id and role. Otherwise, the advisory message is only sent to the controller that sent the initial MasterArbitrationUpdate. See the following section for the format of the advisory message.

7.9 Client Arbitration Notifications

For any given device_id and role, any time a new primary is chosen, a primary downgrades its status to a backup, a primary disconnects, or the role.config is updated by the primary, all controllers for that (device_id, role) are informed of this by sending a StreamMessageResponse. The MasterArbitrationUpdate is populated as follows:

- device_id and role.id as given.
- role.config is set to the role configuration the server received most recently in a MasterArbitrationUpdate from a primary.
- election_id is populated as follows:

1 If there has not been any primary at all, the election_id is left unset.

2 Otherwise, election_id is set to the highest election ID that the server has seen for this device_id and role (which is the election_id of the current primary if there is any).

- status is set differently based on whether the notification is sent to the primary or a backup controller:

  If there is a primary:

  1 For the primary, status is OK (with status.code set to google.rpc.OK).

  2 For all backup controllers, status is set to non-OK (with status.code set to google.rpc.ALREADY_EXISTS).

  Otherwise, if there is no primary currently, for all backup controllers, status is set to non-OK.

7.10 Example of usage of P4Runtime

In our project we simulated a controller by using a python script that communicates with the switches. The controller acts like a master for switches and it communicates with them through the use of p4runtime library that is possible to import from python :

```
from p4.v1 import p4runtime_pb2
from p4.v1 import p4runtime_pb2_grpc
```

In the script 'switch.py' (available on Github at p4runtime_switch.py), it is defined a switch connection class that contains MasterArbitrationUpdate and SetForwarding-PipelineConfig functions that corresponds to the p4Runtime messages to build and install rules. In particular the method MasterArbitrationUpdate() builds a request to be sent to the switch with the method StreamMessageRequest() offered by p4_runtime_pb2 library; then the request is customized with information of the switch we want to control. Instead the SetForwardingPipelineConfig() sends the request built to the switch and the P4 program is installed on it.

```


    def MasterArbitrationUpdate(self, dry_run=False, **kwargs):
        request = p4runtime_pb2.StreamMessageRequest()
        request.arbitration.device_id = self.device_id
        request.arbitration.election_id.high = 0
        request.arbitration.election_id.low = 1

        if dry_run:
            print("P4Runtime MasterArbitrationUpdate: ", request)
        else:
```

```
            self.requests_stream.put(request)
            for item in self.stream_msg_resp:
                return item  # just one

    def SetForwardingPipelineConfig(self, p4info, dry_run=False, **kwargs)
        :
        device_config = self.buildDeviceConfig(**kwargs)
        request = p4runtime_pb2.SetForwardingPipelineConfigRequest()
        request.election_id.low = 1
        request.device_id = self.device_id
        config = request.config

        config.p4info.CopyFrom(p4info)
        config.p4_device_config = device_config.SerializeToString()

        request.action = (
            p4runtime_pb2.SetForwardingPipelineConfigRequest.
                VERIFY_AND_COMMIT
        )
        if dry_run:
            print("P4Runtime SetForwardingPipelineConfig:", request)
        else:
            self.client_stub.SetForwardingPipelineConfig(request)
```

In the script controller.py we use functions defined in the switch.py script in order to interact with switches. The first called method is Bmv2SwitchConnection() and it creates a switch connection object for s1; this method is supported by a P4Runtime gRPC connection and it dumps all P4Runtime messages sent to switch to a given txt file.

```
    s1 = bmv2.Bmv2SwitchConnection(
            name="s1",
            address="0.0.0.0:50051",
            device_id=1,
            proto_dump_file="p4runtime1.log",
        )
```

Send master arbitration update message to establish this controller as master (required by P4Runtime before performing any other write operation).

```
    if s1.MasterArbitrationUpdate() == None:
            print("Failed to establish the connection")
```

Install the P4 program on the switch.

```

    s1.SetForwardingPipelineConfig(
```

```
            p4info=p4info_helper.p4info, bmv2_json_file_path=
                bmv2_file_path
        )
    print("Installed P4 Program using SetForwardingPipelineConfig on s1")
```

After, we created a connection between the switch and the controller, it is now possible to write new rules to manage and modify the forwarding tables installed on the switch.

```


    writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.
        ipv4_lpm_for_ssh", dst_ip_addr="8.27.67.188", dst_eth_addr= "
        08:00:00:00:01:11",port=1)
        writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.
            ipv4_lpm_for_ssh", dst_ip_addr="31.28.27.50", dst_eth_addr= "
            08:00:00:00:00:02",port=10)
        writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.
            ipv4_lpm_for_ssh", dst_ip_addr="89.46.106.33", dst_eth_addr= "
            08:00:00:00:00:02",port=10)
        writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.
            ipv4_lpm_for_ssh", dst_ip_addr="95.110.235.107", dst_eth_addr=
            "08:00:00:00:00:04",port=11)
        writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.ipv4_lpm
            ", dst_ip_addr="8.27.67.188", dst_eth_addr= "08:00:00:00:01:11"
            ,port=1)
        writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm"
            , dst_ip_addr="31.28.27.50", dst_eth_addr= "08:00:00:00:00:02",
            port=10)
        writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm"
            , dst_ip_addr="89.46.106.33", dst_eth_addr= "08:00:00:00:00:02"
            ,port=10)
        writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm"
            , dst_ip_addr="95.110.235.107", dst_eth_addr= "
            08:00:00:00:00:04",port=11)
        writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
            8.27.67.188", port=1)
        writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
            31.28.27.50", port=10)
        writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
            89.46.106.33", port=10)
        writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
            95.110.235.107", port=11)
```

These steps are repeated for every P4 switch in the topology, but the correct port number must be passed to methods according to the network topology.

CHAPTER 8

Practical experiment

8.1 Readme

The objective of this tutorial is to write a P4 program that enables a P4 programmed switch to support the following features:

- It simply forwards all non-SSH incoming packets carried over IPV4 according to the content of the forwarding tables.
- It drops all SSH incoming packets that do not have a valid GSS header (custom header).
- It drops all SSH incoming packets that have a given geoname_id (custom header protocol field), inserted at runtime by the network controller in the firewall table.
- It forwards all SSH incoming packets that have a geoname_id, not present in the firewall table.
- It forwards all packets coming from the controller, i.e. all the packet-outs according to the forwarding rules.
- It forwards to the controller all SSH incoming packets that have a valid GSS header and an allowed geoname_id (i.e., all the SSH packets that have passed the firewall rules) whenever the controller has to perform spoof checks.

Our P4 program will be written for the V1Model architecture implemented on P4.org's bmv2 software switch. We will use the topology shown in 8.1 for this exercise. In order to redo the exercise, these simple steps need to be followed:

- Download the VM with preinstalled tools available here
- Enter in the p4 account and use the "git clone" command to clone the online repository available at the following link, after entering the "Dekstop" folder.
- Install geoip2 Python package 4.4
- Give the command "sudo python3 run_mininet.py –p4-file switch_config.p4" after entering the "P4_Project" folder contained in the previously cloned repository. This command will:

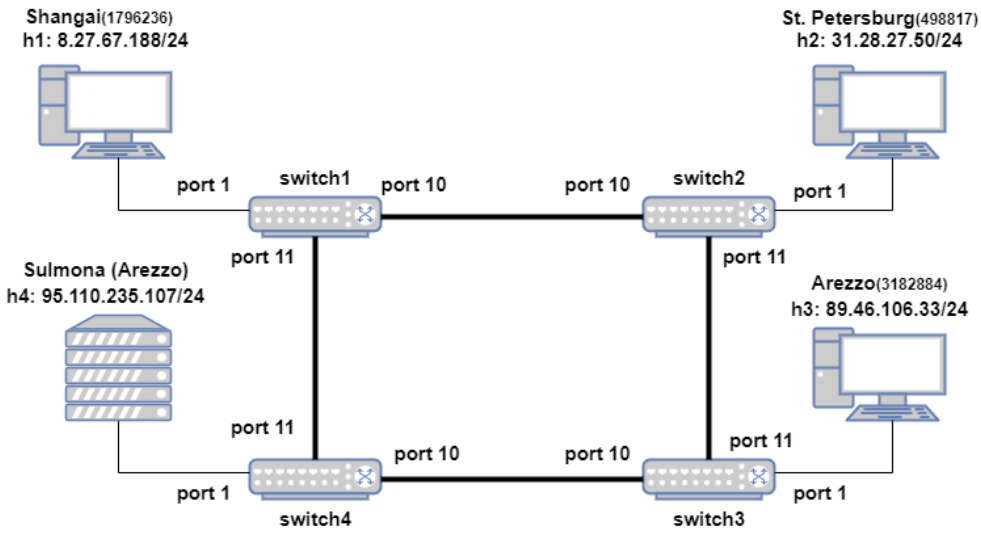


Figure 8.1: Network topology

1. compile switch_config.p4
2. start the topology shown in 8.1 in Mininet
3. configure all switches with the P4 program
4. configure all hosts with the commands listed in the file topology.json

- Give the command "sudo python3 controller.py" after entering the "P4_Project" folder contained in the previously cloned repository from another terminal. This command will start the network controller. The controller will:
  - configure all switches with the appropriate table entries
  - perform Dos or DDos detection (by simply reading a counter periodically)
  - perform spoofing check (on the geoname_id GSS field) when requested from the user
- Give the command pingall in the Mininet CLI to check if all the previous steps were successful

A note about the control plane A P4 program defines a packet-processing pipeline, but the rules within each table are inserted by the control plane. When a rule matches a packet, its action is invoked with parameters supplied by the control plane as part of the rule. We use P4Runtime to install the control plane rules.

8.2 Lab setup

8.2.1 How we used Mininet

To test the behavior of the network, consisting of switches configured via the P4 custom program, we set up a simple network consisting of four hosts and four switches. Each host is connected to a switch, and the switches are in turn connected via a ring topology. To build the network with a given topology, a given number of hosts and switches, we wrote a JSON file in which all these settings have been reported. Here you can specify the hosts, switches and links that connect them. Below the JSON file used to build the network used in this project is shown.

```
{
    "hosts": {
        "h1": {"ip": "8.27.67.188/24", "mac": "08:00:00:00:01:11","commands":["route add
            default gw 8.27.67.1 dev eth0",
                    "arp -i eth0 -s 8.27.67.1 08:00:00:00:00:01"]},
        "h2": {"ip": "31.28.27.50/24", "mac": "08:00:00:00:01:22","commands":["route add
            default gw 31.28.27.1 dev eth0",
                    "arp -i eth0 -s 31.28.27.1 08:00:00:00:00:02"]},
        "h3": {"ip": "89.46.106.33/24", "mac": "08:00:00:00:01:33","commands":["route add
            default gw 89.46.106.1 dev eth0",
                    "arp -i eth0 -s 89.46.106.1 08:00:00:00:00:03"]},
        "h4": {"ip": "95.110.235.107/24", "mac": "08:00:00:00:01:44","commands":["route add
            default gw 95.110.235.1 dev eth0",
                    "arp -i eth0 -s 95.110.235.1 08:00:00:00:00:04"]}
    },
    "switches": {
        "s1": { "name" : "s1", "grpc_port": "50051", "device_id": "1", "thrift_port": "9090"
            },
        "s2": { "name" : "s2", "grpc_port": "50052", "device_id": "2", "thrift_port": "9091"
            },
        "s3": { "name" : "s3", "grpc_port": "50053", "device_id": "3", "thrift_port": "9092"
            },
        "s4": { "name" : "s4", "grpc_port": "50054", "device_id": "4", "thrift_port": "9093"}
    },
    "links": [
      ["h1", "s1", 1], ["h2", "s2", 1], ["h3", "s3", 1], ["h4", "s4", 1],
       ["s1", "s2", 10, 10],["s2", "s3", 11, 11],["s3", "s4", 10, 10],["s4", "s1", 11, 11]
    ]
}
```

Listing 8.1: MultiSwitchTopo() class

As can be seen, in the JSON file, in addition to the name, the IPV4 and MAC addresses assigned to each host, there are also some configuration commands that are executed on each host once they are created. Moreover each switch is assigned a device id, grpc_port and thrift_port in addition to its name. The latter two parameters allow the switch to be controlled and reconfigured at runtime by using a network controller and by accessing the switch's CLI, respectively. Starting from the description of the network to be started in the JSON file, you can instantiate and use it in Mininet by exploiting the MultiSwitchTopo() class shown below.

```
class MultiSwitchTopo(Topo):
    "Single switch connected to n (< 256) hosts."
    def __init__(self, topo_file, sw_path, json_path, n, **opts):
        # Initialize topology and default options
```

```
        Topo.__init__(self, **opts)
        with open(topo_file, 'r') as f:
            topo = json.load(f)
        hosts = topo['hosts']
        switches = topo['switches']
        links=topo['links']
        for sw in switches:
            switch = self.addSwitch(
                switches[sw]["name"],
                sw_path=sw_path,
                json_path=json_path,
                grpc_port=switches[sw]["grpc_port"],
                thrift_port=int(switches[sw]["thrift_port"]),
                device_id=int(switches[sw]["device_id"]),
                cpu_port="255",
            )
        for h in hosts:
            host=self.addHost(h, ip=hosts[h]["ip"],mac =hosts[h]["mac"])
        for link in links:
            if len(link) == 3:
                host = link[0]
                switch = link[1]
                port=link[2]
                self.addLink(host, switch, port)
            if len(link) == 4:
                switch1 = link[0]
                switch2 = link[1]
                port1=link[2]
                port2=link[3]
```

Listing 8.2: MultiSwitchTopo() class

An object of class MultiSwitchTopo() is created and hosts and switches are added to it according to the configurations specified in the JSON file. Finally the links between hosts and switches and between switches and switches are added. To program the switches with P4 and reconfigure them at runtime using P4runtime, P4GrpcSwitch class was used. This choice allows you to specify the grpc port of the P4 switch through which it will be able to communicate with the controller at runtime. This class is reported in the appendix. In run_mininet.py script, also included in the appendix, the network is created, and it is started with the start() method. The network is configured with MultiSwitchTopo() topology and with the parameters specified in the JSON file. The network is shown in figure 8.2

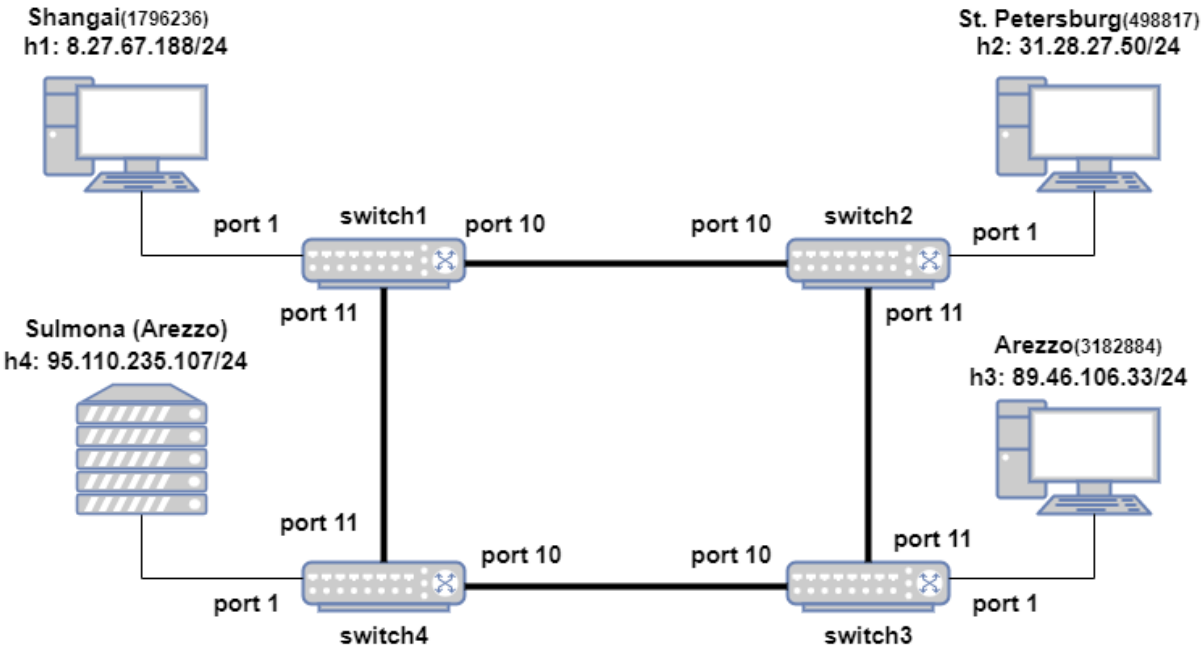


Figure 8.2: Network topology

For each switch, port 255 was set as cpu_port. In order to enable packet-in and packet-out operations, the cpu_port of the switch was used. This mechanism has been used whenever an incoming packet to the switch had to be analyzed by the controller before being forwarded by the switch.

8.3 P4 program description

8.3.0.1 Implementation

In order to ensure that the switch supports the required functionalities, the following steps have to be performed:

- Defining a custom header
- Implementing the switch logic to parse headers
- Performing ipv4 packet forwarding according to firewall and forwarding rules
- Communicating with the network controller in order to detect spoofing attacks (when this check is enabled)
- Instantiating and updating the counter used to detect the Dos or DDos attack

GSS custom header, which stands for Geoname Secure Service, consists of just one field, geoname_id, that is 24 bits long. This protocol field is sufficient to achieve the goal of locating an IPV4 address. GSS header is carried over Ethernet, and we will use the Ethernet type 0x1212 (TYPE_GSS) to indicate its presence in the parsing process. A packet that is destined to TCP port # 22, in order to not be dropped by the firewall rules implemented by the switch, must be formatted as shown in figure 8.3.

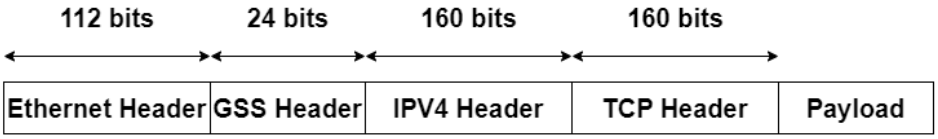


Figure 8.3: GSS packet structure

In lines 1-60 8.4,

```
1  /* -*- P4_16 -*- */
2
3  #include <core.p4>
4  #include <v1model.p4>
5
6  const bit<16> TYPE_IPV4 = 0x800;
7  const bit<8>  TYPE_TCP  = 0x6;
8  const bit<16> SSH_DEFAULT_PORT = 0x16;
9  const bit<16> TYPE_GSS = 0x1212;
10 #define CPU_PORT 255
11 /*************************************************************************
12 ********************** H E A D E R S  ***********************************
13 *************************************************************************/
14
15 typedef bit<9>  egressSpec_t;
16 typedef bit<48> macAddr_t;
17 typedef bit<32> ip4Addr_t;
18
19 > header ethernet_t {…
23 }
24 header GSS_t {
25     bit<24> geoname_id;
26 }
27
28 > header ipv4_t {…
41 }
42
43 > header tcp_t{…
61 }
```

Figure 8.4: Imports, and variables and headers definition

after the usual imports, Ethernet, GSS, IPV4, and TCP headers are defined. In lines 66-75 packet_in and packet_out headers are defined and are annotated with @controller_header annotation. ControllerPacketMetadata messages are used to describe any

metadata associated with controller packet-in and packet-out. A packet-in is defined as a data plane packet that is sent by the P4Runtime server to the control plane for further inspection. Similarly, a packet-out is defined as a data packet generated by the control plane and injected in the data plane via the P4Runtime server. When inspecting a packet-in, the control plane might need to have access to additional information such as the original data plane port where the packet was received, the timestamp when the packet was received, etc. Similarly, when sending a packet-out, the control plane might need to specify additional information used by the device to process the data packet. Such additional information for packet-in and packet-out can be expressed by means of P4 headers carrying P4 standard annotations @controller_header(”packet_in”) and @controller_header(”packet_out”), respectively, as lines 66-75 show 8.5.

```
66  @controller_header("packet_in")
67  header packet_in_t {
68      bit<24> geoname_id; //geoname_id
69      bit<32> srcAddr;
70  }
71
72  @controller_header("packet_out")
73  header packet_out_t {
74    bit<24> reason_id;
75  }
```

Figure 8.5: controller_header() annotation

Such messages can carry arbitrary metadata specified by means of P4 headers annotated with @controller_header. In our example the packet-in is composed of just two fields:

- geoname_id
- IPV4 source address

These fields are used by the controller to perform the spoofing check, every time this feature is enabled, e.g, after an unusual network behavior experienced.

Packet-out, instead, is composed of just one field:

- reason_id

In our implementation, this field assumes always the value 1, but it will be useful in the future for adding new features. At this point, after having defined the custom headers to be used in the program, they need to be assembled into a single struct (lines 77-84) 8.6.

```
77  struct headers {
78      packet_in_t  packetin;
79      packet_out_t packetout;
80      ethernet_t   ethernet;
81      GSS_t         gss;
82      ipv4_t       ipv4;
83      tcp_t        tcp;
84      }
85
```

Figure 8.6: struct header() definition

”headers” name will be used throughout the program when referring to the headers. User-defined metadata, which are passed from one block to another as the packet propagates through the architecture are not required by this program, so this field is left empty (lines 63-64)8.7

```
63  struct metadata {
64  }
```

Figure 8.7: Empty struct metadata

Parser implementation Figure 8.8

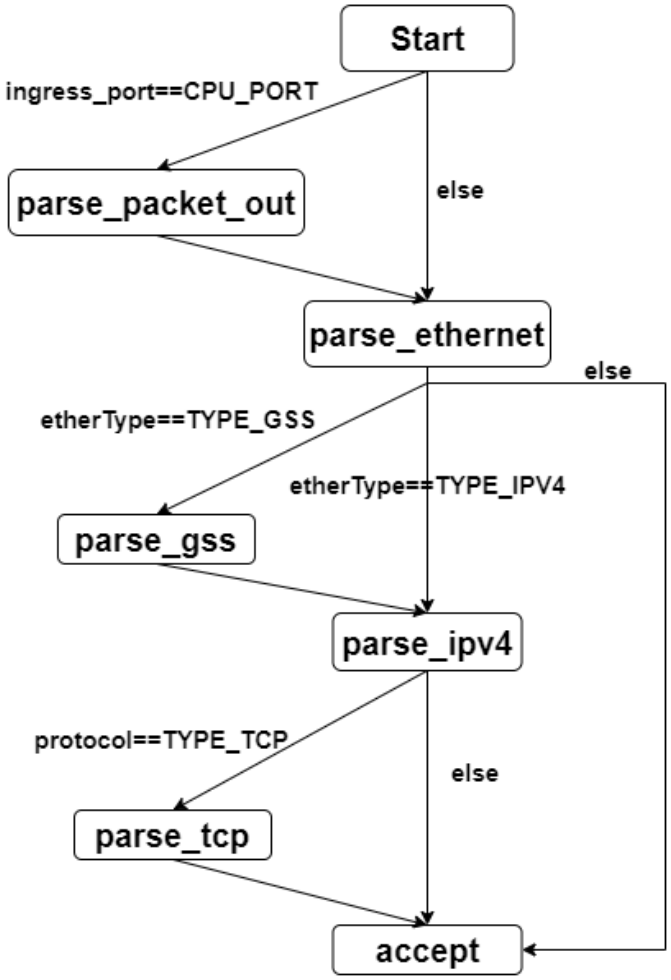


Figure 8.8: Graphical illustration of the parser

depicts the graphical representation of the parser implemented in lines 87-133 8.9.

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
    transition select(standard_metadata.ingress_port){
            CPU_PORT: parse_packet_out;
               default: parse_ethernet;
        }
    }
    state parse_packet_out {
        packet.extract(hdr.packetout);
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            TYPE_GSS: parse_gss;
            default: accept;
        }
    }
    state parse_gss {
        packet.extract(hdr.gss);
        transition parse_ipv4;
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol){
        TYPE_TCP: parse_tcp;
        default: accept;
        }
    }
    state parse_tcp {
        packet.extract(hdr.tcp);
           transition accept;
    }
}
```

Figure 8.9: MyParser() implementation

In the start state, the ingress port (standard_metadata.ingress_port) of packet is examined using the select statement, and the program branches to the state parse_packet_out if it corresponds to CPU_PORT (switch port #255), in which packet-out header (hdr.packetout) is extracted, before moving to parse_ethernet state. Otherwise the state transitions straight to parse_ethernet if the received packet does not come from the CPU_PORT, i.e. it does not come from the controller, so it is not a packet_out. In this state, ethernet header (hdr.ethernet) is extracted, etherType field is inspected using the select statement, and the program branches to

- parse_gss if etherType field corresponds to TYPE_GSS,
- parse_ipv4 if etherType field corresponds to TYPE_IPV4,
- accept in all other cases.

The state transitions straight to accept if the packet contains neither the ipv4 header nor the gss header. As discussed in the 6.7.1.1, this is because explicit transition to reject is not supported on the target we are programming. Dropping the packet that contains neither the ipv4 header nor the gss header will be handled in MyIngress() block and not

in the parser. Once parse_gss state is reached, gss header (hdr.gss) is extracted, before moving to parse_ipv4. In this state, ipv4 header (hdr.ipv4) is extracted, protolcol field (hdr.ipv4.protocol) is inspected using the select statement, and the program branches to

- parse_tcp if protocol field corresponds to TYPE_TCP,
- accept, otherwise.

Finally, in parse_tcp state, the tcp header is extracted, and the program unconditionally transitions to the accept state. It is important to recall that a header also contains a hidden Boolean "validity" field, set to false automatically when the header is created. When the "validity" bit is true we say that the "header is valid" and this bit can be manipulated by using the some methods. It is also assigned the value "true" on a successful extract() method call in the parser. These validity bits will be used in MyIngress(), to process the packet in the right way, according to its composition. Given the choices made in MyParser(), the switch will be able to correctly parse:

- regular IPV4 packets
- packets carrying GSS and IPV4 protocols at the same time

The switch will not be able to correctly parse:

- regular IPV6 packets
- packets carrying GSS protocol without IPV4 protocol

In addition, unlike for TCP protocol, UDP header fields of an IPV4 packet, for example, will not be accessible in subsequent blocks of the program, and the packet will be simply forwarded according to the forwarding rules.

MyVerifyChecksum() As already said in 6.7.1.1, this control block can be used to verifiy the checksum for the packet by applying the verify_checksum() extern. It verifies the checksum of the supplied data and if detects that a checksum of the data is not correct, then the value of the standard_metadata checksum_error field will be equal to 1 when the packet begins ingress processing. In this project, as shown in figure 8.10,

```
135 /*************************************************************************
136 ************   C H E C K S U M    V E R I F I C A T I O N   ************
137 *************************************************************************/
138
139 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
140     apply {  }
141 }
142
143
```

Figure 8.10: MyVerifyChecksum() implementation

it is left empty, since it is not implemented.

MyIngress() control block Scrolling down the script, myIngress() control block is encountered, where packet processing is actually done. In lines 152-165 8.11

```
144 /*************************************************************************
145 *************  I N G R E S S   P R O C E S S I N G   *******************
146 *************************************************************************/
147
148 control MyIngress(inout headers hdr,
149                   inout metadata meta,
150                   inout standard_metadata_t standard_metadata) {
151
152     action drop() {
153         mark_to_drop(standard_metadata);
154     }
155
156     action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
157         standard_metadata.egress_spec = port;
158         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
159         hdr.ethernet.dstAddr = dstAddr;
160         hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
161     }
162     action secure_forward(egressSpec_t port) {
163         standard_metadata.egress_spec = port;
164         hdr.ipv4.ttl = hdr.ipv4.ttl + 1;
165     }
```

Figure 8.11: MyIngress() actions declaration

the actions that will be called to process incoming packets are immediately defined:

- drop(): it causes the packet to be dropped at the end of the ingress processing, by invoking the mark_to_drop primitive.
- ipv4_forward(): it defines the following operations on a packet header:
  1. Updating the egress port so the packet is forwarded to its destination through the correct port
  2. Updating the source MAC address with the packet's previous destination MAC address
  3. Changing the destination MAC address of the packet with the one corresponding to the next hop
  4. Decrementing the time-to-live (TTL) field in the IPv4 header
- secure_forward(): it defines the following operations on a packet header:
  1. Updating the egress port so the packet is forwarded to its destination through the correct port
  2. Incrementing the time-to-live (TTL) field in the IPv4 header

In lines 167-215 8.12

```
167     table ipv4_lpm {
168         key = {
169             hdr.ipv4.dstAddr: lpm;
170         }
171         actions = {
172             ipv4_forward;
173             drop;
174             NoAction;
175         }
176         size = 1024;
177         default_action = drop();
178     }
179     table ipv4_lpm_for_ssh {
180         key = {
181             hdr.ipv4.dstAddr: lpm;
182         }
183         actions = {
184             ipv4_forward;
185             drop;
186             NoAction;
187         }
188         size = 1024;
189         default_action = drop();
190     }
191     table secure_lpm {
192         key = {
193             hdr.ipv4.dstAddr: lpm;
194         }
195         actions = {
196             secure_forward;
197             NoAction;
198         }
199         size = 1024;
200         default_action = NoAction();
201     }
202     table firewall_exact {
203         key = {
204             hdr.gss.geoname_id: exact;
205         }
206         actions = {
207             drop;
208             NoAction;
209         }
210         size = 1024;
211         default_action = NoAction();
212     }
```

Figure 8.12: MyIngress() tables definition

the tables that will be used by the switch are defined:

- ipv4_lpm: The match is against the destination IP address using the longest prefix match (lpm) lookup method. The actions associated with the table are ipv4_forward(), drop() and NoAction(). The default action which is invoked when there is a miss is drop(). The maximum number of entries this table can support is 1024.
- ipv4_lpm_for_ssh: This table is defined in the same way as ipv4_lpm. The reason why two tables with the same structure are created in the same program is that they will be used for different purposes, being populated at runtime by the control plane with different data.
- secure_lpm: The match is against the destination IP address using the longest prefix match (lpm) lookup method. The actions associated with the table are secure_forward() and NoAction(). The default action which is invoked when there is a miss is NoAction(). The maximum number of entries this table can support is 1024.

- firewall_exact: The match is against the geoname_id field using the exact lookup method. The actions associated with the table are drop() and NoAction(). The default action which is invoked when there is a miss is NoAction(). The maximum number of entries this table can support is 1024.

The control block starts executing from the apply statement (lines 218-246) 8.13

```
215     apply {
216         if (hdr.tcp.isValid()){
217             if (hdr.tcp.dstPort != SSH_DEFAULT_PORT){
218                 ipv4_lpm.apply();
219             }
220             else{
221                 ipv4_lpm_for_ssh.apply();
222                 if (standard_metadata.ingress_port == CPU_PORT) { //packet out
223                     hdr.packetout.setInvalid();
224                     secure_lpm.apply();
225                 }
226                 if (standard_metadata.egress_spec == CPU_PORT) { // packet in
227                     hdr.packetin.setValid();
228                     hdr.packetin.geoname_id= hdr.gss.geoname_id; //geoname_id
229                     hdr.packetin.srcAddr=hdr.ipv4.srcAddr;
230                 }
231                 if (hdr.gss.isValid() && standard_metadata.ingress_port != CPU_PORT){
232                     firewall_exact.apply();
233                 }
234                 if (!hdr.gss.isValid()){
235                     drop();
236                 }
237             }
238         }
239         else{
240             ipv4_lpm.apply();
241         }
242     }
243 }
```

Figure 8.13: MyIngress() apply statement

which contains the control logic. In this program, tables are applied as follows:

- ipv4_lpm table is enabled when the incoming packet has a valid TCP header and the TCP port number is different from 22 (SSH_DEFAULT_PORT) or whenever an incoming packet does not have a TCP header (e.g., UDP or ICMP packets).
- ipv4_lpm_for_ssh is enabled when an incoming packet has a valid TCP header and the TCP port number is 22 (SSH_DEFAULT_PORT). In this branch, the program also performs the following operations:
  - It sets invalid the packet-out header and applies the secure_lpm table if the ingress port (standard_metadata.ingress_port) is the CPU_PORT, i.e., it is a packet-out.
  - It sets valid the packet-in header if the ingress port (standard_metadata.egress_port) is the CPU_PORT, i.e., it is a packet-in. Then, geoname_id and srcAddr fields of the packet_in header are filled in properly.
  - It applies the firewall_exact table if the incoming packet has a valid GSS header and the ingress port (standard_metadata.ingress_port) is different from the CPU_PORT, i.e. it is not a packet out.

  - It drops all incoming packets that do not have a valid GSS header.

In the rest of this document we call SSH packets all packets whose destination TCP port is #22. As a result of the decisions taken in the apply block, the switch programmed with this P4 code processes an incoming packet in the following way:

- It simply forwards all non-SSH incoming packets carried over IPV4 according to the content of the ipv4_lpm table.
- It drops all SSH incoming packets that do not have a valid GSS header.
- It drops all SSH incoming packets that have a given geoname_id, inserted at runtime by the network controller in the firewall_exact table.
- It forwards all SSH incoming packets that have a geoname_id, not inserted at runtime by the network controller in the firewall_exact table.
- It forwards all packets coming from the controller, i.e. all the packet-outs according to the content of the secure_lpm table.
- It forwards to the controller all SSH incoming packets that have a valid GSS header and an allowed geoname_id (i.e., all the SSH packets that have passed the firewall rules) whenever the controller has to perform spoof checks.

Notice that packets that do not have TCP port # 22 as their TCP destination port, do not require GSS header to be present between Ethernet and IPV4 headers in order to be forwarded. The reason for this choice is that we want to secure only the service hosted on TCP port #22, service that will be accessible only by hosts that successfully pass firewall rules. Network controller enables spoof checks by setting the port parameter (action_param) to 255 (the CPU_PORT), to be passed as input to the ipv4_forward() action in the ipv4_lpm_for_ssh table. This approach forces the switch to forward the packet to the controller, via packet-in, sending it to the CPU_PORT. In order to disable this feature, the controller restores the content of the table as it was before. This mechanism will be further discussed in the section dedicated to the network controller implementation 8.4.

MyEgress() control block To detect Dos or DDos attacks, switch and controller cooperate:

- The switch updates a counter whenever a packet is sent to the host to be protected by incrementing it by the number of packet bytes.
- The controller reads this counter periodically, and computes the incoming data rate by making the difference between the current value of the counter and the previous value (stored), and dividing this quantity by the number of seconds elapsed between the two readings.

The counter used for this purpose, called c, is defined in MyEgress(), in line 256 8.14.

```
245 /*************************************************************************
246 ***************  E G R E S S   P R O C E S S I N G   ******************
247 *************************************************************************/
248
249 control MyEgress(inout headers hdr,
250                  inout metadata meta,
251                  inout standard_metadata_t standard_metadata) {
252
253     counter(1, CounterType.bytes) c;
254     apply {
255         if (hdr.ipv4.dstAddr== (bit<32>)1601104747 && standard_metadata.egress_spec!=CPU_PORT){
256             c.count(0);
257         }
258     }
259 }
```

Figure 8.14: MyEgress() implementation

It is a byte counter since we want to measure the amount of traffic in terms of bytes, and not in terms of number of packets. It is incremented by an amount equal to the number of bytes contained in the packet that is going to be forwarded by the switch, whenever the IPV4 destination address matches the address of the potential victim of the Dos attack we want to detect. The counter is not incremented when the packet is forwarder to the controller to perform the spoofing check, via the CPU_PORT. The counter was defined and implemented in MyEgress() so that it is incremented only when the packet is forwarded by the switch, after firewall rules and eventual spoofing check have been successfully passed.

MyComputeChecksum() control block Since the program is implementing a routing function, the Time-to-live (TTL) must be decreased. Any change to the header fields will cause the checksum value to change. Therefore, it is necessary to recompute the checksum in the P4 program in case modifications are made to the header fields. MyComputeChecksum() in lines 267-285 8.15

```
/*************************************************************************
*************   C H E C K S U M    C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
     apply {
    update_checksum(
        hdr.ipv4.isValid(),
            { hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}
```

Figure 8.15: MyComputeChecksum() implementation

updates the checksum of the packet by applying the update_checksum() extern.

MyDeparser() control block The deparser, executed after finishing the packet processing by the other control blocks, assembles the packet headers back and serializes them for transmission. Its implementation is shown in figure 8.16

```
/*************************************************************************
**********************  D E P A R S E R  ******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.packetin);
        packet.emit(hdr.ethernet);
        packet.emit(hdr.gss);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

Figure 8.16: MyDeparser() implementation

Notice that the order of emitting packets' headers is important, and the headers are only emitted in case they are valid, followed by the original payload of the packet.

Programming the pipeline sequence Figure 8.17 shows how to write the pipeline sequence in the switch_config.p4 program. In order to define the pipeline sequence according to the V1Model architecture that has been used, it is sufficient to write those lines of code at the end of the P4 file. They specify the parser, the checksum verification block, the ingress and egress blocks, the checksum recomputation block and finally the deparser, previously declared in the P4 program.

```
298  /*************************************************************************
299  ***********************  S W I T C H  ***********************************
300  *************************************************************************/
301
302  V1Switch(
303  MyParser(),
304  MyVerifyChecksum(),
305  MyIngress(),
306  MyEgress(),
307  MyComputeChecksum(),
308  MyDeparser()
309  ) main;
310
```

Figure 8.17: Writing the pipeline sequence

8.3.1 simple_switch_CLI tool

This section introduces a tool (simple_switch_CLI) that is used with the software switch (BMv2) to manage the tables, read the counters, and manage other P4 objects at runtime. The control plane uses the simple_switch_CLI tool to interact with the data plane. The simple_switch_CLI includes a program-independent CLI and a Thrift client which connects to the program-independent control server residing on the BMv2 switch. The working scenario is shown in figure 8.18. To inspect or change the switch con-

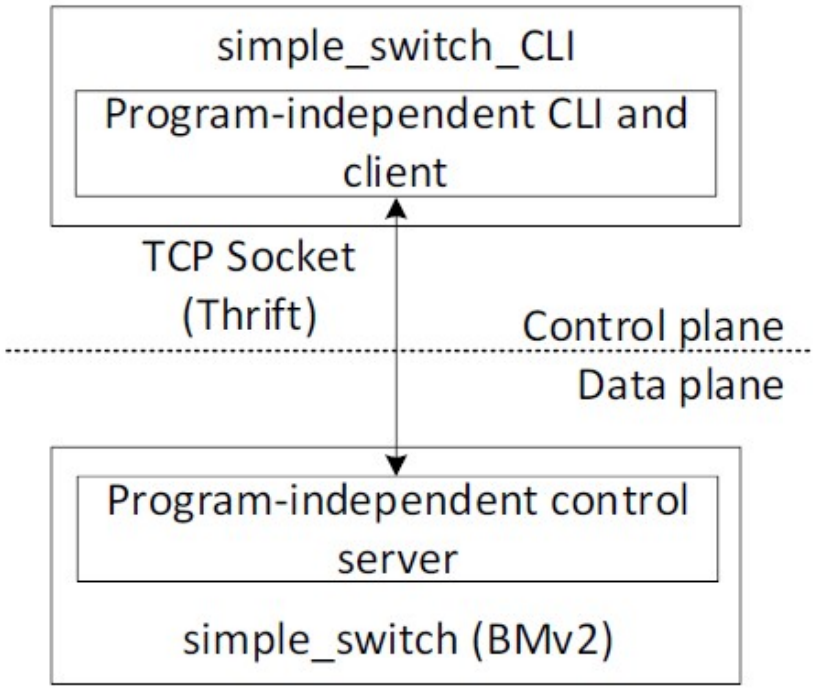


Figure 8.18: Runtime management of a P4 target (BMv2) via simple_switch_CLI

figuration, connect to its CLI from your host operating system using this command:

simple_switch_CLI –thrift-port <switch thrift port>, as shown in figure 8.19. In the same figure all available commands in the tool are displayed. This can be done by simply typing a question mark in the CLI tool. If you need help on a specific command, type help <topic>, where <topic> is the command that you would like to explore. For example, the syntax of table_modify command, is shown in figure 8.20.

```
p4@p4:~/Desktop/Originale/P4_Project$ simple_switch_CLI --thrift-port 9090
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: ?

Documented commands (type help <topic>):
========================================
act_prof_add_member_to_group       serialize_state
act_prof_create_group              set_crc16_parameters
act_prof_create_member             set_crc32_parameters
act_prof_delete_group              set_queue_depth
act_prof_delete_member             set_queue_rate
act_prof_dump                      set_toeplitz_hash_key
act_prof_dump_group                shell
act_prof_dump_member               show_actions
act_prof_modify_member             show_ports
act_prof_remove_member_from_group  show_pvs
counter_read                       show_tables
counter_reset                      swap_configs
counter_write                      switch_info
get_time_elapsed                   table_add
get_time_since_epoch               table_clear
help                               table_delete
load_new_config_file               table_dump
mc_dump                            table_dump_entry
mc_mgrp_create                     table_dump_entry_from_key
mc_mgrp_destroy                    table_dump_group
mc_node_associate                  table_dump_member
mc_node_create                     table_indirect_add
mc_node_destroy                    table_indirect_add_member_to_group
mc_node_dissociate                 table_indirect_add_with_group
mc_node_update                     table_indirect_create_group
mc_set_lag_membership              table_indirect_create_member
meter_array_set_rates              table_indirect_delete
meter_get_rates                    table_indirect_delete_group
meter_set_rates                    table_indirect_delete_member
mirroring_add                      table_indirect_modify_member
mirroring_add_mc                   table_indirect_remove_member_from_group
mirroring_delete                   table_indirect_reset_default
mirroring_get                      table_indirect_set_default
port_add                           table_indirect_set_default_with_group
port_remove                        table_info
pvs_add                            table_modify
pvs_clear                          table_num_entries
pvs_get                            table_reset_default
pvs_remove                         table_set_default
register_read                      table_set_timeout
register_reset                     table_show_actions
register_write                     write_config_to_file
reset_state

Undocumented commands:
======================
EOF  greet

RuntimeCmd:
```

Figure 8.19: Starting and displaying the available commands in the simple_switch_CLI

```
p4@p4:~/Desktop/Originale/P4_Project$ simple_switch_CLI --thrift-port 9090
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: table_info MyIngress.ipv4_lpm
MyIngress.ipv4_lpm            [implementation=None, mk=ipv4.dstAddr(lpm, 32)]
**********************************************************************
MyIngress.drop                []
MyIngress.ipv4_forward        [dstAddr(48),    port(9)]
NoAction                      []
RuntimeCmd: help table_modify
Add entry to a match table: table_modify <table name> <action name> <entry handle> [action parameters]
RuntimeCmd: table_info MyIngress.ipv4_lpm
MyIngress.ipv4_lpm            [implementation=None, mk=ipv4.dstAddr(lpm, 32)]
**********************************************************************
MyIngress.drop                []
MyIngress.ipv4_forward        [dstAddr(48),    port(9)]
NoAction                      []
RuntimeCmd: table_num_entries MyIngress.ipv4_lpm
4
RuntimeCmd: table_modify MyIngress.ipv4_lpm MyIngress.ipv4_forward 0 ff:ff:ff:ff:ff:ff 1
Modifying entry 0 for lpm match table MyIngress.ipv4_lpm
RuntimeCmd:
```

Figure 8.20: Example of simple_switch_CLI usage

In the same figure also the commands to display table information, to retrieve the number of entries of a table and to modify a table entry of a table are shown.

8.4 Network controller

In our project, we set a controller as the entity that gives instructions to switches and perform operations like Dos and DDos detecting and also spoofing checking. In particular, we implemented a python script that can be run in the terminal with the command:

```
$sudo python3 controller.py
```

This controller will run in background while the network started by Mininet is already instantiated. In fact if the network is not already started, the controller will not run, giving a gRPC error; this is because by utililizing the runtime method MasterArbitrationUpdate() the controller will fail to connect to switches addresses that have not already been istantiated. In Figure 8.21 it is possible to see the controller script actually running and controlling the network:

```
The spoofing check is currently disabled!

0
s4 c 0: 0 packets (0 bytes)
Normal traffic
05:14:52_PM bit_rate: 0.0 KBps
Press 1 if you want to enable the spoofing check, 0 if you want to disable it, or nothing to continue in this way:
s4 c 0: 0 packets (0 bytes)
Normal traffic
05:14:57_PM bit_rate: 0.0 KBps
Press 1 if you want to enable the spoofing check, 0 if you want to disable it, or nothing to continue in this way:
If you want to enable spoofing check answer 1:
The spoofing check is currently disabled!
```

Figure 8.21: Background running controller

8.4.1 Main functions of the controller

The implemented controller is involved in three main tasks :

- Instructing the switches network by giving basic rules for forwarding
- Inserting a firewall protection to the packets that are passed through the hosts
- check of possible spoofing from hosts that send packets through the network
- detection of Dos attacks against a specific host

Because of the controller needs to be able to repeat and also to do simultaneously these actions, a multi thread execution of methods was used with three different threads.

8.4.2 Switch instructions from controller

One of the controller roles is to give to switches the forwarding instructions to direct packets passing through the network. First of all the controller creates an object of SwitchConnection, then the instruction flows between the controller and the switch is created with the methods MasterArbutrationUpdate() and SetforwardingPipeline(), imported from P4Runtime library in python. In particular, the first method is used to establish the controller as master of the switch (this operation is required by p4 runtime before doing every write operation); the second method is used to actually install the p4 program on the switch (rapresented by a switchconnection object). After the creation of the switch object with a P4 program installed, the controller can write new forwarding rules on the switch, by using these methods implemented in our project as reported in the code.

```
1     s1 = bmv2.Bmv2SwitchConnection(
2             name="s1",
3             address="0.0.0.0:50051",
4             device_id=1,
5             proto_dump_file="p4runtime1.log",
6     if s1.MasterArbitrationUpdate() == None:
7             print("Failed to establish the connection")
8     # Install the P4 program on the switches
9     s1.SetForwardingPipelineConfig(
10            p4info=p4info_helper.p4info, bmv2_json_file_path=
                  bmv2_file_path
11        )
12
13
14    #S1
15    writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.
          ipv4_lpm_for_ssh", dst_ip_addr="8.27.67.188", dst_eth_addr= "
          08:00:00:00:01:11",port=1)
16    writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.
          ipv4_lpm_for_ssh", dst_ip_addr="31.28.27.50", dst_eth_addr= "
          08:00:00:00:00:02",port=10)
```

```
    writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.
        ipv4_lpm_for_ssh", dst_ip_addr="89.46.106.33", dst_eth_addr= "
        08:00:00:00:00:02",port=10)
    writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.
        ipv4_lpm_for_ssh", dst_ip_addr="95.110.235.107", dst_eth_addr= "
        08:00:00:00:00:04",port=11)
    writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.ipv4_lpm",
        dst_ip_addr="8.27.67.188", dst_eth_addr= "08:00:00:00:01:11",port
        =1)
    writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm",
        dst_ip_addr="31.28.27.50", dst_eth_addr= "08:00:00:00:00:02",port
        =10)
    writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm",
        dst_ip_addr="89.46.106.33", dst_eth_addr= "08:00:00:00:00:02",port
        =10)
    writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm",
        dst_ip_addr="95.110.235.107", dst_eth_addr= "08:00:00:00:00:04",
        port=11)
    writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
        8.27.67.188", port=1)
    writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
        31.28.27.50", port=10)
    writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
        89.46.106.33", port=10)
    writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="
        95.110.235.107", port=11)
```

Here are reported implemented methods defined on the controller to manage and modify the forwarding tables on switches:

```
def writeIpv4Rules(p4info_helper,table, sw_id, dst_ip_addr, dst_eth_addr,
    port):
    table_entry = p4info_helper.buildTableEntry(
        table_name=table,
        match_fields={"hdr.ipv4.dstAddr": (dst_ip_addr, 32)},
        action_name="MyIngress.ipv4_forward",
        action_params={
            "dstAddr": dst_eth_addr,
            "port": port},
    )
    sw_id.WriteTableEntry(table_entry)
    print("Installed ipv4 rule on %s" % sw_id.name)

def writeIpv4SecureRules(p4info_helper, sw_id, dst_ip_addr, port):
    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.secure_lpm",
        match_fields={"hdr.ipv4.dstAddr": (dst_ip_addr, 32)},
        action_name="MyIngress.secure_forward",
        action_params={"port": port},
    )
    sw_id.WriteTableEntry(table_entry)
```

```
    print("Installed ipv4 secure rule on %s" % sw_id.name)


def writeFirewallRules(p4info_helper, ingress_sw, geoname_id):
        table_entry = p4info_helper.buildTableEntry(
            table_name="MyIngress.firewall_exact",
            match_fields={
                "hdr.gss.geoname_id": geoname_id
            },
            action_name="MyIngress.drop",
            action_params={
            })
        ingress_sw.WriteTableEntry(table_entry)
        print("Installed ifirewall rule on %s" % ingress_sw.name)
```

These methods build tables for the forwarding rules of the switch with parameters that are passed as inputs:

- writeIpv4Rules() method builds a table_entry object using the table name (given as input), by setting MyIngress.ipv4_forward action as action to be performed when there is a match against the input dst_ip_addr and by setting dst_eth_addr and "port" input parameters as action parameters. Then, this table_entry is written on the switch given as input and a message of successful table_entry installation is printed by the controller, beside the name of the switch.

- writeIpv4SecureRules() method builds a table_entry object using the table name MyIngress.secure_lpm, by setting MyIngress.secure_forward action as action to be performed when there is a match against the input dst_ip_addr and by setting "port" input parameter as action parameter. Then, this table_entry is written on the switch given as input and a message of successful table_entry installation is printed by the controller, beside the name of the switch. This method will be applied only after the packet is sent back to the switch from the controller with a port value that is the 255 (CPU PORT); the controller performed a spoofing check on the packet before sending it to the original switch.

- writeFirewallRules() method builds a table_entry object using the table name MyIngress.firewall_exact by setting MyIngress.drop action as action to be performed when there is a match against the input geoname_id . Then, this table_entry is written on the switch given as input and a message of successful table_entry installation is printed by the controller, beside the name of the switch given as input. The switch will drop the packet if the geoname_id of the packet matches the input geoname_id of this function. This fucntion will apply only if the destination port is the #22 TCP port and the packet also contains a valid GSS header.

This procedure is repeated for every switch in the topology, the only difference is the parameters to be passed that are chosen according to the switch that will be instructed.

Due the effect of instructions given by controller, it is possible to run the command pingall in the Mininet network. As it is possible to see from figure 8.22, trying to pingall before launching the controller will give a 100% packets dropping.

```
Starting mininet!

*********************************************************************
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>

*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
mininet>
```

Figure 8.22: Pingall command before running controller

Then we run the controller script, that installs all rules on the switches in the topology, as illustrated in figure 8.23.

```
p4@p4:~/Desktop/Originale/P4_Project$ sudo python3 controller.py
Installed P4 Program using SetForwardingPipelineConfig on s1
Installed P4 Program using SetForwardingPipelineConfig on s2
Installed P4 Program using SetForwardingPipelineConfig on s3
Installed P4 Program using SetForwardingPipelineConfig on s4
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 rule on s1
Installed ipv4 secure rule on s1
Installed ipv4 secure rule on s1
Installed ipv4 secure rule on s1
Installed ipv4 secure rule on s1
```

Figure 8.23: Running controller

Thanks to the forwarding rules installed by the controller on the switches, the pingall command (also generic send and receive actions for packets in the network) will result in a 0% dropped packets (figure 8.24).

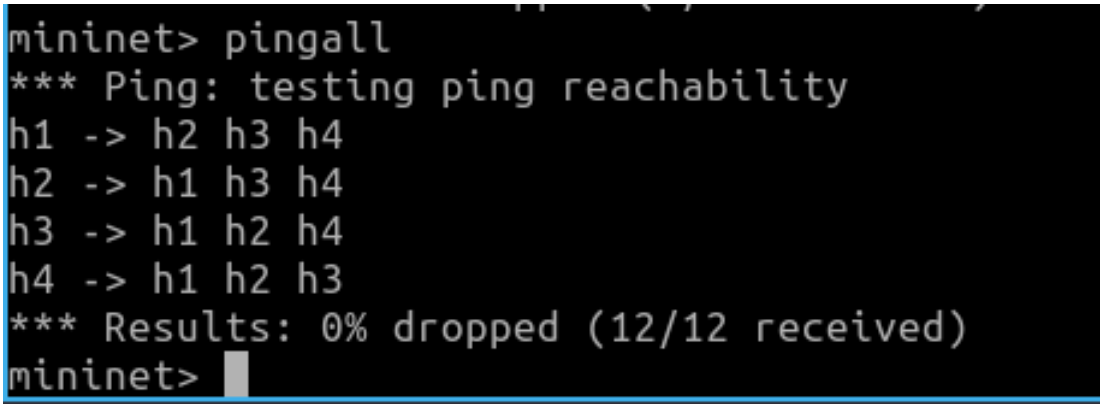


Figure 8.24: Pingall command after running controller

8.4.3 Firewall Protection

Firewall Protection on the network is enabled when a packet has the TCP port #22 as destination port. To set the port number in the packet to be sent, the script send.py must be modified with the desired port number. In our project we set the sending and receiving operations using the scapy library from python. Here are reported codes of send.py, receive.py and gss_header.py, respectively.In particular:

- in send.py script we construct the packet structure with headers of Ethernet, gss (if present), IP and TCP. It is possible to build packets with the GSS header or not (if GSS is not present, IPV4 will be the default layer after Ethernet), also it is possible to specify the port where the packet is sent(ssh is the protected one).
- in receive.py we take the value of headers from the received packet and also print them. Moreover the payload of the packet is taken.
- in gss_header.py we define a new header that is 24 bits long with a field geoname_id. Also the TYPE_GSS value is defined to bind layers in the last lines of code. In fact, the parser will understand if GSS header or IP header (GSS header not present) is positioned after the Ethernet header by looking at the value of etherType field in the Ethernet header.

```
1 #!/usr/bin/env python3
2 import argparse
3 import random
4 import socket
5 from scapy.all import *
6
7 from gss_header import gss
8 from scapy.all import IP, TCP, Ether, get_if_hwaddr, get_if_list, sendp
```

```
9
10
11 def get_if():
12     ifs=get_if_list()
13     iface=None # "h1-eth0"
14     for i in get_if_list():
15         if "eth0" in i:
16             iface=i
17             break;
18     if not iface:
19         print("Cannot find eth0 interface")
20         exit(1)
21     return iface
22
23 def main():
24     parser = argparse.ArgumentParser()
25     parser.add_argument('ip_addr', type=str, help="The destination IP address to use")
26     parser.add_argument('message', type=str, help="The message to include in packet")
27     parser.add_argument('--geoname_id', type=int, default=None, help='The geoname_id the
           packet comes fro, if unspecified then gss header will not be included in packet')
28     args = parser.parse_args()
29
30     addr = socket.gethostbyname(args.ip_addr)
31     geoname_id = args.geoname_id
32     iface = get_if()
33     if (geoname_id is not None):
34     #if (geoname_id is not None):
35         #print("sending on interface {} to dst_id {}".format(iface, str(dst_id)))
36         pkt =  Ether(src=get_if_hwaddr(iface), dst='ff:ff:ff:ff:ff:ff')
37         pkt = pkt / gss(geoname_id=geoname_id)/ IP(dst=addr) /TCP(dport=22, sport=random.
               randint(49151,49153))/ args.message
38         #pkt = pkt / geolocalizzazione(CAP=CAP) / IP(dst=addr) / args.message
39     else:
40         print("sending on interface {} to IP addr {}".format(iface, str(addr)))
41         pkt =  Ether(src=get_if_hwaddr(iface), dst='ff:ff:ff:ff:ff:ff')
42         pkt = pkt / IP(dst=addr) / TCP(dport=22,sport=random.randint(49157,49159))/args.
               message
43     pkt.show2()
44 #    hexdump(pkt)
45 #    print "len(pkt) = ", len(pkt)
46     sendp(pkt, iface=iface, verbose=False)
47     print(pkt.summary())
48     print(geoname_id)
49
50 if __name__ == '__main__':
51     main()
```

Listing 8.3: send.py

```
1  #!/usr/bin/env python3
2  import os
3  import sys
4
5  from gss_header import gss
6  from scapy.all import IP, TCP, get_if_list, sniff
7
8
9
10 def get_if():
11     ifs=get_if_list()
12     iface=None
13     for i in get_if_list():
14         if "eth0" in i:
15             iface=i
16             break;
17     if not iface:
```

```
18          print("Cannot find eth0 interface")
19          exit(1)
20      return iface
21
22  def handle_pkt(pkt):
23          if IP in pkt:
24              if pkt.getlayer(IP).ttl < 64: #to print only the received packet
25                  if TCP in pkt:
26                      print("got a TCP packet")
27                  if gss in pkt:
28                      print("got a gss packet")
29                  pkt.show2()
30                  sys.stdout.flush()
31  def main():
32      ifaces = [i for i in os.listdir('/sys/class/net/') if 'eth' in i]
33      iface = ifaces[0]
34      print("sniffing on %s" % iface)
35      sys.stdout.flush()
36      sniff(iface = iface,
37            prn = lambda x: handle_pkt(x))
38
39  if __name__ == '__main__':
40      main()
```

Listing 8.4: receive.py

```
1  from scapy.all import *
2
3  TYPE_GSS = 0x1212
4
5  class gss(Packet):
6      name = "gss"
7      fields_desc = [
8          ThreeBytesField("geoname_id",0)
9      ]
10     def mysummary(self):
11         return self.sprintf("geoname_id=%geoname_id%")
12
13
14 bind_layers(Ether, gss, type=TYPE_GSS)
15 bind_layers(gss, IP)
```

Listing 8.5: gss_header.py

In order to show results a communication between h1 and h4 was simulated. As shown in figures 8.25 and 8.26, according also to the p4 action implementation on the switches, if we transmit a packet on the TCP destination port 22 with a not specified geonameId value or with a value that comes from a black list (in our case 1796236 and 498817 for Shangai and St. Petersburg) this will be discarded (figures 8.25 and 8.26) by the switch without arriving to controller.

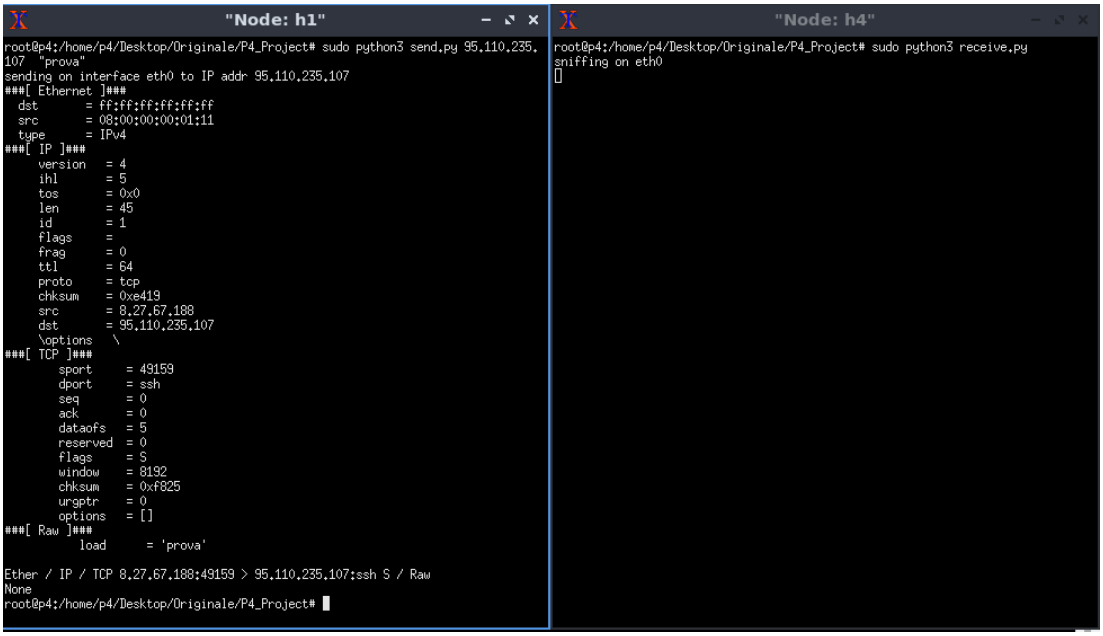


Figure 8.25: h1 sending packets to h4 with no value for geonameId

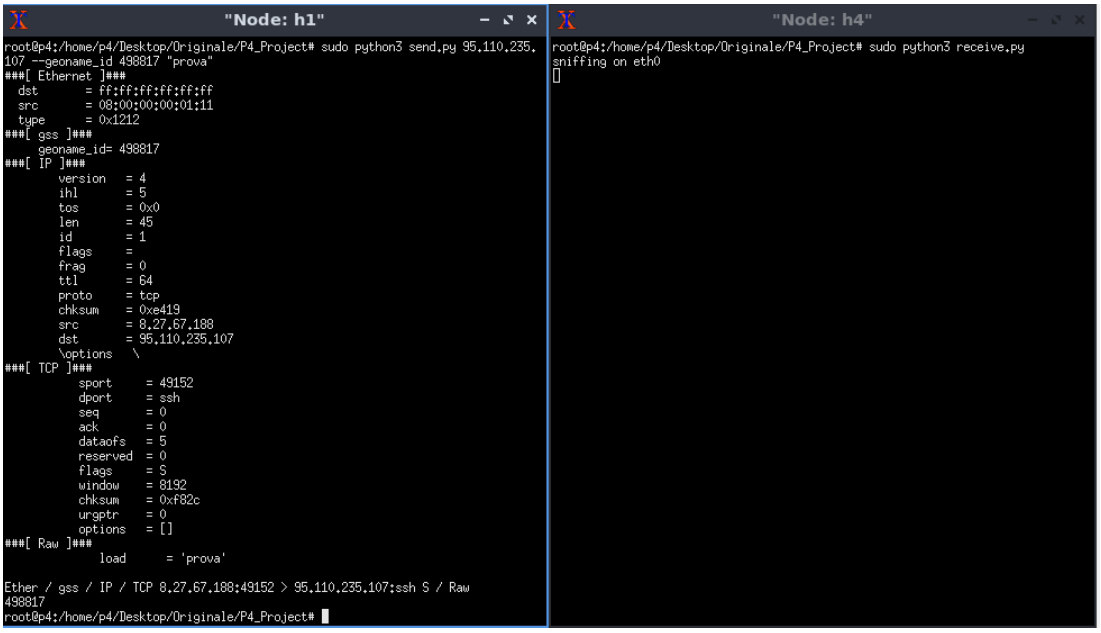


Figure 8.26: h1 sending packets to h4 with real value for geonameId

8.4.4 Dos and DDos detection

In the third thread of the controller script, a detection of Dos and DDos on the switch 4 was implemented. The thread is defined in this way:

```
    def thread3(p4info_helper, switch_connection_list):
    s4=switch_connection_list[3]
    old_counter=0
    new_counter=printCounter(p4info_helper, s4, "c", 0)
    data_rate=(new_counter-old_counter)/5000
    localtime=time.localtime()
    result=time.strftime("%I:%M:%S_%p", localtime)
    data_rate="bit_rate: %s KBps" % (data_rate)
    print(result, data_rate)
    sleep(5)
    while True:
        old_counter=new_counter
        trashold=1000 #1 MBps
        new_counter=printCounter(p4info_helper, s4,  "c", 0)
        data_rate=(new_counter-old_counter)/5000
        if data_rate > trashold:
            print("DDos detected")
        else:
            print("Normal traffic")
        localtime=time.localtime()
        result=time.strftime("%I:%M:%S_%p", localtime)
        data_rate="bit_rate: %s KBps" % (data_rate)

        print(result, data_rate)
        print("Press 1 if you want to enable the spoofing check, 0 if you
            want to disable it, or nothing to continue in this way: ")
        sleep(5)
```

From the code it is shown how the Dos and DDos are detected. In particular the first lines are used to read the counter (of bytes) passed through the switch 4; this is obtained by the function printCounter() that accesses to the switch and reads the counter. Then a 5 second sleep of the thread let us read the updated value of the counter, after a substracion between the new and old counter is performed and the result is diveded by 5000. If the data rate calculated as $(new\_counter - old\_counter)/5000$ (because treshold is set to 1 KBps) is greater than the treshold Dos or DDos is detected and the controller will print a detection message. This detection by reading the counter is done every 5 second continuosly by a 'while True' loop in python. The packets traffic simulation was done by iperf3 tool previously described. An example of detection is shown in following figures 8.27 and 8.28. In this example we simulate traffic from h1 to h4. In this case, h4 is set as server listening on port 5201, while h1 is set as client generating 1 Gbps of traffic directed on h4 on port 5201 (figure 8.27). In this case the generated traffic of 1 Gbps is not enough to trigger the DDos detector on the controller. The controller will show us a message of 'Normal traffic'(figure 8.28).

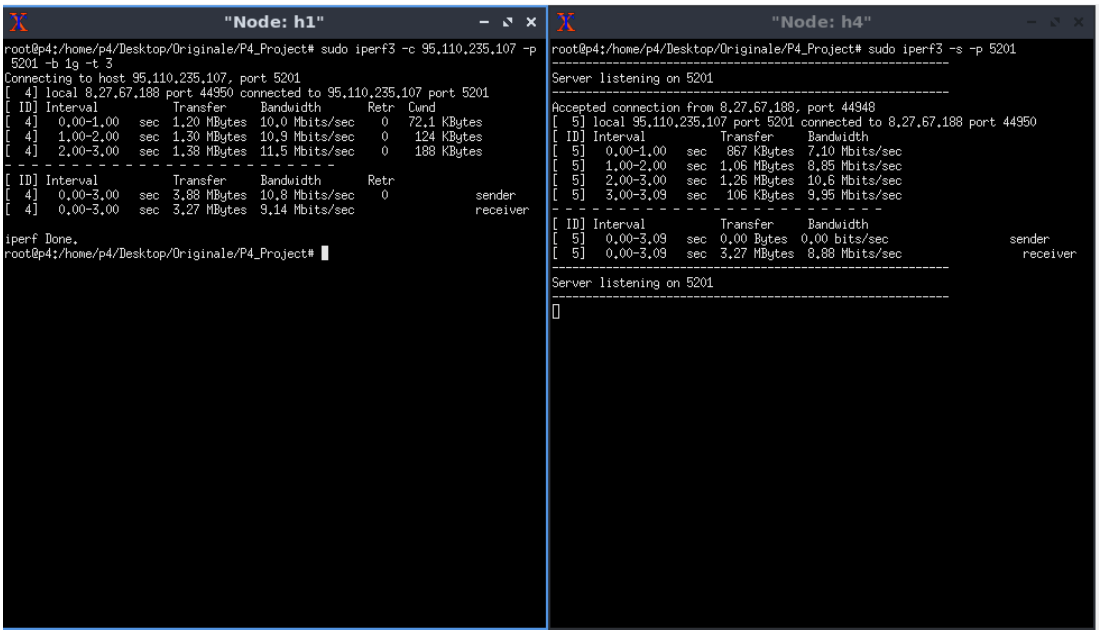


Figure 8.27: h1 genereting 1Gbps traffic for h4

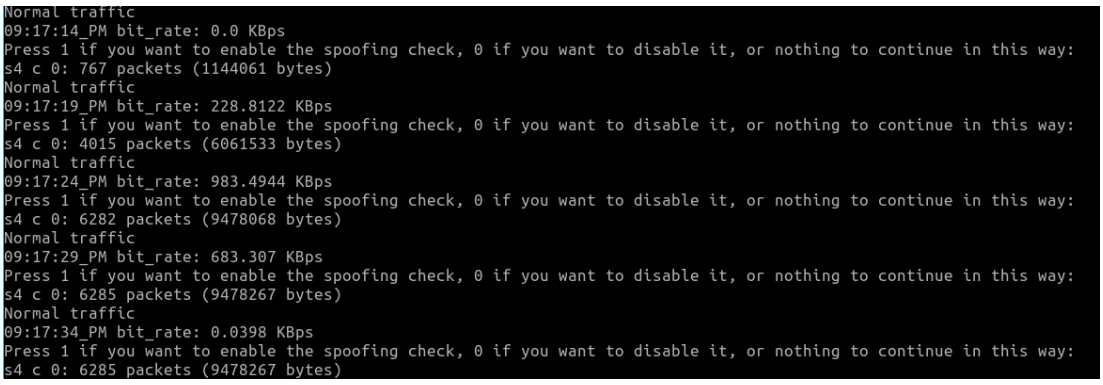


Figure 8.28: Normal traffic detected

In this other case, we generate more traffic (10 GBps) from h1 to h4 (figure 8.29) and the DDos is detected by the controller with a message(figure 8.30). In particular these data rates are not reachable on Mininet, because resources are partitioned among hosts.

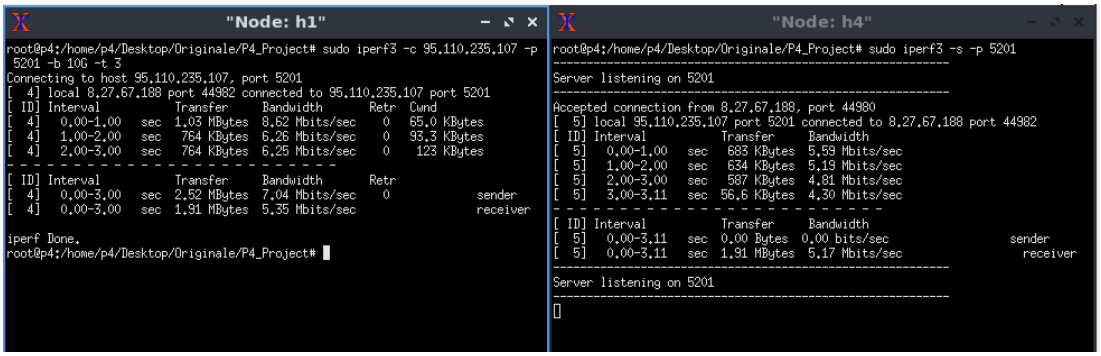


Figure 8.29: h1 genereting 10 GBps for h4

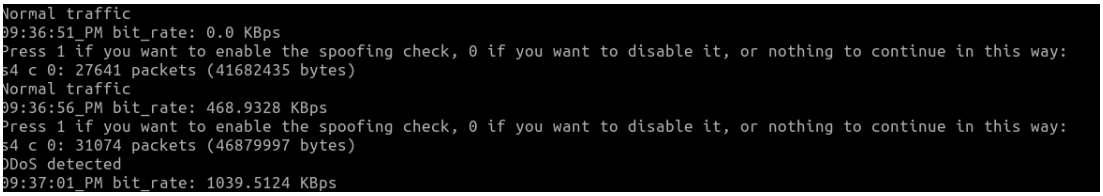


Figure 8.30: DDos detected

Moreover, the DDos can be tested and detected by the controller in the same way described before. In Figure 8.31, we simulated 10 GBps of traffic from h1 to h4 and also from h3 to h4 on port 5201 and 5202, respectively. The running controller detects the amount of traffic and advertises the user with a DDos detection message (figure 8.32).

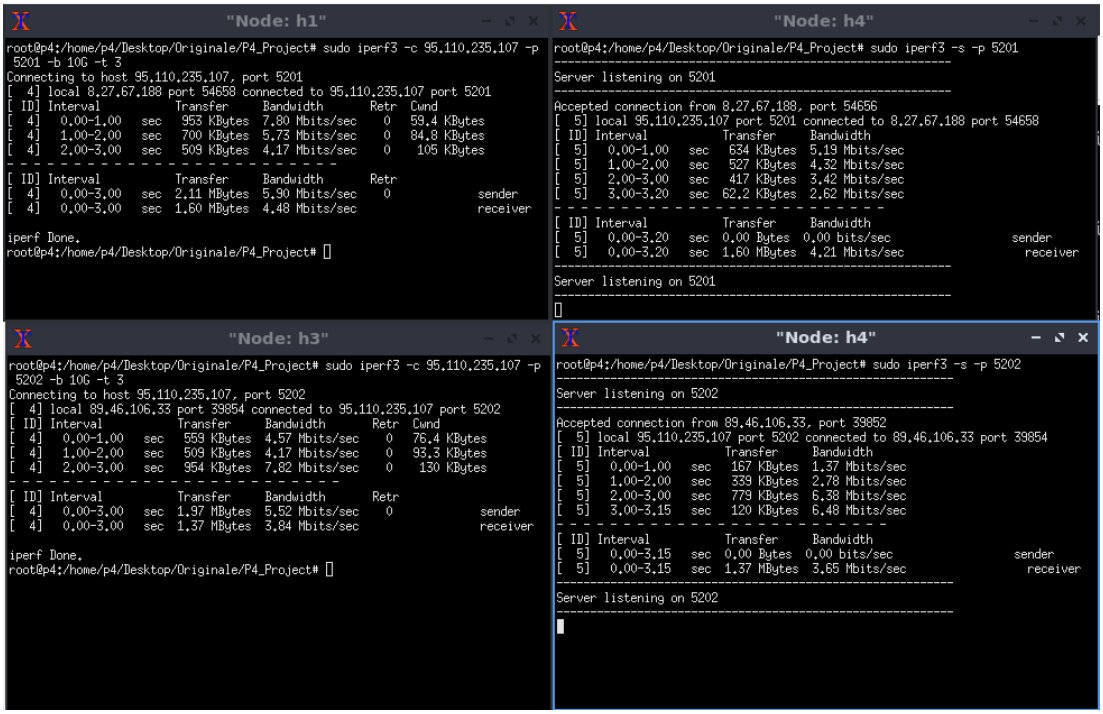


Figure 8.31: h1 and h3 generating 10 GBps of traffic to h4

```
Normal traffic
10:11:56_AM bit_rate: 0.0 KBps
Press 1 if you want to enable the spoofing check, 0 if you want to disable it, or nothing to continue in this way:
s4 c 0: 5518 packets (8241068 bytes)
DDoS detected
10:12:01_AM bit_rate: 168.8578 KBps
Press 1 if you want to enable the spoofing check, 0 if you want to disable it, or nothing to continue in this way:
s4 c 0: 7323 packets (10925228 bytes)
DDoS detected
10:12:06_AM bit_rate: 536.832 KBps
Press 1 if you want to enable the spoofing check, 0 if you want to disable it, or nothing to continue in this way:
s4 c 0: 7323 packets (10925228 bytes)
```

Figure 8.32: DDos detected

8.4.5 Spoofing check implementation

Spoofing check is a very important role implemented in this controller; the check is done on the geonameId retrieved from the ip address of the packet and compared with the one coming from the GeoIp2 databases(described in chapter 4). In order to implement these functions the following code was used:

```
def thread1(p4info_helper, switch_connection):
    while True:
        packetin = switch_connection.PacketIn()# Packet in!
        if packetin is not None:
            print("PACKET IN received")
            #print(packetin)
            packet = packetin.packet.payload
            d = MessageToDict(packetin.packet)
            geoname_id=from_base64_to_decimal(d['metadata'][0]['value'])
            srcAddr=from_base64_to_ipv4(d['metadata'][1]['value'])
            print("Geoname_ID: %d\nsrcAddr: %s" % (geoname_id,srcAddr))
            print("Spoofing check result: %s" %(check_spoofing(srcAddr,
                geoname_id)))
            print(switch_connection.name)
            if check_spoofing(srcAddr, geoname_id):
                packetout = p4info_helper.buildPacketOut(
                    payload=packet,
                    metadata={
                        1: encodeNum(1, 16) #the reason_id assumes always
                            the value 1, useful in future for new features
                    },
                )
                print("send PACKET OUT")
                switch_connection.PacketOut(packetout)

def thread2(p4info_helper, switch_connection_list):
    check_spoofing=0
    print("The spoofing check is currently disabled!\n")
    num=int(input("If you want to enable spoofing check, please answer 1:
        \n"))
    s1=switch_connection_list[0]
    s2=switch_connection_list[1]
    s3=switch_connection_list[2]
    s4=switch_connection_list[3]
    while True:
```

```
33          if num == 1:
34              table_update_spoofing_on(p4info_helper,
                    switch_connection_list)
35              print("The spoofing check is currently enabled!\n")
36              sleep(10)
37              num=int(input("If you want to disable spoofing check, please
                    answer 0: \n"))
38
39          elif num == 0:
40              table_update_spoofing_off(p4info_helper,
                    switch_connection_list)
41              print("The spoofing check is currently disabled!\n")
42              sleep(10)
43              num=int(input("If you want to enable spoofing check answer 1:
                     \n"))
44          check_spoofing=num
```

Thread 1 is the function that checks a possible spoofing on the geonameId with the function check_spoofing. If the spoofing check is enabled by the user, the controller modifies at runtime the tables entries by modifying the exit port and setting the 255 value for that port; this is done in order to perform the packetIn operation. When spoofing check is disabled the controller restores the contents of the tables and the forwarding without spoofing operation is restarted. Thread 2 is the function that takes input from user to enable or disable the spoofing check, also updating tables on switches. Thread 1 uses check_spoofing function that matches the geonameId of the packetIn with the real one coming from the GeoIp2 database, returning a boolean value.

```
1  def check_spoofing(ipv4_address, geoname_id):
2      result=False
3      try:
4          reader=geoip2.database.Reader("/home/p4/Desktop/Originale/GeoLite2
               -City_20221004/GeoLite2-City.mmdb")
5          response=reader.city(ipv4_address)
6          real_geoname_id=response.city.geoname_id
7          #print(response.city.name)
8          #print(response.city.geoname_id)
9          if geoname_id == real_geoname_id:
10             result=True
11     except geoip2.errors.AddressNotFoundError:
12         print("Address not present in the database")
13     reader.close()
14     return result
```

The spoofing check is done only if the TCP destination port is 22, otherwise the controller cannot access the packet because no packetIn operation is performed. In particular, the controller can be in a detection of spoofing state or not, this is indicated by the user that can decide if spoofing check should be enabled or not at runtime on controller by just pressing 1 or 0 . As shown in figure 8.33 the current state of the controller is spoofing

check disabled. So a spoofing forwarding will not be detected in the network and the packet will be regularly forwarded.

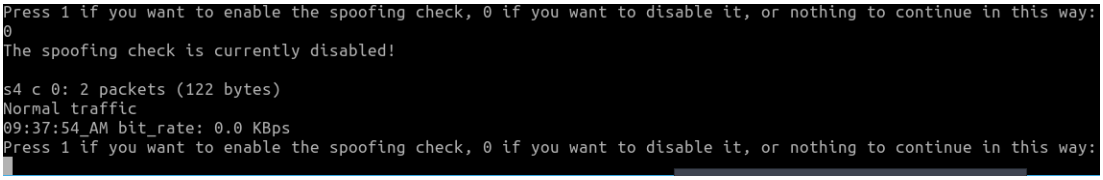


Figure 8.33: Spoofing disabled

So starting from a spoofing check disabled state, trying to send a packet from h1 to h4 will be normally executed and packets will arrive to h4. First we set h4 as receiver, then we send packets from h1 to h4, with two different geonameId 498817 and 498815, respectively (figures 8.34 and 8.35). In particular, the true geonameid (the one taken from the GeoIp database) of h1 is 498817; if the spoofing check is disabled the first packet will be discarded because of firewall rules installed on the switch (figure 8.34) while the spoofed one will regularly arrive to h4 (figure 8.35).

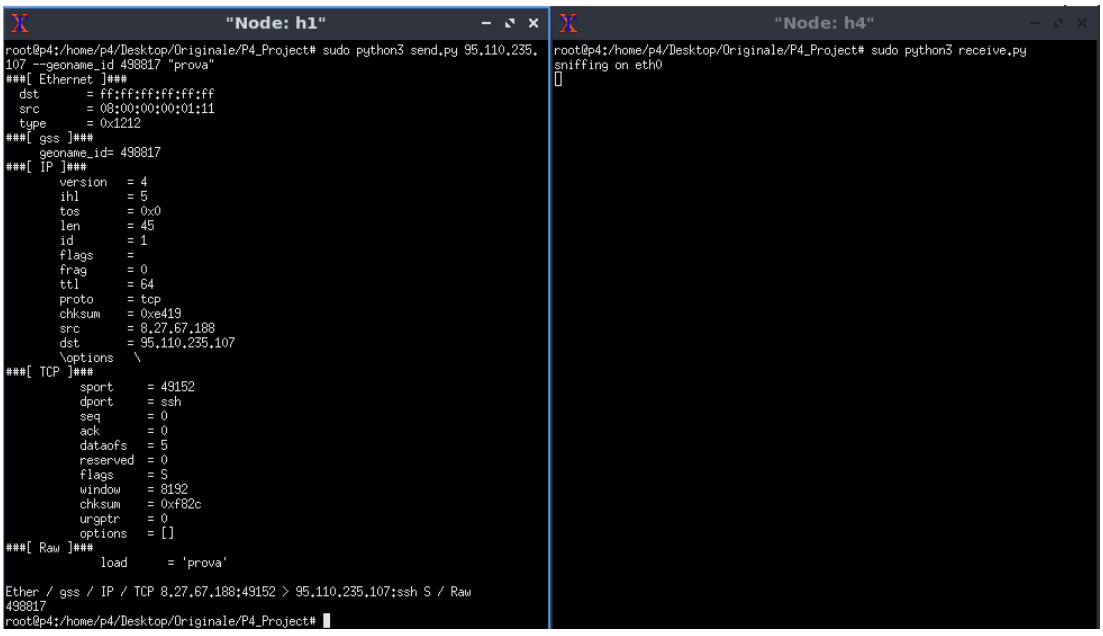


Figure 8.34: h1 sending to h4 with the real value of geonameId

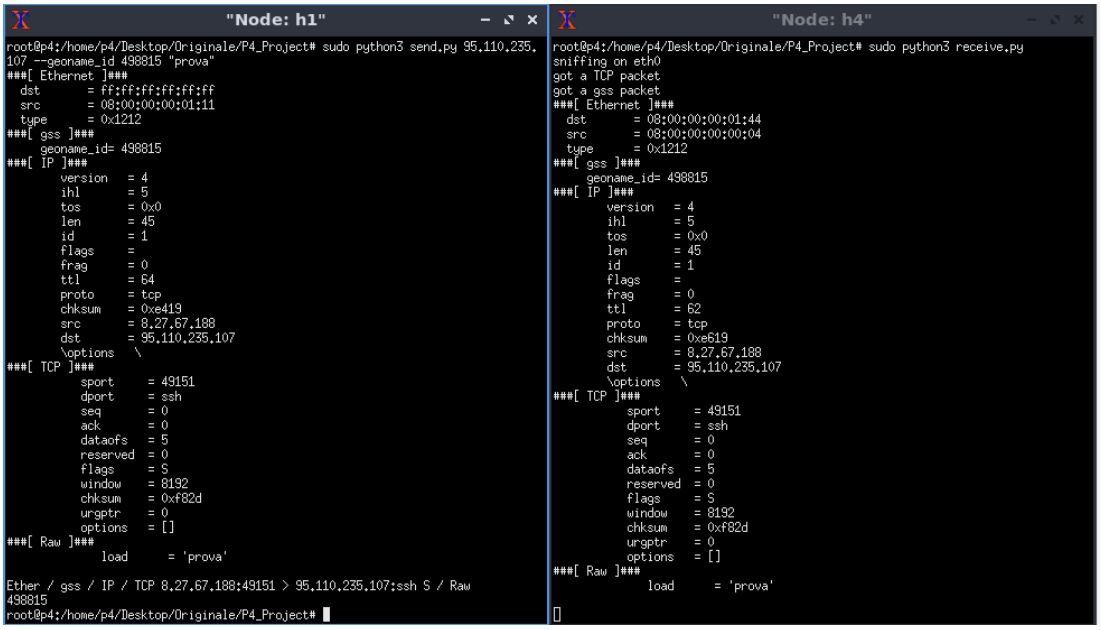


Figure 8.35: h1 sending to h4 with the spoofed value of geonameId

Suppose now spoofing check is enabled by the user by pressing 1 on the input interface of the controller, as shown in figure 8.36 .

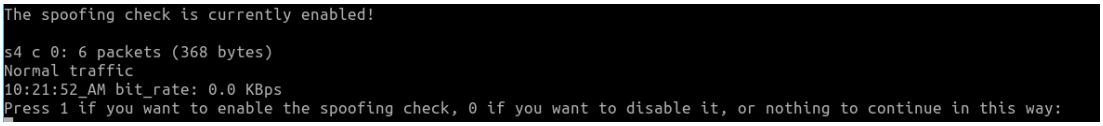


Figure 8.36: Spoofing check enabled

If we send a packet from h1 to h4 with the same send and receive commands shown before, the packet with the correct geonameId will be discarded from the switch because of the firewall rules(figures 8.37)

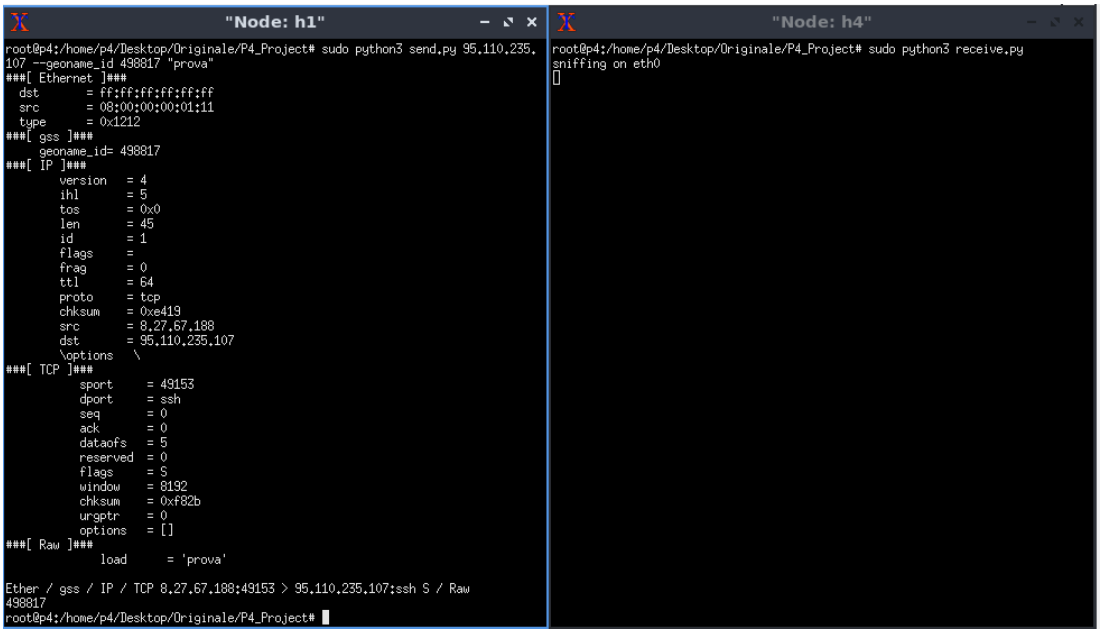


Figure 8.37: h1 sending to h4 with real value of geonameId and spoofing check enabled on controller

Instead the packet with the spoofed geonameId will be discarded from the controller because of the spoofing detection on the geonameId header of the packet (figures 8.38 and 8.39).

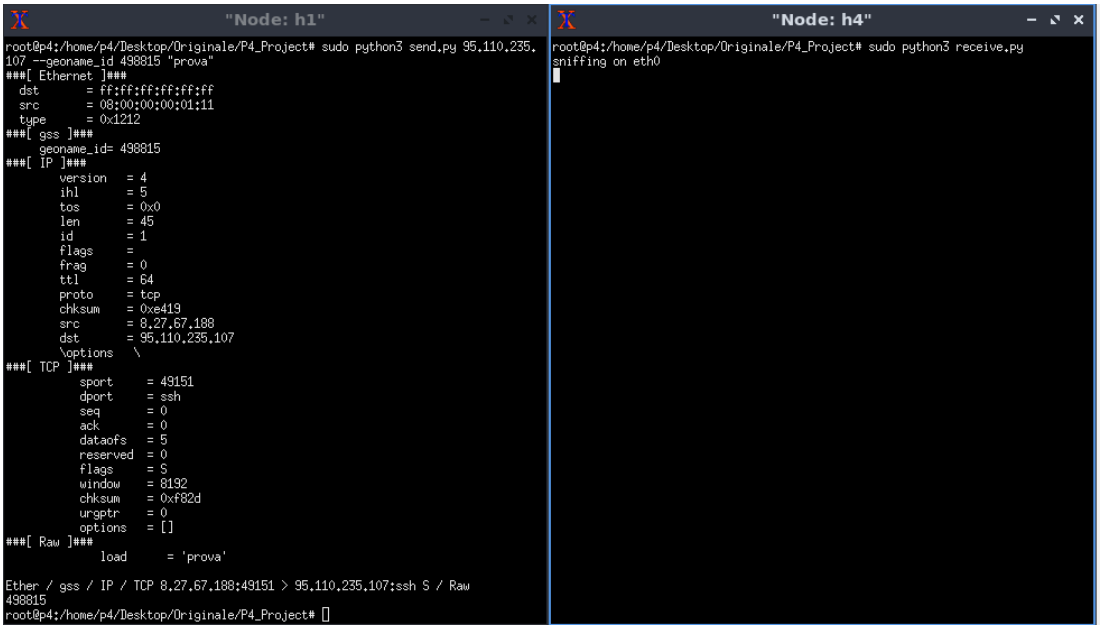


Figure 8.38: h1 sending to h4 with spoofed value of geonameId and spoofing check enabled on controller

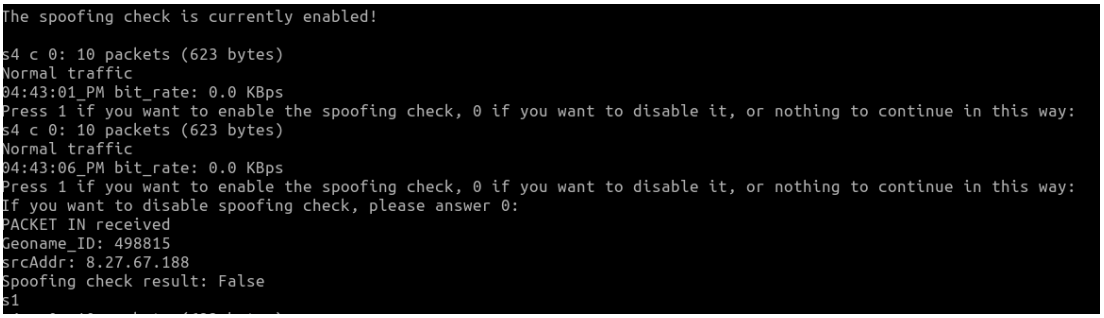


Figure 8.39: Controller discarding after spoofing detected

In the second case the packet arrives to the controller (PacketIn and result of the spoofing check messages are printed by the controller in figure 8.39) because of the spoofing made on the geonameId, granting this field not being contained in the black list, otherwise it would be simply discarded by the switch without involving controller.

In the last case, h3 sends same packet to h4 (one with no spoofing and one with spoofing in figures 8.40 and 8.42); the true geonameId of h3 is not contained in the black list, so this packet will not be discarded by the switch. The controller will check if the geonameId is the true one or not (also the result of the check is printed as shown in figures 8.41 and 8.43); if the check is true the packet will arrive to h4 (figure 8.40), otherwise the controller will drop it.

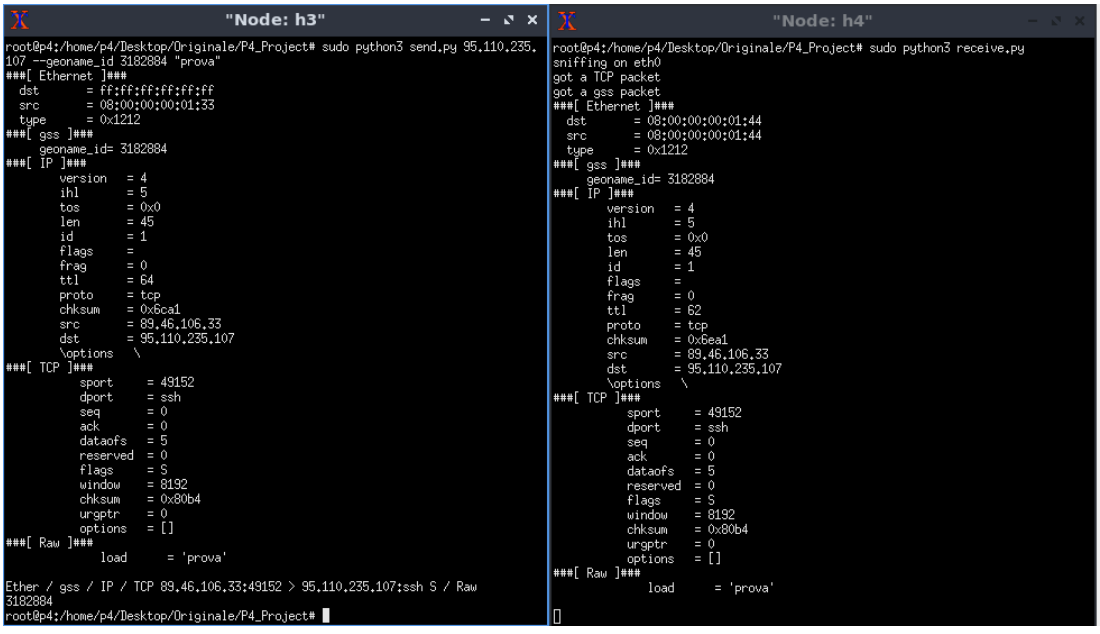


Figure 8.40: h3 sending to h4 with no spoofing on geonameId

Figure 8.41: Controller forwarding the packet after check result is True

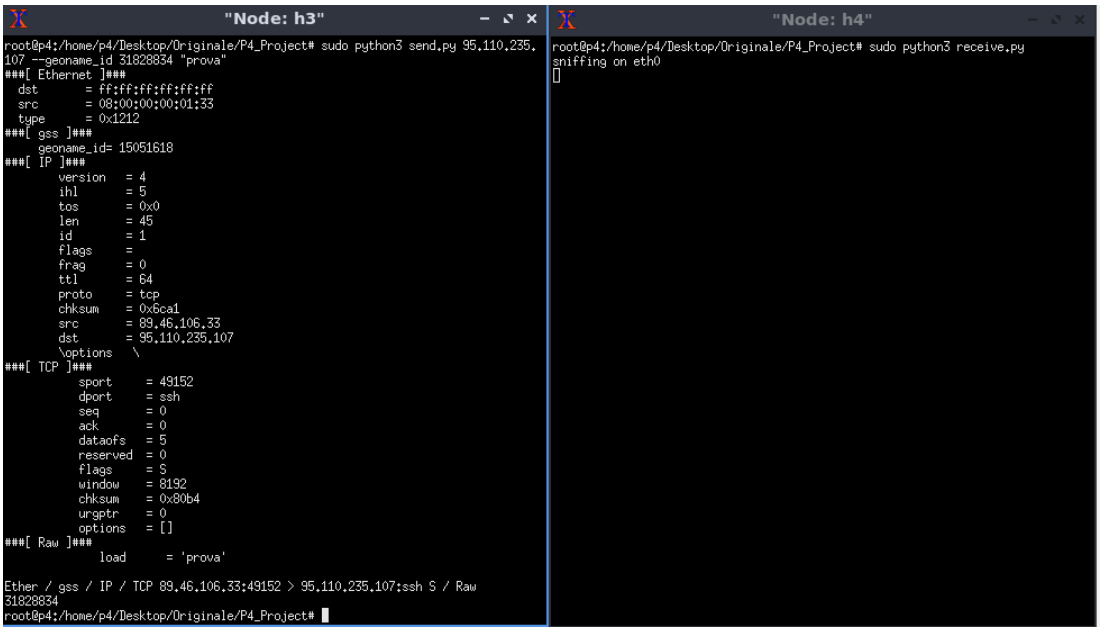


Figure 8.42: h3 sending to h4 with spoofing on geonameId

Figure 8.43: Controller discarding the packet after check result is False

CHAPTER 9

Conclusion

This project has demonstrated the potential of using P4 programming language to define the behavior of network switches in a Mininet-based virtual network. The use of a Python controller has also been shown to be a powerful tool for network management, specifically in the areas of DoS and DDoS detection, spoofing detection, firewall rule implementation, and giving forwarding rules to switches by using P4Runtime.The experiments conducted on the network have shown that the P4-based switch behavior and Python controller are effective in detecting and mitigating various network security threats, such as DDoS attacks and spoofing. Additionally, the firewall rules and forwarding rules implemented on the network have been shown to effectively control and manage the flow of traffic through the network. In conclusion, this project has highlighted the potential of using P4 and a Python controller for network management and security. The approach used in this project could be applied in real-world networks to improve the performance and security of networks.

APPENDIX A

Appendix A

```
/* -*- P4_16 -*- */

#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
const bit<8>  TYPE_TCP  = 0x6;
const bit<16> SSH_DEFAULT_PORT = 0x16;
const bit<16> TYPE_GSS = 0x1212;
#define CPU_PORT 255
/*************************************************************************
********************* H E A D E R S  *********************************
*************************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}
header GSS_t {
    bit<24> geoname_id;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

header tcp_t{
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<1>  cwr;
    bit<1>  ece;
    bit<1>  urg;
    bit<1>  ack;
```

```
54      bit<1>  psh;
55      bit<1>  rst;
56      bit<1>  syn;
57      bit<1>  fin;
58      bit<16> window;
59      bit<16> checksum;
60      bit<16> urgentPtr;
61  }
62
63  struct metadata {
64  }
65
66  @controller_header("packet_in")
67  header packet_in_t {
68      bit<24> geoname_id; //geoname_id
69      bit<32> srcAddr;
70  }
71
72  @controller_header("packet_out")
73  header packet_out_t {
74    bit<24> reason_id;
75  }
76
77  struct headers {
78      packet_in_t  packetin;
79      packet_out_t packetout;
80      ethernet_t   ethernet;
81      GSS_t         gss;
82      ipv4_t       ipv4;
83      tcp_t        tcp;
84      }
85
86
87  /*************************************************************************
88  ********************** P A R S E R  ***************************************
89  *************************************************************************/
90
91  parser MyParser(packet_in packet,
92                  out headers hdr,
93                  inout metadata meta,
94                  inout standard_metadata_t standard_metadata) {
95
96      state start {
97    transition select(standard_metadata.ingress_port){
98              CPU_PORT: parse_packet_out;
99                default: parse_ethernet;
100         }
101     }
102     state parse_packet_out {
103         packet.extract(hdr.packetout);
104         transition parse_ethernet;
105     }
106
107     state parse_ethernet {
108         packet.extract(hdr.ethernet);
109         transition select(hdr.ethernet.etherType) {
110             TYPE_IPV4: parse_ipv4;
111             TYPE_GSS: parse_gss;
112             default: accept;
113         }
114     }
115     state parse_gss {
116        packet.extract(hdr.gss);
117          transition parse_ipv4;
118      }
119
```

```
120     state parse_ipv4 {
121         packet.extract(hdr.ipv4);
122         transition select(hdr.ipv4.protocol){
123         TYPE_TCP: parse_tcp;
124         default: accept;
125         }
126     }
127
128     state parse_tcp {
129         packet.extract(hdr.tcp);
130          transition accept;
131     }
132
133 }
134
135 /*************************************************************************
136 ************   C H E C K S U M    V E R I F I C A T I O N   ************
137 *************************************************************************/
138
139 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
140     apply {  }
141 }
142
143
144 /*************************************************************************
145 **************  I N G R E S S   P R O C E S S I N G   *******************
146 *************************************************************************/
147
148 control MyIngress(inout headers hdr,
149                   inout metadata meta,
150                   inout standard_metadata_t standard_metadata) {
151
152     action drop() {
153         mark_to_drop(standard_metadata);
154     }
155
156     action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
157         standard_metadata.egress_spec = port;
158         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
159         hdr.ethernet.dstAddr = dstAddr;
160         hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
161     }
162     action secure_forward(egressSpec_t port) {
163         standard_metadata.egress_spec = port;
164         hdr.ipv4.ttl = hdr.ipv4.ttl + 1;
165     }
166
167     table ipv4_lpm {
168         key = {
169             hdr.ipv4.dstAddr: lpm;
170         }
171         actions = {
172             ipv4_forward;
173             drop;
174             NoAction;
175         }
176         size = 1024;
177         default_action = drop();
178     }
179     table ipv4_lpm_for_ssh {
180         key = {
181             hdr.ipv4.dstAddr: lpm;
182         }
183         actions = {
184             ipv4_forward;
185             drop;
```

```
186             NoAction;
187         }
188         size = 1024;
189         default_action = drop();
190     }
191     table secure_lpm {
192         key = {
193             hdr.ipv4.dstAddr: lpm;
194         }
195         actions = {
196             secure_forward;
197             NoAction;
198         }
199         size = 1024;
200         default_action = NoAction();
201     }
202     table firewall_exact {
203         key = {
204             hdr.gss.geoname_id: exact;
205         }
206         actions = {
207             drop;
208             NoAction;
209         }
210         size = 1024;
211         default_action = NoAction();
212     }
213
214
215     apply {
216         if (hdr.tcp.isValid()){
217             if (hdr.tcp.dstPort != SSH_DEFAULT_PORT){
218                 ipv4_lpm.apply();
219             }
220             else{
221                 ipv4_lpm_for_ssh.apply();
222                 if (standard_metadata.ingress_port == CPU_PORT) { //packet out
223                     hdr.packetout.setInvalid();
224                     secure_lpm.apply();
225                 }
226                 if (standard_metadata.egress_spec == CPU_PORT) { // packet in
227                     hdr.packetin.setValid();
228                     hdr.packetin.geoname_id= hdr.gss.geoname_id; //geoname_id
229                     hdr.packetin.srcAddr=hdr.ipv4.srcAddr; //srcAddr
230                 }
231                 if (hdr.gss.isValid() && standard_metadata.ingress_port != CPU_PORT){
232                     firewall_exact.apply();
233                 }
234                 if (!hdr.gss.isValid()){
235                     drop();
236                 }
237             }
238         }
239         else{
240             ipv4_lpm.apply();
241         }
242     }
243 }
244
245 /*************************************************************************
246 ***************  E G R E S S   P R O C E S S I N G   *******************
247 *************************************************************************/
248
249 control MyEgress(inout headers hdr,
250                  inout metadata meta,
251                  inout standard_metadata_t standard_metadata) {
```

```
252
253     counter(1, CounterType.bytes) c;
254     apply {
255         if (hdr.ipv4.dstAddr== (bit<32>)1601104747 && standard_metadata.egress_port!=CPU_PORT
                )){
256             c.count(0);
257         }
258     }
259 }
260 /*************************************************************************
261 *************   C H E C K S U M    C O M P U T A T I O N   **************
262 *************************************************************************/
263
264 control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
265      apply {
266   update_checksum(
267       hdr.ipv4.isValid(),
268             { hdr.ipv4.version,
269             hdr.ipv4.ihl,
270               hdr.ipv4.diffserv,
271               hdr.ipv4.totalLen,
272               hdr.ipv4.identification,
273               hdr.ipv4.flags,
274               hdr.ipv4.fragOffset,
275               hdr.ipv4.ttl,
276               hdr.ipv4.protocol,
277               hdr.ipv4.srcAddr,
278               hdr.ipv4.dstAddr },
279             hdr.ipv4.hdrChecksum,
280             HashAlgorithm.csum16);
281     }
282 }
283
284 /*************************************************************************
285 ***********************  D E P A R S E R  ********************************
286 *************************************************************************/
287
288 control MyDeparser(packet_out packet, in headers hdr) {
289     apply {
290         packet.emit(hdr.packetin);
291         packet.emit(hdr.ethernet);
292         packet.emit(hdr.gss);
293         packet.emit(hdr.ipv4);
294         packet.emit(hdr.tcp);
295     }
296 }
297
298 /*************************************************************************
299 ***********************  S W I T C H  ************************************
300 *************************************************************************/
301
302 V1Switch(
303 MyParser(),
304 MyVerifyChecksum(),
305 MyIngress(),
306 MyEgress(),
307 MyComputeChecksum(),
308 MyDeparser()
309 ) main;
```

Listing A.1: switch_config.p4

```
1 import argparse
2 import grpc
3 import os
4 import sys
```

```
from time import sleep
import threading
from utils.switch import *
import json
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)), "utils/"))
import bmv2
#from switch import ShutdownAllSwitchConnections
from utils.convert import encodeNum, decodeNum, from_base64_to_ipv4, from_base64_to_decimal, encodeIPv4
from utils.helper import *
#import helper
from google.protobuf.json_format import MessageToDict
from geoip2 import *
import geoip2.errors
import geoip2.database

def json_read_table_entries(json_file):
    with open(json_file, 'r') as f:
        table= json.load(f)
        f.close()
    table_entries = table['table_entries']
    table_entries.pop(0)
    return table_entries

    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.secure_lpm",
        match_fields={
            "hdr.ipv4.dstAddr": (dst_ip_addr, 32)
        },
        action_name="MyIngress.secure_forward",
        action_params={
            "port": port
        })

def table_update_spoofing_on(p4info_helper, switch_connection_list):
    for switch_connection in switch_connection_list:
        json_file="table_entries_"+switch_connection.name+".json"
        table_entries=json_read_table_entries(json_file)
        for entry in table_entries:
            table_entry=p4info_helper.buildTableEntry(table_name=entry["table_name"],
            #match_fields={list(entry["match_fields"].keys())[0] : (entry["match_fields"]["hdr.ipv4.dstAddr"][0], entry["match_fields"]["hdr.ipv4.dstAddr"][1])},
            match_fields={"hdr.ipv4.dstAddr" : (entry["match_fields"]["hdr.ipv4.dstAddr"][0], entry["match_fields"]["hdr.ipv4.dstAddr"][1])},
            action_name=entry["action_name"],
            action_params={"dstAddr":entry["action_params"]["dstAddr"], "port": 255})
            switch_connection.ModifyTableEntry(table_entry)

def table_update_spoofing_off(p4info_helper, switch_connection_list):
    for switch_connection in switch_connection_list:
        json_file="table_entries_"+switch_connection.name+".json"
        table_entries=json_read_table_entries(json_file)
        for entry in table_entries:
            table_entry=p4info_helper.buildTableEntry(table_name=entry["table_name"],
            match_fields={list(entry["match_fields"].keys())[0] : (entry["match_fields"]["hdr.ipv4.dstAddr"][0], entry["match_fields"]["hdr.ipv4.dstAddr"][1])},
            action_name=entry["action_name"],
            action_params={"dstAddr":entry["action_params"]["dstAddr"], "port": entry["action_params"]["port"]})
            switch_connection.ModifyTableEntry(table_entry)

def writeIpv4Rules(p4info_helper,table, sw_id, dst_ip_addr, dst_eth_addr, port):
    table_entry = p4info_helper.buildTableEntry(
        table_name=table,
        match_fields={"hdr.ipv4.dstAddr": (dst_ip_addr, 32)},
        action_name="MyIngress.ipv4_forward",
```

```
66          action_params={
67              "dstAddr": dst_eth_addr,
68              "port": port},
69      )
70      sw_id.WriteTableEntry(table_entry)
71      print("Installed ipv4 rule on %s" % sw_id.name)
72
73  def writeIpv4SecureRules(p4info_helper, sw_id, dst_ip_addr, port):
74      table_entry = p4info_helper.buildTableEntry(
75          table_name="MyIngress.secure_lpm",
76          match_fields={"hdr.ipv4.dstAddr": (dst_ip_addr, 32)},
77          action_name="MyIngress.secure_forward",
78          action_params={"port": port},
79      )
80      sw_id.WriteTableEntry(table_entry)
81      print("Installed ipv4 secure rule on %s" % sw_id.name)
82
83
84  def writeFirewallRules(p4info_helper, ingress_sw, geoname_id):
85          table_entry = p4info_helper.buildTableEntry(
86              table_name="MyIngress.firewall_exact",
87              match_fields={
88                  "hdr.gss.geoname_id": geoname_id
89              },
90              action_name="MyIngress.drop",
91              action_params={
92              })
93          ingress_sw.WriteTableEntry(table_entry)
94          print("Installed ifirewall rule on %s" % ingress_sw.name)
95
96
97  def readTableRules(p4info_helper, sw):
98      """
99      Reads the table entries from all tables on the switch.
100     :param p4info_helper: the P4Info helper
101     :param sw: the switch connection
102     """
103     print("\n----- Reading tables rules for %s -----" % sw.name)
104     for response in sw.ReadTableEntries():
105         for entity in response.entities:
106             entry = entity.table_entry
107             print(entry)
108
109 def check_spoofing(ipv4_address, geoname_id):
110     result=False
111     try:
112         reader=geoip2.database.Reader("/home/p4/Desktop/Originale/GeoLite2-City_20221004/
                GeoLite2-City.mmdb")
113         response=reader.city(ipv4_address)
114         real_geoname_id=response.city.geoname_id
115         #print(response.city.name)
116         #print(response.city.geoname_id)
117         if geoname_id == real_geoname_id:
118             result=True
119     except geoip2.errors.AddressNotFoundError:
120         print("Address not present in the database")
121     reader.close()
122     return result
123
124 def printCounter(p4info_helper, sw, counter_name, index):
125     """
126     Reads the specified counter at the specified index from the switch. In our
127     program, the index is the tunnel ID. If the index is 0, it will return all
128     values from the counter.
129     :param p4info_helper: the P4Info helper
130     :param sw:  the switch connection
```

```
131     :param counter_name: the name of the counter from the P4 program
132     :param index: the counter index (in our case, the tunnel ID)
133     """
134     for response in sw.ReadCounters(p4info_helper.get_counters_id(counter_name), index):
135         for entity in response.entities:
136             counter = entity.counter_entry
137             print("%s %s %d: %d packets (%d bytes)" % (
138                 sw.name, counter_name, index,
139                 counter.data.packet_count, counter.data.byte_count
140             ))
141             return counter.data.byte_count
142
143
144
145 def printGrpcError(e):
146     print("gRPC Error:", e.details(), end="")
147     status_code = e.code()
148     print("(%s)" % status_code.name, end="")
149     traceback = sys.exc_info()[2]
150     print("[%s:%d]" % (traceback.tb_frame.f_code.co_filename, traceback.tb_lineno))
151
152
153 def thread1(p4info_helper, switch_connection):
154     while True:
155         packetin = switch_connection.PacketIn()# Packet in!
156         if packetin is not None:
157             print("PACKET IN received")
158             #print(packetin)
159             packet = packetin.packet.payload
160             d = MessageToDict(packetin.packet)
161             geoname_id=from_base64_to_decimal(d['metadata'][0]['value'])
162             srcAddr=from_base64_to_ipv4(d['metadata'][1]['value'])
163             print("Geoname_ID: %d\nsrcAddr: %s" % (geoname_id,srcAddr))
164             print("Spoofing check result: %s" %(check_spoofing(srcAddr, geoname_id)))
165             print(switch_connection.name)
166             if check_spoofing(srcAddr, geoname_id):
167                 packetout = p4info_helper.buildPacketOut(
168                     payload=packet,
169                     metadata={
170                         1: encodeNum(1, 16) #the reason_id assumes always the value 1, useful
                                in future for new features
171                     },
172                 )
173                 print("send PACKET OUT")
174                 switch_connection.PacketOut(packetout)
175
176
177 def thread2(p4info_helper, switch_connection_list):
178     check_spoofing=0
179     print("The spoofing check is currently disabled!\n")
180     num=int(input("If you want to enable spoofing check, please answer 1: \n"))
181     s1=switch_connection_list[0]
182     s2=switch_connection_list[1]
183     s3=switch_connection_list[2]
184     s4=switch_connection_list[3]
185     while True:
186         if num == 1:
187             table_update_spoofing_on(p4info_helper, switch_connection_list)
188             print("The spoofing check is currently enabled!\n")
189             sleep(10)
190             num=int(input("If you want to disable spoofing check, please answer 0: \n"))
191
192         elif num == 0:
193             table_update_spoofing_off(p4info_helper, switch_connection_list)
194             print("The spoofing check is currently disabled!\n")
195             sleep(10)
```

```
196             num=int(input("If you want to enable spoofing check answer 1: \n"))
197         check_spoofing=num
198
199 def thread3(p4info_helper, switch_connection_list):
200     s4=switch_connection_list[3]
201     old_counter=0
202     new_counter=printCounter(p4info_helper, s4, "c", 0)
203     data_rate=(new_counter-old_counter)/5000
204     localtime=time.localtime()
205     result=time.strftime("%I:%M:%S_%p", localtime)
206     data_rate="bit_rate: %s KBps" % (data_rate)
207     print(result, data_rate)
208     sleep(5)
209     while True:
210         old_counter=new_counter
211         trashold=1000 #1 MBps
212         new_counter=printCounter(p4info_helper, s4,  "c", 0)
213         data_rate=(new_counter-old_counter)/5000
214         if data_rate > trashold:
215             print("DDoS detected")
216         else:
217             print("Normal traffic")
218         localtime=time.localtime()
219         result=time.strftime("%I:%M:%S_%p", localtime)
220         data_rate="bit_rate: %s KBps" % (data_rate)
221
222         print(result, data_rate)
223         print("Press 1 if you want to enable the spoofing check, 0 if you want to disable it,
                or nothing to continue in this way: ")
224         sleep(5)
225
226
227 def main(p4info_file_path, bmv2_file_path):
228     # Instantiate a P4Runtime helper from the p4info file
229     p4info_helper = P4InfoHelper(p4info_file_path)
230     try:
231         # Create a switch connection object for s1, s2, s3 and s4;
232         # this is backed by a P4Runtime gRPC connection.
233         # Also, dump all P4Runtime messages sent to switch to given txt files.
234         s1 = bmv2.Bmv2SwitchConnection(
235             name="s1",
236             address="0.0.0.0:50051",
237             device_id=1,
238             proto_dump_file="p4runtime1.log",
239         )
240         s2 = bmv2.Bmv2SwitchConnection(
241             name="s2",
242             address="0.0.0.0:50052",
243             device_id=2,
244             proto_dump_file="p4runtime2.log",
245         )
246         s3 = bmv2.Bmv2SwitchConnection(
247             name="s3",
248             address="0.0.0.0:50053",
249             device_id=3,
250             proto_dump_file="p4runtime3.log",
251         )
252         s4 = bmv2.Bmv2SwitchConnection(
253             name="s4",
254             address="0.0.0.0:50054",
255             device_id=4,
256             proto_dump_file="p4runtime4.log",
257         )
258         # Send master arbitration update message to establish this controller as
259         # master (required by P4Runtime before performing any other write operation)
260         if s1.MasterArbitrationUpdate() == None:
```

```
261             print("Failed to establish the connection")
262         # Install the P4 program on the switches
263         s1.SetForwardingPipelineConfig(
264             p4info=p4info_helper.p4info, bmv2_json_file_path=bmv2_file_path
265         )
266         print("Installed P4 Program using SetForwardingPipelineConfig on s1")
267 
268         if s2.MasterArbitrationUpdate() == None:
269             print("Failed to establish the connection")
270         s2.SetForwardingPipelineConfig(
271             p4info=p4info_helper.p4info, bmv2_json_file_path=bmv2_file_path
272         )
273         print("Installed P4 Program using SetForwardingPipelineConfig on s2")
274 
275         if s3.MasterArbitrationUpdate() == None:
276             print("Failed to establish the connection")
277         s3.SetForwardingPipelineConfig(
278             p4info=p4info_helper.p4info, bmv2_json_file_path=bmv2_file_path
279         )
280         print("Installed P4 Program using SetForwardingPipelineConfig on s3")
281 
282         if s4.MasterArbitrationUpdate() == None:
283             print("Failed to establish the connection")
284         s4.SetForwardingPipelineConfig(
285             p4info=p4info_helper.p4info, bmv2_json_file_path=bmv2_file_path
286         )
287         print("Installed P4 Program using SetForwardingPipelineConfig on s4")
288 
289         #S1
290         writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="8.27.67.188", dst_eth_addr= "08:00:00:00:01:11",port=1)
291         writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="31.28.27.50", dst_eth_addr= "08:00:00:00:00:02",port=10)
292         writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="89.46.106.33", dst_eth_addr= "08:00:00:00:00:02",port=10)
293         writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="95.110.235.107", dst_eth_addr= "08:00:00:00:00:04",port=11)
294         writeIpv4Rules(p4info_helper,  sw_id=s1, table="MyIngress.ipv4_lpm", dst_ip_addr="
                8.27.67.188", dst_eth_addr= "08:00:00:00:01:11",port=1)
295         writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm", dst_ip_addr="
                31.28.27.50", dst_eth_addr= "08:00:00:00:00:02",port=10)
296         writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm", dst_ip_addr="
                89.46.106.33", dst_eth_addr= "08:00:00:00:00:02",port=10)
297         writeIpv4Rules(p4info_helper, sw_id=s1, table="MyIngress.ipv4_lpm", dst_ip_addr="
                95.110.235.107", dst_eth_addr= "08:00:00:00:00:04",port=11)
298         writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="8.27.67.188", port=1)
299         writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="31.28.27.50", port=10)
300         writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="89.46.106.33", port=10)
301         writeIpv4SecureRules(p4info_helper,  sw_id=s1, dst_ip_addr="95.110.235.107", port=11)
302 
303         #S2
304         writeIpv4Rules(p4info_helper,  sw_id=s2, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="31.28.27.50", dst_eth_addr= "08:00:00:00:01:22",port=1)
305         writeIpv4Rules(p4info_helper, sw_id=s2, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="8.27.67.188", dst_eth_addr= "08:00:00:00:00:01",port=10)
306         writeIpv4Rules(p4info_helper, sw_id=s2, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="89.46.106.33", dst_eth_addr= "08:00:00:00:00:03",port=11)
307         writeIpv4Rules(p4info_helper, sw_id=s2, table="MyIngress.ipv4_lpm_for_ssh",
                dst_ip_addr="95.110.235.107", dst_eth_addr= "08:00:00:00:00:03",port=11)
308         writeIpv4Rules(p4info_helper,  sw_id=s2, table="MyIngress.ipv4_lpm", dst_ip_addr="
                31.28.27.50", dst_eth_addr= "08:00:00:00:01:22",port=1)
309         writeIpv4Rules(p4info_helper, sw_id=s2, table="MyIngress.ipv4_lpm", dst_ip_addr="
                8.27.67.188", dst_eth_addr= "08:00:00:00:00:01",port=10)
310         writeIpv4Rules(p4info_helper, sw_id=s2, table="MyIngress.ipv4_lpm", dst_ip_addr="
                89.46.106.33", dst_eth_addr= "08:00:00:00:00:03",port=11)
311         writeIpv4Rules(p4info_helper, sw_id=s2, table="MyIngress.ipv4_lpm", dst_ip_addr="
```

```
            95.110.235.107", dst_eth_addr= "08:00:00:00:00:03",port=11)
312         writeIpv4SecureRules(p4info_helper,  sw_id=s2, dst_ip_addr="8.27.67.188", port=10)
313         writeIpv4SecureRules(p4info_helper,  sw_id=s2, dst_ip_addr="31.28.27.50", port=1)
314         writeIpv4SecureRules(p4info_helper,  sw_id=s2, dst_ip_addr="89.46.106.33", port=11)
315         writeIpv4SecureRules(p4info_helper,  sw_id=s2, dst_ip_addr="95.110.235.107", port=11)
316
317         #S3
318         writeIpv4Rules(p4info_helper,  sw_id=s3, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="89.46.106.33", dst_eth_addr= "08:00:00:00:01:33",port=1)
319         writeIpv4Rules(p4info_helper, sw_id=s3, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="31.28.27.50", dst_eth_addr= "08:00:00:00:00:02",port=11)
320         writeIpv4Rules(p4info_helper, sw_id=s3, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="8.27.67.188", dst_eth_addr= "08:00:00:00:00:02",port=11)
321         writeIpv4Rules(p4info_helper, sw_id=s3, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="95.110.235.107", dst_eth_addr= "08:00:00:00:00:04",port=10)
322         writeIpv4Rules(p4info_helper,  sw_id=s3, table="MyIngress.ipv4_lpm", dst_ip_addr="
            89.46.106.33", dst_eth_addr= "08:00:00:00:01:33",port=1)
323         writeIpv4Rules(p4info_helper, sw_id=s3, table="MyIngress.ipv4_lpm", dst_ip_addr="
            31.28.27.50", dst_eth_addr= "08:00:00:00:00:02",port=11)
324         writeIpv4Rules(p4info_helper, sw_id=s3, table="MyIngress.ipv4_lpm", dst_ip_addr="
            8.27.67.188", dst_eth_addr= "08:00:00:00:00:02",port=11)
325         writeIpv4Rules(p4info_helper, sw_id=s3, table="MyIngress.ipv4_lpm", dst_ip_addr="
            95.110.235.107", dst_eth_addr= "08:00:00:00:00:04",port=10)
326         writeIpv4SecureRules(p4info_helper,  sw_id=s3, dst_ip_addr="89.46.106.33", port=1)
327         writeIpv4SecureRules(p4info_helper,  sw_id=s3, dst_ip_addr="8.27.67.188", port=11)
328         writeIpv4SecureRules(p4info_helper,  sw_id=s3, dst_ip_addr="31.28.27.50", port=11)
329         writeIpv4SecureRules(p4info_helper,  sw_id=s3, dst_ip_addr="95.110.235.107", port=10)
330
331         #S4
332         writeIpv4Rules(p4info_helper,  sw_id=s4, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="95.110.235.107", dst_eth_addr= "08:00:00:00:01:44",port=1)
333         writeIpv4Rules(p4info_helper, sw_id=s4, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="8.27.67.188", dst_eth_addr= "08:00:00:00:00:01",port=11)
334         writeIpv4Rules(p4info_helper, sw_id=s4, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="31.28.27.50", dst_eth_addr= "08:00:00:00:00:03",port=10)
335         writeIpv4Rules(p4info_helper, sw_id=s4, table="MyIngress.ipv4_lpm_for_ssh",
            dst_ip_addr="89.46.106.33", dst_eth_addr= "08:00:00:00:00:03",port=10)
336         writeIpv4Rules(p4info_helper,  sw_id=s4, table="MyIngress.ipv4_lpm", dst_ip_addr="
            95.110.235.107", dst_eth_addr= "08:00:00:00:01:44",port=1)
337         writeIpv4Rules(p4info_helper, sw_id=s4, table="MyIngress.ipv4_lpm", dst_ip_addr="
            8.27.67.188", dst_eth_addr= "08:00:00:00:00:01",port=11)
338         writeIpv4Rules(p4info_helper, sw_id=s4, table="MyIngress.ipv4_lpm", dst_ip_addr="
            31.28.27.50", dst_eth_addr= "08:00:00:00:00:03",port=10)
339         writeIpv4Rules(p4info_helper, sw_id=s4, table="MyIngress.ipv4_lpm", dst_ip_addr="
            89.46.106.33", dst_eth_addr= "08:00:00:00:00:03",port=10)
340         writeIpv4SecureRules(p4info_helper,  sw_id=s4, dst_ip_addr="95.110.235.107", port=1)
341         writeIpv4SecureRules(p4info_helper,  sw_id=s4, dst_ip_addr="8.27.67.188", port=11)
342         writeIpv4SecureRules(p4info_helper,  sw_id=s4, dst_ip_addr="31.28.27.50", port=10)
343         writeIpv4SecureRules(p4info_helper,  sw_id=s4, dst_ip_addr="89.46.106.33", port=10)
344
345
346
347         writeFirewallRules(p4info_helper, s1, 498817) #San Pietroburgo
348         writeFirewallRules(p4info_helper, s1,1796236) #Shangai
349         writeFirewallRules(p4info_helper, s2, 498817) #San Pietroburgo
350         writeFirewallRules(p4info_helper, s2,1796236) #Shangai
351         writeFirewallRules(p4info_helper, s3, 498817) #San Pietroburgo
352         writeFirewallRules(p4info_helper, s3,1796236) #Shangai
353         printCounter(p4info_helper, s4, "c", 0)
354
355         # read all table rules
356         #readTableRules(p4info_helper, s1)
357         lista=[s1, s2, s3, s4]
358         t_s1 = threading.Thread(target=thread1, args=(p4info_helper, s1))
359         t_s2 = threading.Thread(target=thread1, args=(p4info_helper, s2))
360         t_s3 = threading.Thread(target=thread1, args=(p4info_helper, s3))
```

```
361         t_s4 = threading.Thread(target=thread1, args=(p4info_helper, s4))
362         t2 = threading.Thread(target=thread2, args=(p4info_helper, lista))
363         t3 = threading.Thread(target=thread3, args=(p4info_helper, lista))
364
365
366         # starting threads
367         t_s1.start()
368         t_s2.start()
369         t_s3.start()
370         t_s4.start()
371         t2.start()
372         t3.start()
373
374         t_s1.join()
375         t_s2.join()
376         t_s3.join()
377         t_s4.join()
378         t2.join()
379         t3.join()
380
381     except KeyboardInterrupt:
382         print(" Shutting down.")
383     except grpc.RpcError as e:
384         printGrpcError(e)
385     ShutdownAllSwitchConnections()
386 if __name__ == "__main__":
387     parser = argparse.ArgumentParser(description="P4Runtime Controller")
388     parser.add_argument(
389         "--p4info",
390         help="p4info proto in text format from p4c",
391         type=str,
392         action="store",
393         required=False,
394         default="./switch_config.p4info.txt",
395     )
396     parser.add_argument(
397         "--bmv2-json",
398         help="BMv2 JSON file from p4c",
399         type=str,
400         action="store",
401         required=False,
402         default="./switch_config.json",
403     )
404     args = parser.parse_args()
405     if not os.path.exists(args.p4info):
406         parser.print_help()
407         print("\np4info file %s not found!" % args.p4info)
408         parser.exit(1)
409     if not os.path.exists(args.bmv2_json):
410         parser.print_help()
411         print("\nBMv2 JSON file %s not found!" % args.bmv2_json)
412         parser.exit(2)
413     main(args.p4info, args.bmv2_json)
```

Listing A.2: controller.py

```
1  from mininet.net import Mininet
2  from mininet.node import Switch, Host
3  from mininet.log import setLogLevel, info
4
5
6  class P4Host(Host):
7      def config(self, **params):
8          r = super(Host, self).config(**params)
9
10         self.defaultIntf().rename("eth0")
```

```
11
12          for off in ["rx", "tx", "sg"]:
13              cmd = "/sbin/ethtool --offload eth0 %s off" % off
14              self.cmd(cmd)
15
16          # disable IPv6
17          self.cmd("sysctl -w net.ipv6.conf.all.disable_ipv6=1")
18          self.cmd("sysctl -w net.ipv6.conf.default.disable_ipv6=1")
19          self.cmd("sysctl -w net.ipv6.conf.lo.disable_ipv6=1")
20
21          return r
22
23      def describe(self):
24          print("**********")
25          print(self.name)
26          print(
27              "default interface: %s\t%s\t%s"
28              % (
29                  self.defaultIntf().name,
30                  self.defaultIntf().IP(),
31                  self.defaultIntf().MAC(),
32              )
33          )
34          print("**********")
35
36  class P4GrpcSwitch(Switch):
37      """P4 virtual switch"""
38
39      device_id = 0
40
41      def __init__(
42          self,
43          name,
44          sw_path=None,
45          json_path=None,
46          thrift_port=None,
47          grpc_port=None,
48          pcap_dump=False,
49          verbose=False,
50          device_id=None,
51          enable_debugger=False,
52          cpu_port=None,
53          **kwargs
54      ):
55          Switch.__init__(self, name, **kwargs)
56          assert sw_path
57          self.sw_path = sw_path
58          self.json_path = json_path
59          self.verbose = verbose
60          self.thrift_port = thrift_port
61          self.grpc_port = grpc_port
62          self.enable_debugger = enable_debugger
63          self.cpu_port = cpu_port
64          if device_id is not None:
65              self.device_id = device_id
66              P4GrpcSwitch.device_id = max(P4GrpcSwitch.device_id, device_id)
67          else:
68              self.device_id = P4GrpcSwitch.device_id
69              P4GrpcSwitch.device_id += 1
70
71      @classmethod
72      def setup(cls):
73          pass
74
75      def start(self, controllers):
76          "Start up a new P4 switch"
```

```
        print("Starting P4 switch", self.name)
        args = [self.sw_path]
        for port, intf in self.intfs.items():
            if not intf.IP():
                args.extend(["-i", str(port) + "@" + intf.name])
        if self.thrift_port:
            args.extend(["--thrift-port", str(self.thrift_port)])

        args.extend(["--device-id", str(self.device_id)])
        P4GrpcSwitch.device_id += 1
        if self.json_path:
            args.append(self.json_path)
        else:
            args.append("--no-p4")

        args.append("--log-flush --log-level trace --log-file %s.log" % self.name)
        if self.grpc_port:
            args.append(
                "-- --grpc-server-addr 0.0.0.0:"
                + str(self.grpc_port)
                + " --cpu-port "
                + self.cpu_port
            )
        print(" ".join(args))
        self.cmd(" ".join(args) + " > %s.log 2>&1 &" % self.name)
        print("switch has been started")

    def stop(self):
        "Terminate IVS switch."
        self.cmd("kill %" + self.sw_path)
        self.cmd("wait")
        self.deleteIntfs()

    def attach(self, intf):
        "Connect a data port"
        assert 0

    def detach(self, intf):
        "Disconnect a data port"
        assert 0
```

Listing A.3: p4_mininet.py

```
from mininet.net import Mininet
from mininet.topo import Topo
from mininet.log import setLogLevel, info
from mininet.cli import CLI
from mininet.link import TCLink
from p4_mininet import P4Host, P4GrpcSwitch
import json
import argparse
import subprocess
import sys
import os
import psutil
parser = argparse.ArgumentParser(description="Mininet demo")
parser.add_argument(
    "--num-hosts",
    help="Number of hosts to connect to switch",
    type=int,
    action="store",
    default=1,
)
parser.add_argument(
    "--p4-file", help="Path to P4 file", type=str, action="store", required=False
)
```

```
args = parser.parse_args()
def get_all_virtual_interfaces():
    try:
        return (
            subprocess.check_output(
                ["ip addr | grep s.-eth. | cut -d':' -f2 | cut -d'@' -f1"], shell=True
            )
            .decode(sys.stdout.encoding)
            .splitlines()
        )
    except subprocess.CalledProcessError as e:
        print("Cannot retrieve interfaces.")
        print(e)
        return ""

class MultiSwitchTopo(Topo):
    "Single switch connected to n (< 256) hosts."
    def __init__(self, topo_file, sw_path, json_path, n, **opts):
        # Initialize topology and default options
        Topo.__init__(self, **opts)
        with open(topo_file, 'r') as f:
            topo = json.load(f)
        hosts = topo['hosts']
        switches = topo['switches']
        links=topo['links']
        for sw in switches:
            switch = self.addSwitch(
                switches[sw]["name"],
                sw_path=sw_path,
                json_path=json_path,
                grpc_port=switches[sw]["grpc_port"],
                thrift_port=int(switches[sw]["thrift_port"]),
                device_id=int(switches[sw]["device_id"]),
                cpu_port="255",
            )
        for h in hosts:
            host=self.addHost(h, ip=hosts[h]["ip"],mac =hosts[h]["mac"])
        for link in links:
            if len(link) == 3:
                host = link[0]
                switch = link[1]
                port=link[2]
                self.addLink(host, switch, port)
            if len(link) == 4:
                switch1 = link[0]
                switch2 = link[1]
                port1=link[2]
                port2=link[3]
                self.addLink(switch1, switch2, port1, port2)
def main():
    num_hosts = int(args.num_hosts)
    result = os.system(
        "p4c --target bmv2 --arch v1model --p4runtime-files switch_config.p4info.txt "
        + args.p4_file
    )
    p4_file = args.p4_file.split("/")[-1]
    json_file = p4_file.split(".")[0] + ".json"
    topo = MultiSwitchTopo("topology.json", "simple_switch_grpc", json_file, num_hosts)
    net = Mininet(
        topo=topo, host=P4Host, switch=P4GrpcSwitch, link=TCLink, controller=None
    )
    net.start()
    hosts=net.hosts
    with open('topology.json', 'r') as f:
        topology = json.load(f)
    hosts_dict=topology["hosts"]
```

```
    for host in hosts:
        name=host.name
        host_inf=hosts_dict.get(name)
        commands_string= host_inf.get("commands")
        host.cmd(commands_string[0])
        host.cmd(commands_string[1])

    interfaces = get_all_virtual_interfaces()
    for i in interfaces:
        if i != "":
            os.system("ip link set {} mtu 1600 > /dev/null".format(i))
            os.system("ethtool --offload {} rx off  tx off > /dev/null".format(i))
    net.staticArp()
    if result != 0:
        print("Error while compiling!")
        exit()
    switch_running = "simple_switch_grpc" in (p.name() for p in psutil.process_iter())
    if switch_running == False:
        print("The switch didnt start correctly! Check the path to your P4 file!!")
        exit()
    print("Starting mininet!")
    print('')
    print('==========================================================================')
    print('Welcome to the BMV2 Mininet CLI!')
    print('==========================================================================')
    print('Your P4 program is installed into the BMV2 software switch')
    print('and your initial runtime configuration is loaded. You can interact')
    print('with the network using the mininet CLI below.')
    print('')
    print('To inspect or change the switch configuration, connect to')
    print('its CLI from your host operating system using this command:')
    print('  simple_switch_CLI --thrift-port <switch thrift port>')
    print('')
    CLI(net)
    os.system("rm *.log*")
    os.system("sudo mn -c")
if __name__ == "__main__":
    setLogLevel("info")
    main()
```

Listing A.4: run_mininet.py

```
from queue import Queue
from abc import abstractmethod
from datetime import datetime
import threading
import grpc
from p4.v1 import p4runtime_pb2
from p4.v1 import p4runtime_pb2_grpc
from p4.tmp import p4config_pb2
import time

MSG_LOG_MAX_LEN = 1024

# List of all active connections
connections = []


def ShutdownAllSwitchConnections():
    for c in connections:
        c.shutdown()


class SwitchConnection(object):
    def __init__(
        self, name=None, address="127.0.0.1:50051", device_id=0, proto_dump_file=None
```

```
    ):
        self.name = name
        self.address = address
        self.device_id = device_id
        self.p4info = None
        self.channel = grpc.insecure_channel(self.address)
        if proto_dump_file is not None:
            interceptor = GrpcRequestLogger(proto_dump_file)
            self.channel = grpc.intercept_channel(self.channel, interceptor)
        self.client_stub = p4runtime_pb2_grpc.P4RuntimeStub(self.channel)
        # create requests queue
        self.requests_stream = IterableQueue()
        # get response via requests queue
        self.stream_msg_resp = self.client_stub.StreamChannel(
            iter(self.requests_stream)
        )
        self.proto_dump_file = proto_dump_file
        connections.append(self)

    @abstractmethod
    def buildDeviceConfig(self, **kwargs):
        return p4config_pb2.P4DeviceConfig()

    def shutdown(self):
        self.requests_stream.close()
        self.stream_msg_resp.cancel()

    def MasterArbitrationUpdate(self, dry_run=False, **kwargs):
        request = p4runtime_pb2.StreamMessageRequest()
        request.arbitration.device_id = self.device_id
        request.arbitration.election_id.high = 0
        request.arbitration.election_id.low = 1

        if dry_run:
            print("P4Runtime MasterArbitrationUpdate: ", request)
        else:
            self.requests_stream.put(request)
            for item in self.stream_msg_resp:
                return item  # just one

    def SetForwardingPipelineConfig(self, p4info, dry_run=False, **kwargs):
        device_config = self.buildDeviceConfig(**kwargs)
        request = p4runtime_pb2.SetForwardingPipelineConfigRequest()
        request.election_id.low = 1
        request.device_id = self.device_id
        config = request.config

        config.p4info.CopyFrom(p4info)
        config.p4_device_config = device_config.SerializeToString()

        request.action = (
            p4runtime_pb2.SetForwardingPipelineConfigRequest.VERIFY_AND_COMMIT
        )
        if dry_run:
            print("P4Runtime SetForwardingPipelineConfig:", request)
        else:
            self.client_stub.SetForwardingPipelineConfig(request)

    def WriteTableEntry(self, table_entry, dry_run=False):
        request = p4runtime_pb2.WriteRequest()
        request.device_id = self.device_id
        request.election_id.low = 1
        update = request.updates.add()
        if table_entry.is_default_action:
            update.type = p4runtime_pb2.Update.MODIFY
        else:
```

```
            update.type = p4runtime_pb2.Update.INSERT
        update.entity.table_entry.CopyFrom(table_entry)
        if dry_run:
            print("P4Runtime Write:", request)
        else:
            self.client_stub.Write(request)

    def ModifyTableEntry(self, table_entry, dry_run=False):
        request = p4runtime_pb2.WriteRequest()
        request.device_id = self.device_id
        request.election_id.low = 1
        update = request.updates.add()
        update.type = p4runtime_pb2.Update.MODIFY
        update.entity.table_entry.CopyFrom(table_entry)
        if dry_run:
            print("P4Runtime Write:", request)
        else:
            self.client_stub.Write(request)

    def DeleteTableEntry(self, table_entry, dry_run=False):
        request = p4runtime_pb2.WriteRequest()
        request.device_id = self.device_id
        request.election_id.low = 1
        update = request.updates.add()
        update.type = p4runtime_pb2.Update.DELETE
        update.entity.table_entry.CopyFrom(table_entry)
        if dry_run:
            print("P4Runtime Write:", request)
        else:
            self.client_stub.Write(request)

    def ReadTableEntries(self, table_id=None, dry_run=False):
        request = p4runtime_pb2.ReadRequest()
        request.device_id = self.device_id
        entity = request.entities.add()
        table_entry = entity.table_entry
        if table_id is not None:
            table_entry.table_id = table_id
        else:
            table_entry.table_id = 0
        if dry_run:
            print("P4Runtime Read:", request)
        else:
            for response in self.client_stub.Read(request):
                yield response

    def ReadCounters(self, counter_id=None, index=None, dry_run=False):
        request = p4runtime_pb2.ReadRequest()
        request.device_id = self.device_id
        entity = request.entities.add()
        counter_entry = entity.counter_entry
        if counter_id is not None:
            counter_entry.counter_id = counter_id
        else:
            counter_entry.counter_id = 0
        if index is not None:
            counter_entry.index.index = index
        if dry_run:
            print("P4Runtime Read:", request)
        else:
            for response in self.client_stub.Read(request):
                yield response

    def PacketIn(self, dry_run=False, **kwargs):
        for item in self.stream_msg_resp:
            if dry_run:
```

```
157                 print("P4 Runtime PacketIn: ", request)
158             else:
159                 return item
160
161     def PacketOut(self, packet, dry_run=False, **kwargs):
162         request = p4runtime_pb2.StreamMessageRequest()
163         request.packet.CopyFrom(packet)
164         if dry_run:
165             print("P4 Runtime: ", request)
166         else:
167             self.requests_stream.put(request)
168             # for item in self.stream_msg_resp:
169             return request
170
171     def WritePREEntry(self, pre_entry, dry_run=False):
172         request = p4runtime_pb2.WriteRequest()
173         request.device_id = self.device_id
174         request.election_id.low = 1
175         update = request.updates.add()
176         update.type = p4runtime_pb2.Update.INSERT
177         update.entity.packet_replication_engine_entry.CopyFrom(pre_entry)
178         if dry_run:
179             print("P4Runtime Write:", request)
180         else:
181             self.client_stub.Write(request)
182
183
184 class GrpcRequestLogger(
185     grpc.UnaryUnaryClientInterceptor, grpc.UnaryStreamClientInterceptor
186 ):
187     """Implementation of a gRPC interceptor that logs request to a file"""
188
189     def __init__(self, log_file):
190         self.log_file = log_file
191         with open(self.log_file, "w") as f:
192             # Clear content if it exists.
193             f.write("")
194
195     def log_message(self, method_name, body):
196         with open(self.log_file, "a") as f:
197             ts = datetime.utcnow().strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]
198             msg = str(body)
199             f.write("\n[%s] %s\n----------\n" % (ts, method_name))
200             if len(msg) < MSG_LOG_MAX_LEN:
201                 f.write(str(body))
202             else:
203                 f.write("Message too long (%d bytes)! Skipping log...\n" % len(msg))
204             f.write("----------\n")
205
206     def intercept_unary_unary(self, continuation, client_call_details, request):
207         self.log_message(client_call_details.method, request)
208         return continuation(client_call_details, request)
209
210     def intercept_unary_stream(self, continuation, client_call_details, request):
211         self.log_message(client_call_details.method, request)
212         return continuation(client_call_details, request)
213
214
215 class IterableQueue(Queue):
216     _sentinel = object()
217
218     def __iter__(self):
219         return iter(self.get, self._sentinel)
220
221     def close(self):
222         self.put(self._sentinel)
```

Listing A.5: switch.py

Bibliography

[1] gRPC Authors. *Introduction to gRPC*. Available on line. 2022. URL: https://grpc.io/docs/what-is-grpc/introduction/.

[2] The P4 Language Consortium. *$P4_{16}$ Language Specification*. Available on line. 2022. URL: https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html.

[3] Mininet Project Contributors. *Mininet, An Instant Virtual Network on your Laptop*. Available on line. 2022. URL: http://mininet.org/.

[4] Jorge Crichigno. *Principal Investigator: Jorge Crichigno P4 PROGRAMMABLE DATA PLANES: APPLICATIONS STATEFUL ELEMENTS AND, , CUSTOM PACKET PROCESSING*. Available on line. 2022. URL: http://ce.sc.edu/cyberinfra/workshops/Material/P4-Apps.pdf.

[5] Cybersecurity and Infrastructure Security Agency (CISA). *Understanding Denial-of-Service Attacks*. Available on line. 2022. URL: https://www.cisa.gov/uscert/ncas/tips/ST04-015.

[6] Jon Dugan et al. "iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks." In: *URL: https://github. com/esnet/iperf* (2014).

[7] Maxmind. *GeoIP2 Databases*. Available on line. 2023. URL: https://www.maxmind.com/en/geoip2-databases.

[8] P4 org. *P4 Runtime Spec*. Available on line. 2022. URL: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html#sec-single-remote-controller.

[9] Scapy. *Scapy documentation*. Available on line. 2020. URL: https://scapy.readthedocs.io/en/latest/.