UNIVERSITY OF
SOUTH CAROLINA

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Book Version: 04-20-2023

Principal Investigator: Jorge Crichigno

Contents

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 1: Introduction to Mininet

Document Version: 01-25-2022

Contents

Overview

This lab provides an introduction to Mininet, a virtual testbed used for testing network tools and protocols. It demonstrates how to invoke Mininet from the command-line interface (CLI) utility and how to build and emulate topologies using a graphical user interface (GUI) application.

Objectives

By the end of this lab, you should be able to:

1. Understand what Mininet is and why it is useful for testing network topologies.
2. Invoke Mininet from the CLI.
3. Construct network topologies using the GUI.
4. Save/load Mininet topologies using the GUI.

Lab settings

The information in Table 1 provides the credentials of the Client machine.

Table 1**.** Credentials to access the Client machine.

Device	**Account**	**Password**
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction to Mininet.
2. Section 2: Invoke Mininet using the CLI.
3. Section 3: Build and emulate a network in Mininet using the GUI.

1 Introduction to Mininet

Mininet is a virtual testbed enabling the development and testing of network tools and protocols. With a single command, Mininet can create a realistic virtual network on any type of machine (Virtual Machine (VM), cloud-hosted, or native). Therefore, it provides an inexpensive solution and streamlined development running in line with production networks[1]. Mininet offers the following features:

- Fast prototyping for new networking protocols.

- Simplified testing for complex topologies without the need of buying expensive hardware.
- Realistic execution as it runs real code on the Unix and Linux kernels.
- Open-source environment backed by a large community contributing extensive documentation.

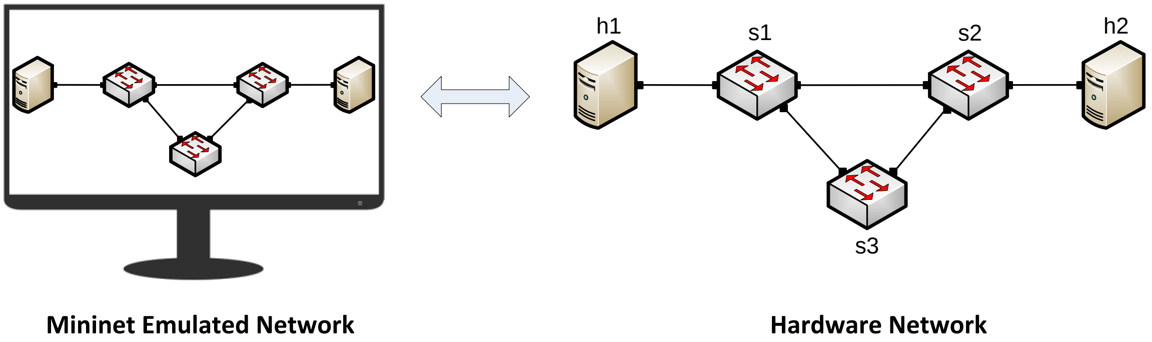


Figure 1. Hardware network vs. Mininet emulated network.

Mininet is useful for development, teaching, and research as it is easy to customize and interact with it through the CLI or the GUI. Mininet was originally designed to experiment with *OpenFlow*[2] and *Software-Defined Networking (SDN)*[3]. This lab, however, only focuses on emulating a simple network environment without SDN-based devices.

Mininet's logical nodes can be connected into networks. These nodes are sometimes called containers, or more accurately, *network namespaces*. Containers consume sufficiently fewer resources that networks of over a thousand nodes have created, running on a single laptop. A Mininet container is a process (or group of processes) that no longer has access to all the host system's native network interfaces. Containers are then assigned virtual Ethernet interfaces, which are connected to other containers through a virtual switch[4]. Mininet connects a host and a switch using a virtual Ethernet (veth) link. The veth link is analogous to a wire connecting two virtual interfaces, as illustrated below.

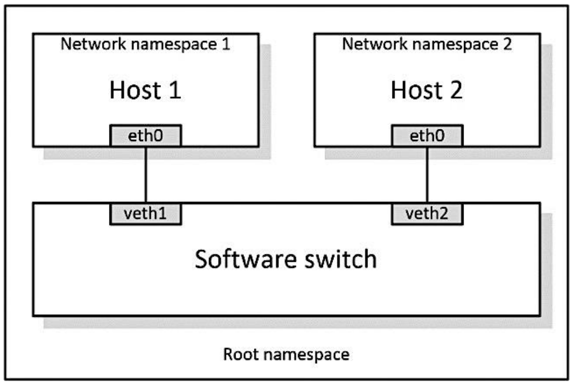


Figure 2. Network namespaces and virtual Ethernet links.

Each container is an independent network namespace, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and Address Resolution Protocol (ARP) tables.

Mininet provides network emulation opposed to simulation, allowing all network software at any layer to be simply run *as is*; i.e. nodes run the native network software of the physical machine. On the other hand, in a simulated environment applications and protocol implementations need to be ported to run within the simulator before they can be used.

2 Invoke Mininet using the CLI

In following subsections, you will start Mininet using the Linux CLI.

2.1 Invoke Mininet using the default topology

Step 1. Launch a Linux terminal by clicking on the Linux terminal icon in the task bar.

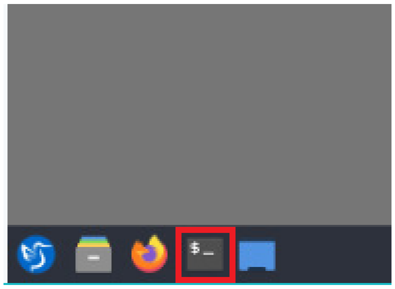
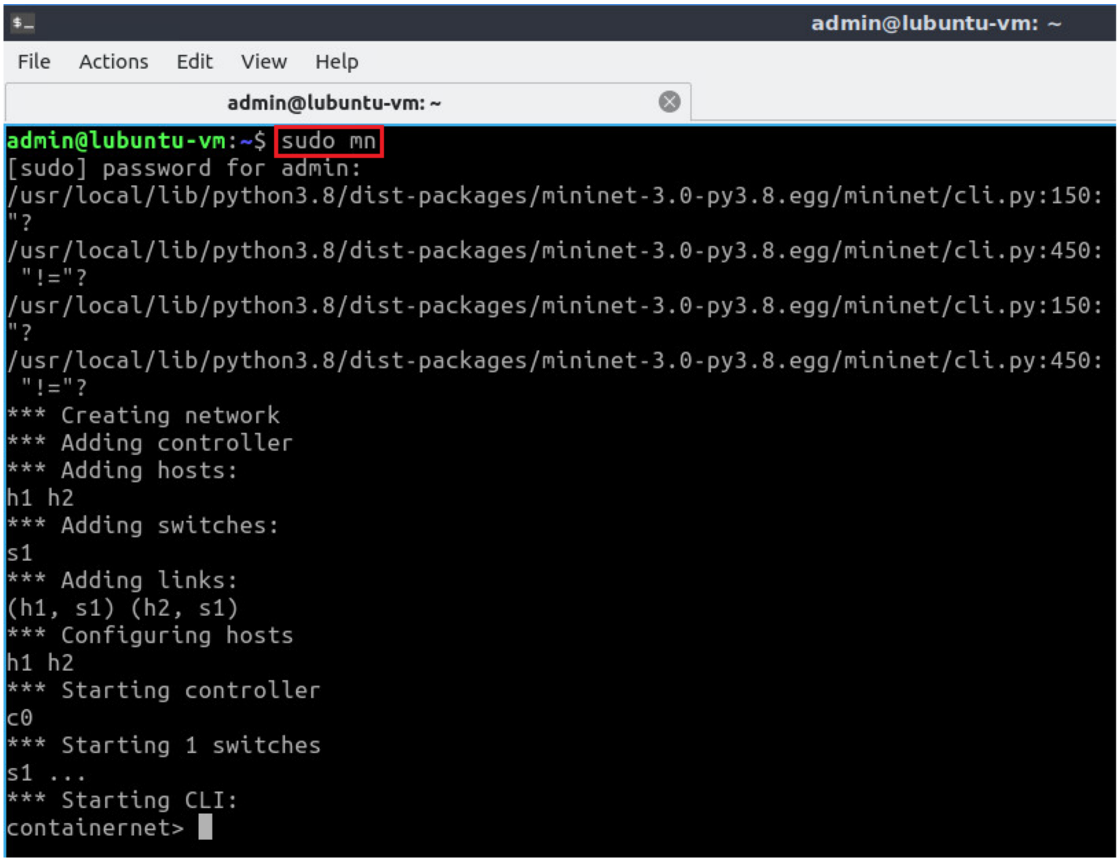


Figure 3. Linux terminal icon.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). A CLI is a program that takes commands from the keyboard and sends them to the operating system for execution.

Step 2. To start a minimal topology, enter the command shown below. When prompted for a password, type `password` and hit enter. Note that the password will not be visible as you type it.

```
sudo mn
```



```
admin@lubuntu-vm:~$ sudo mn
[sudo] password for admin:
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:150:
"?
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:450:
 "!="?
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:150:
"?
/usr/local/lib/python3.8/dist-packages/mininet-3.0-py3.8.egg/mininet/cli.py:450:
 "!="?
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet>
```

Figure 4. Starting Mininet using the CLI.

The above command starts Mininet with a minimal topology, which consists of a switch connected to two hosts as shown below.

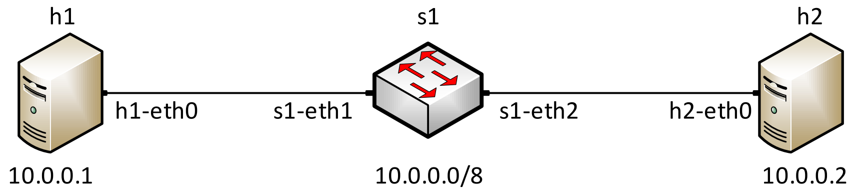


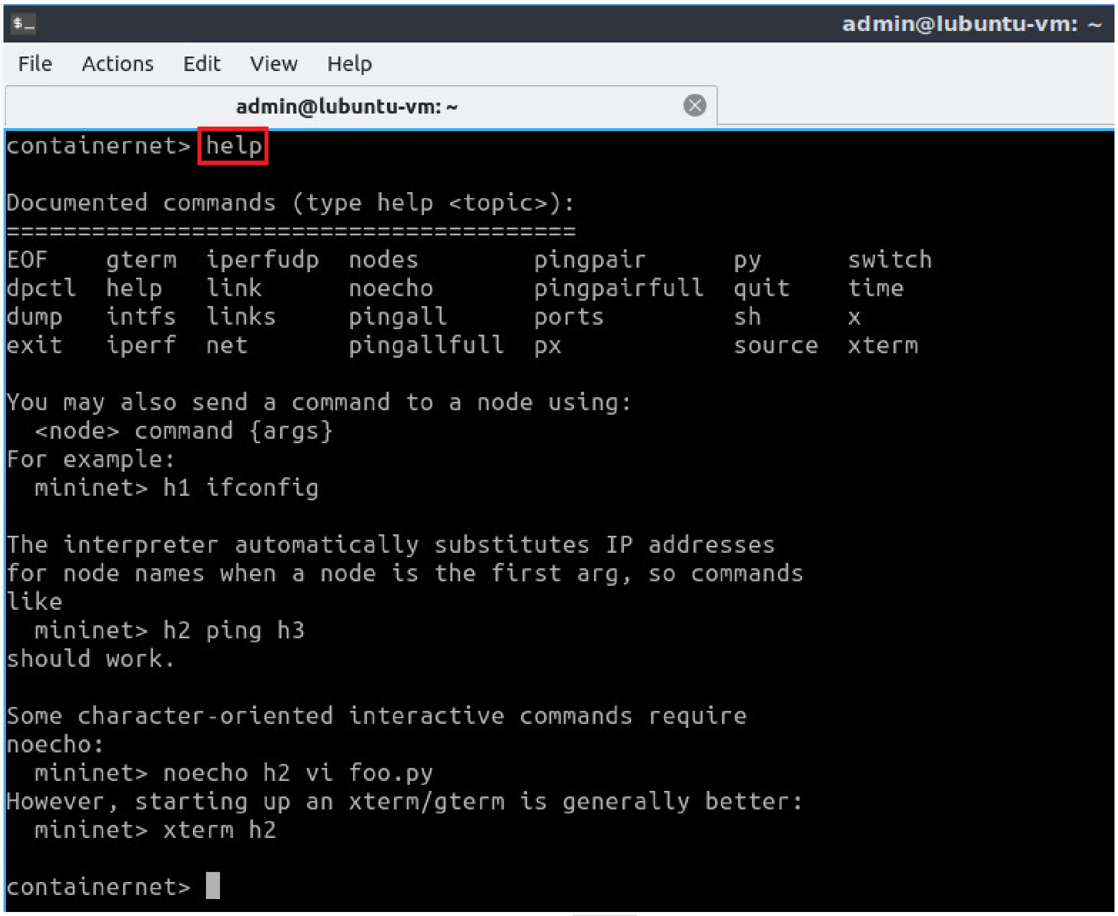
Figure 5. Mininet's default minimal topology.

When issuing the `sudo mn` command, Mininet initializes the topology and launches its command line interface which looks like this:

```
containernet>
```

Step 3. To display the list of Mininet CLI commands and examples on their usage, type the following command:

```
help
```

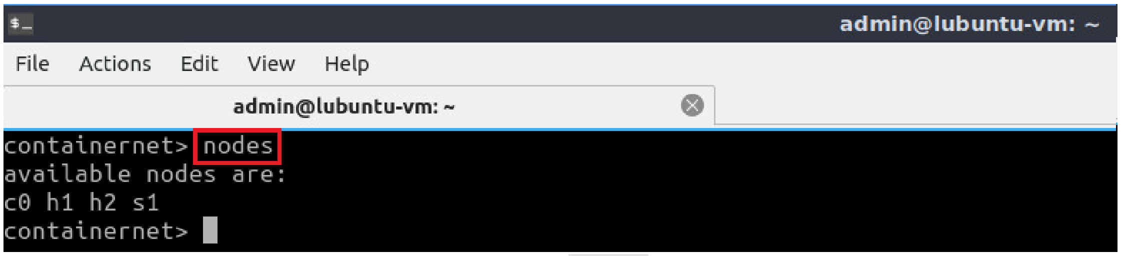


```
containernet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperfudp  nodes         pingpair      py      switch
dpctl  help   link      noecho        pingpairfull  quit    time
dump   intfs  links     pingall       ports         sh      x
exit   iperf  net       pingallfull   px            source  xterm

You may also send a command to a node using:
  <node> command {args}
For example:
  mininet> h1 ifconfig

The interpreter automatically substitutes IP addresses
for node names when a node is the first arg, so commands
like
  mininet> h2 ping h3
should work.

Some character-oriented interactive commands require
noecho:
  mininet> noecho h2 vi foo.py
However, starting up an xterm/gterm is generally better:
  mininet> xterm h2

containernet>
```

Figure 6. Mininet's `help` command.

Step 4. To display the available nodes, type the following command:

```
nodes
```



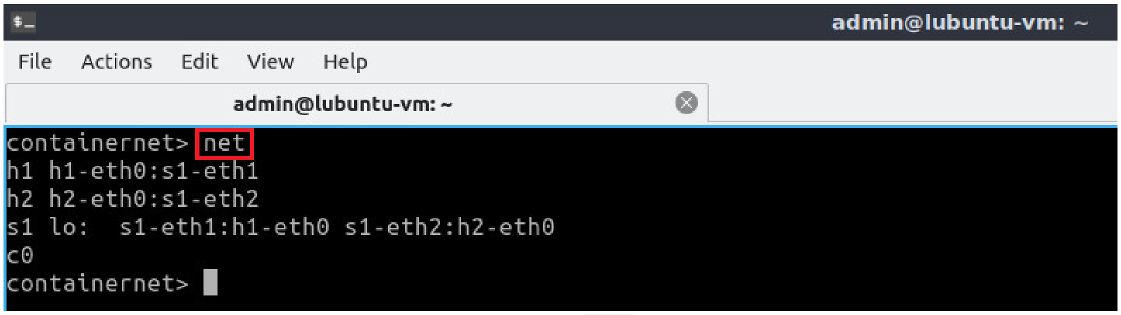
```
containernet> nodes
available nodes are:
c0 h1 h2 s1
containernet>
```

Figure 7. Mininet's `nodes` command.

The output of the `nodes` command shows that there is a controller (c0), two hosts (host h1 and host h2), and a switch (s1).

Step 5. It is useful sometimes to display the links between the devices in Mininet to understand the topology. Issue the command shown below to see the available links.

```
net
```



```
containernet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
containernet>
```

Figure 8. Mininet's `net` command.

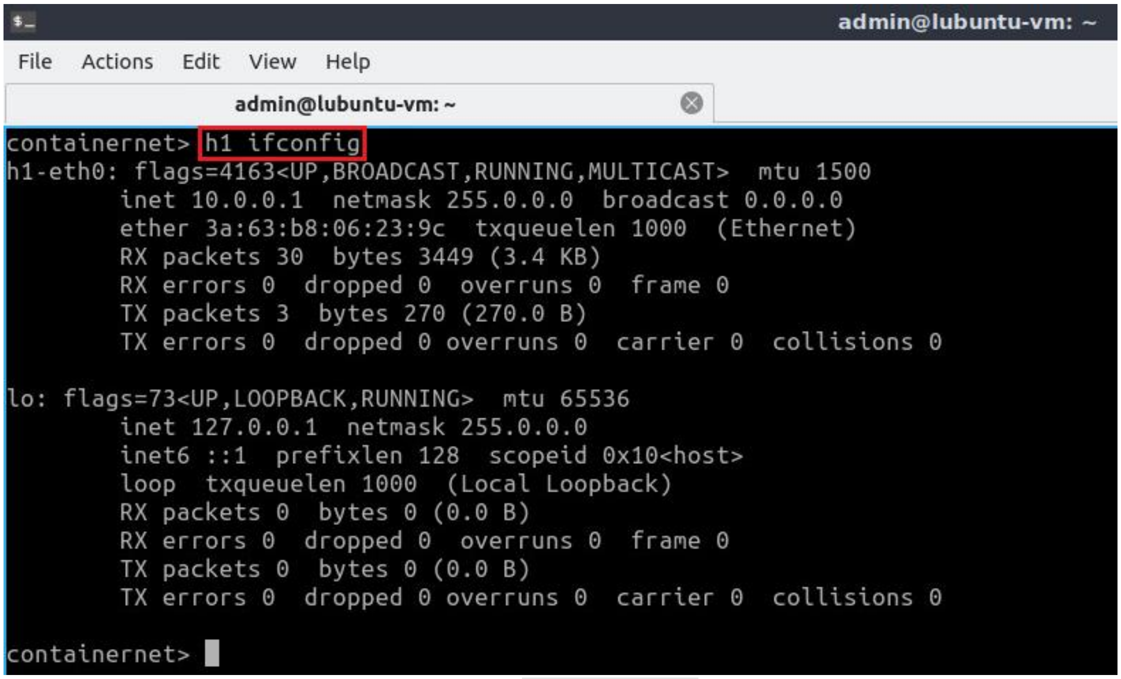
The output of the `net` command shows that:

1. Host h1 is connected using its network interface *h1-eth0* to the switch on interface *s1-eth1*.
2. Host h2 is connected using its network interface *h2-eth0* to the switch on interface *s1-eth2*.
3. Switch s1:
   a. Has a loopback interface *lo*.
   b. Connects to *h1-eth0* through interface *s1-eth1*.
   c. Connects to *h2-eth0* through interface *s1-eth2*.
4. Controller c0 does not have any connection.

Mininet allows you to execute commands on a specific device. To issue a command for a specific node, you must specify the device first, followed by the command.

Step 6. To proceed, issue the command:

```
h1 ifconfig
```



```
containernet> h1 ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 3a:63:b8:06:23:9c  txqueuelen 1000  (Ethernet)
        RX packets 30  bytes 3449 (3.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

containernet>
```

Figure 9. Output of `h1 ifconfig` command.

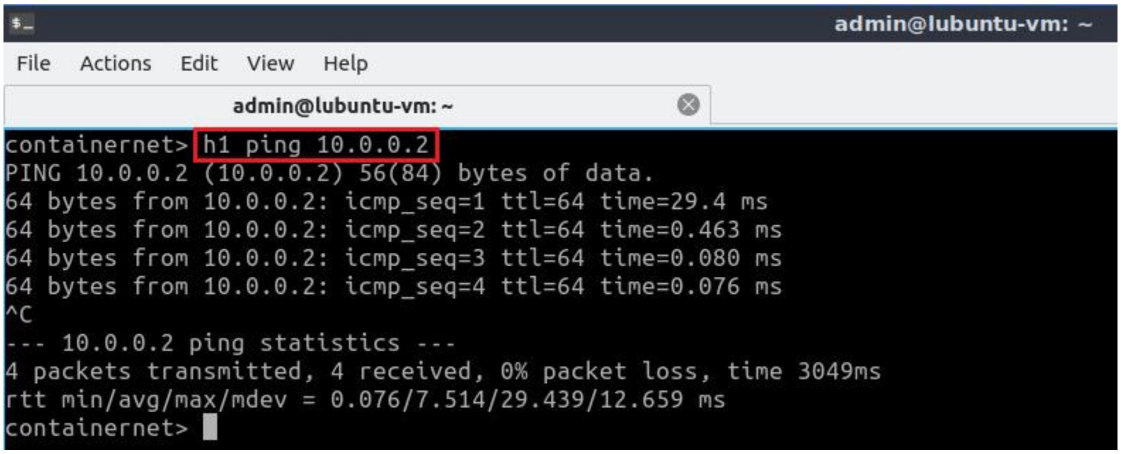
This command `h1 ifconfig` executes the `ifconfig` Linux command on host h1. The command shows host h1's interfaces. The display indicates that host h1 has an interface *h1-eth0* configured with IP address 10.0.0.1, and another interface lo configured with IP address 127.0.0.1 (loopback interface).

2.2 Test connectivity

Mininet's default topology assigns the IP addresses 10.0.0.1/8 and 10.0.0.2/8 to host h1 and host h2 respectively. To test connectivity between them, you can use the command `ping`. The `ping` command operates by sending Internet Control Message Protocol (ICMP) Echo Request messages to the remote computer and waiting for a response or reply. Information available includes how many responses are returned and how long it takes for them to return.

Step 1. On the CLI, type the command shown below. The command `h1 ping 10.0.0.2` tests the connectivity between host h1 and host h2. To stop the test, press `Ctrl+c`. The figure below shows a successful connectivity test. Host h1 (10.0.0.1) sent four packets to host h2 (10.0.0.2) and successfully received the expected responses.

```
h1 ping 10.0.0.2
```

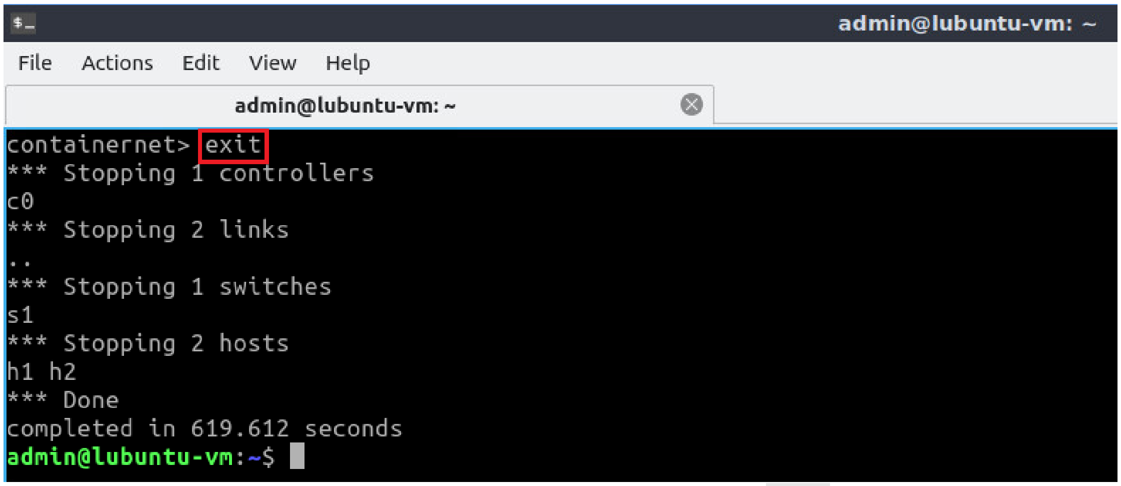


```
containernet> h1 ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=29.4 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.463 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.076 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.076/7.514/29.439/12.659 ms
containernet> 
```

Figure 10. Connectivity test between host h1 and host h2.

Step 2. Stop the emulation by typing the following command:

```
exit
```



```
containernet> exit
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 619.612 seconds
admin@lubuntu-vm:~$
```

Figure 11. Stopping the emulation using `exit`.

If Mininet were to crash for any reason, the `sudo mn – c` command can be utilized to clean a previous instance. However, the `sudo mn -c` command is often used within the Linux terminal and not the Mininet CLI.

Step 3. After stopping the emulation, close the Linux terminal by clicking the ☒ in the upper-right corner.

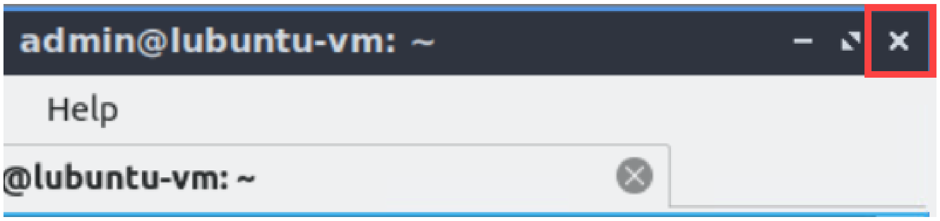


Figure 12. Closing the Linux CLI.

3 Build and emulate a network in Mininet using the GUI

In this section, you will use the application MiniEdit to deploy the topology illustrated below. MiniEdit is a simple GUI network editor for Mininet.

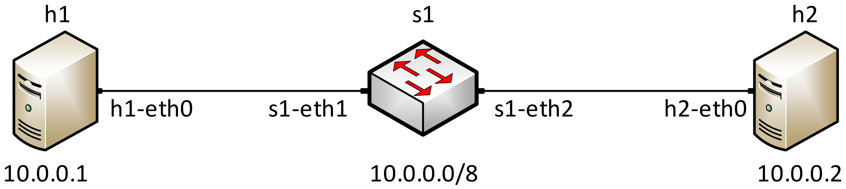


Figure 13. Lab topology.

3.1 Build the network topology

Step 1. A shortcut to MiniEdit is located on the machine's Desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`. MiniEdit will start, as illustrated below.

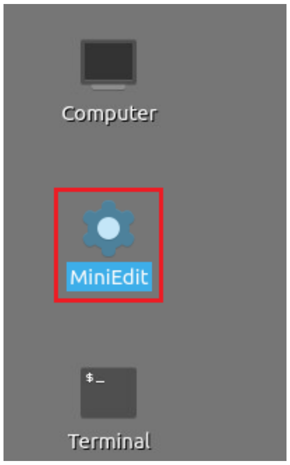


Figure 14. MiniEdit Desktop shortcut.

MiniEdit will start, as illustrated below.

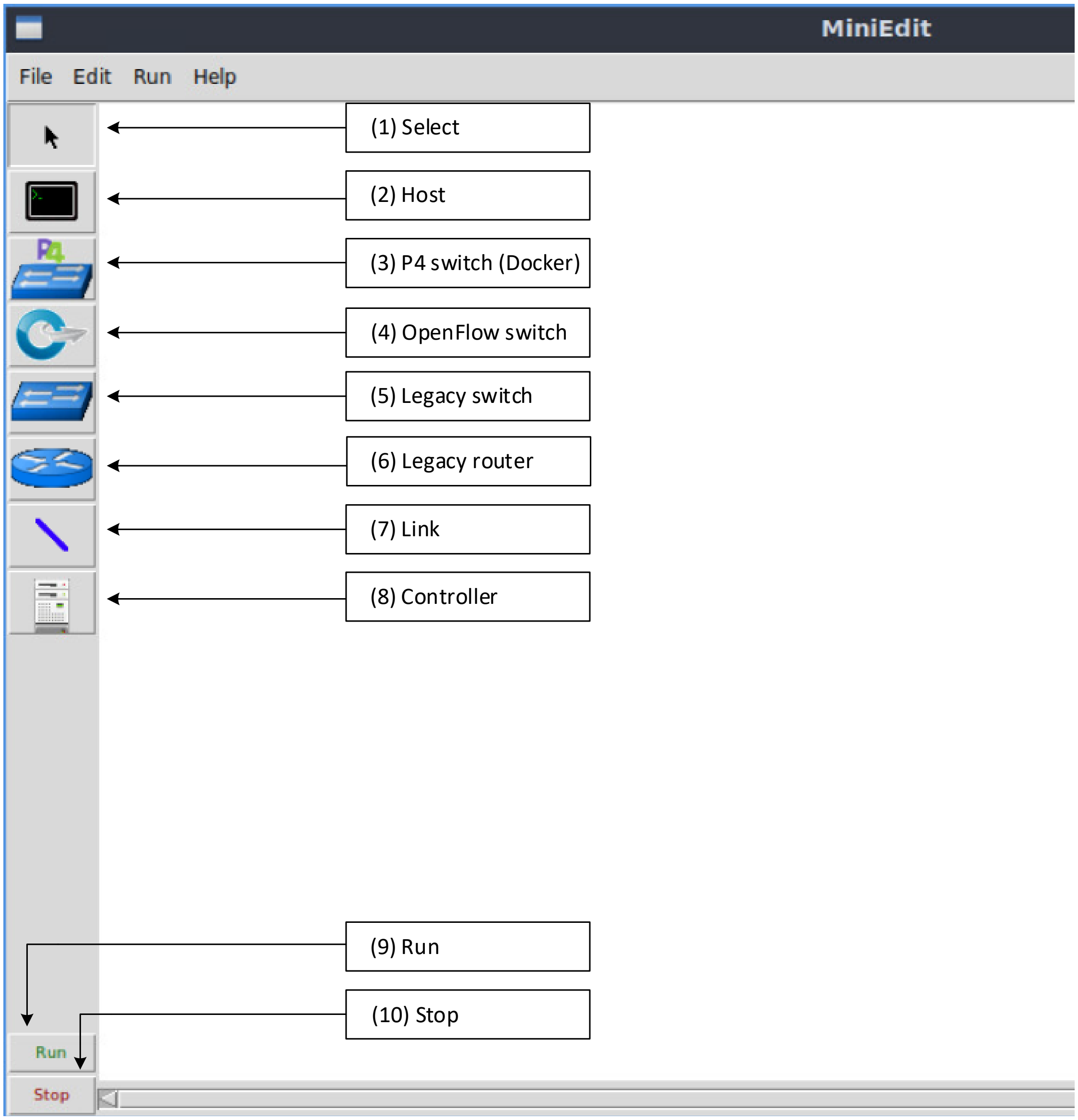


Figure 15. MiniEdit Graphical User Interface (GUI).

The main buttons are:

1. Select: allows selection/movement of the devices. Pressing *Del* on the keyboard after selecting the device removes it from the topology.
2. Host: allows addition of a new host to the topology. After clicking this button, click anywhere in the blank canvas to insert a new host.
3. P4 switch (Docker): allows the addition of P4 switch. After clicking this button, click anywhere in the blank canvas to insert the P4 switch.
4. OpenFlow switch: allows the addition of a new OpenFlow-enabled switch. After clicking this button, click anywhere in the blank canvas to insert the switch.
5. Legacy switch: allows the addition of a new Ethernet switch to the topology. After clicking this button, click anywhere in the blank canvas to insert the switch.
6. Legacy router: allows the addition of a new legacy router to the topology. After clicking this button, click anywhere in the blank canvas to insert the router.
7. Link: connects devices in the topology (mainly switches and hosts). After clicking this button, click on a device and drag to the second device to which the link is to be established.
8. Controller: allows the addition of a new OpenFlow controller.
9. Run: starts the emulation. After designing and configuring the topology, click the run button.
10. Stop: stops the emulation.

Step 2. To build the topology illustrated in Figure 13, two hosts and one switch must be deployed. Deploy these devices in MiniEdit, as shown below.

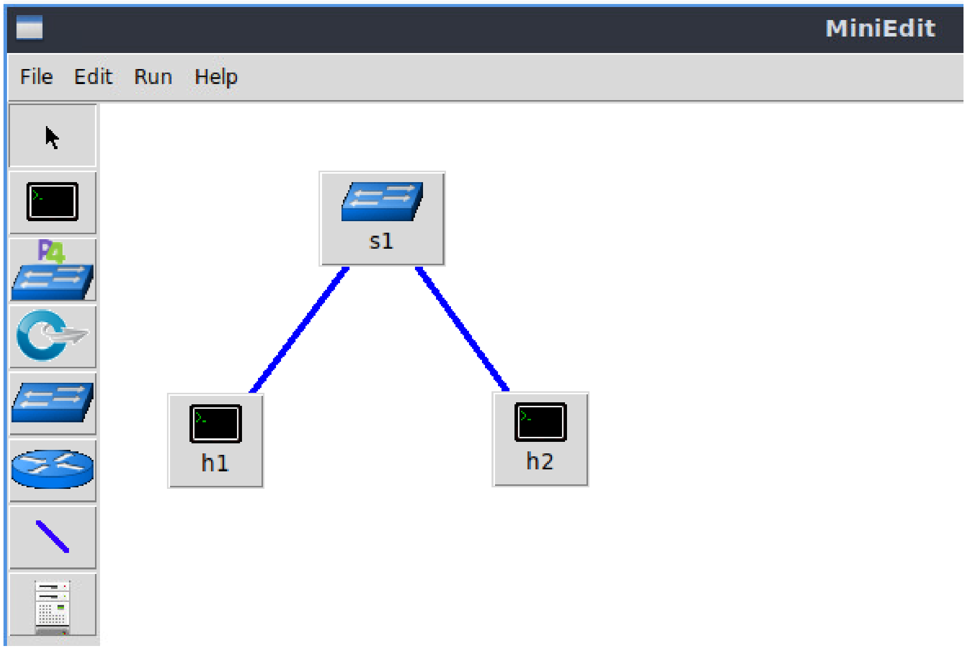
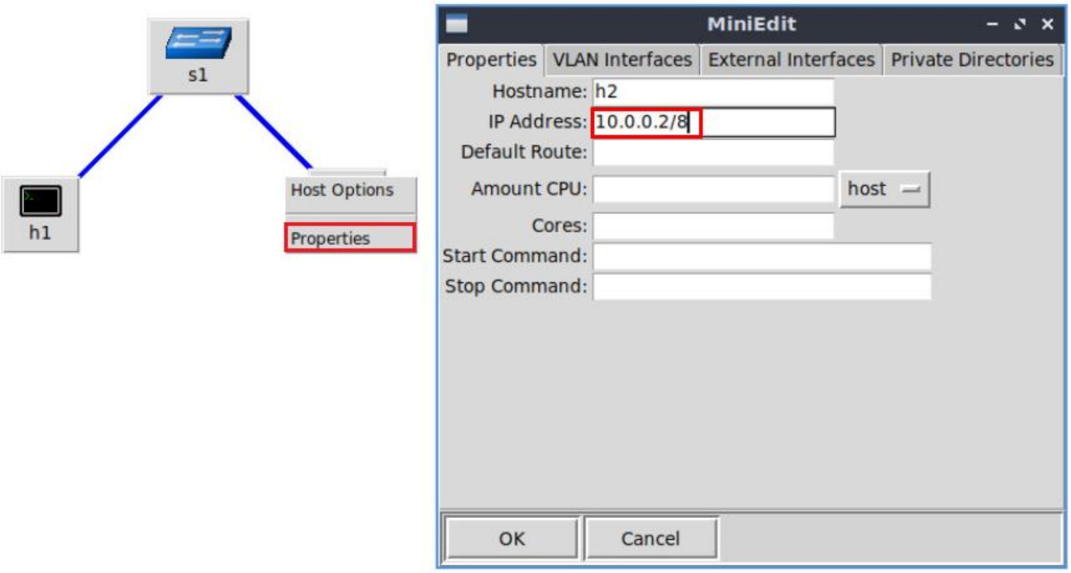


Figure 16. MiniEdit's topology.

Use the buttons described in the previous step to add and connect devices. The configuration of IP addresses is described in Step 3.

Step 3. Configure the IP addresses of host h1 and host h2. Host h1's IP address is 10.0.0.1/8 and host h2's IP address is 10.0.0.2/8. A host can be configured by holding the right click and selecting properties on the device. For example, host h2 is assigned the IP address 10.0.0.2/8 in the figure below. Click *OK* for the settings to be applied.



Hostname: h2
IP Address: 10.0.0.2/8
Default Route:
Amount CPU:
Cores:
Start Command:
Stop Command:

Figure 17. Configuration of a host's properties.

3.2 Test connectivity

Before testing the connection between host h1 and host h2, the emulation must be started.

Step 1. Click the *Run* button to start the emulation. The emulation will start and the buttons of the MiniEdit panel will gray out, indicating that they are currently disabled.

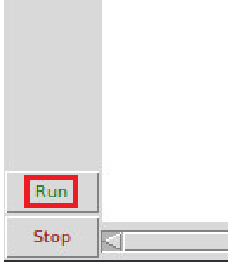


Figure 18. Starting the emulation.

Step 2. Open a terminal by right-clicking on host h1 and select *Terminal*. This opens a terminal on host h1 and allows the execution of commands on the host h1. Repeat the procedure on host h2.

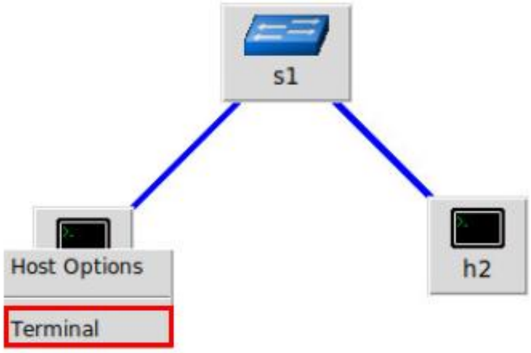
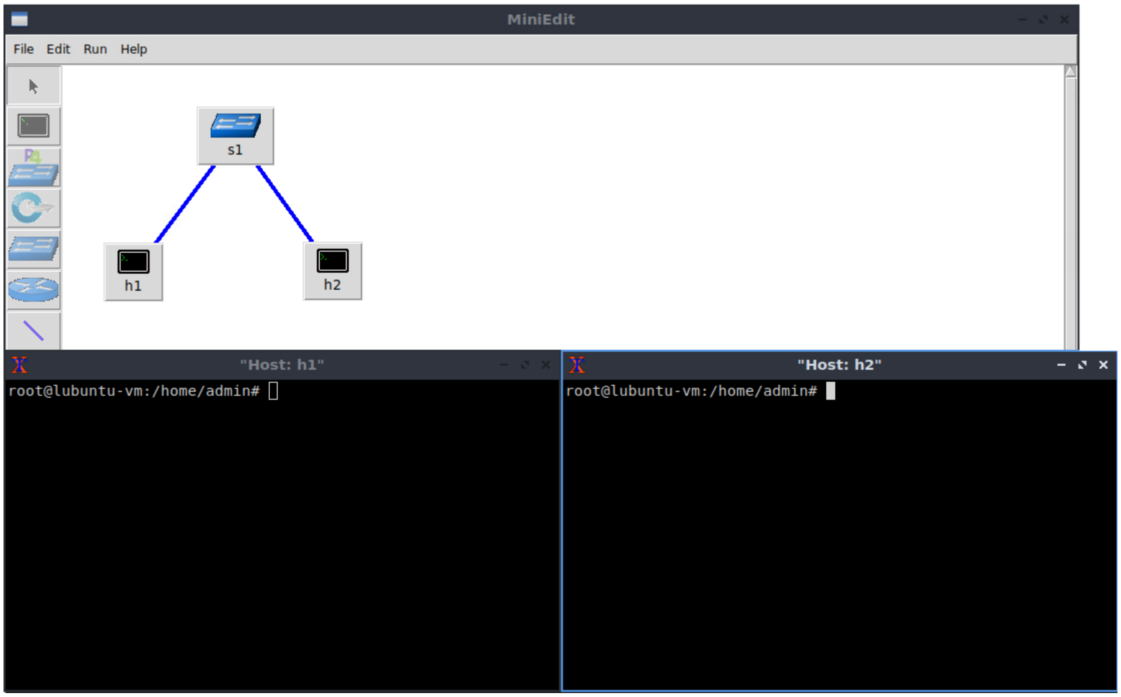


Figure 19. Opening a terminal on host h1.

The network and terminals at host h1 and host h2 will be available for testing.



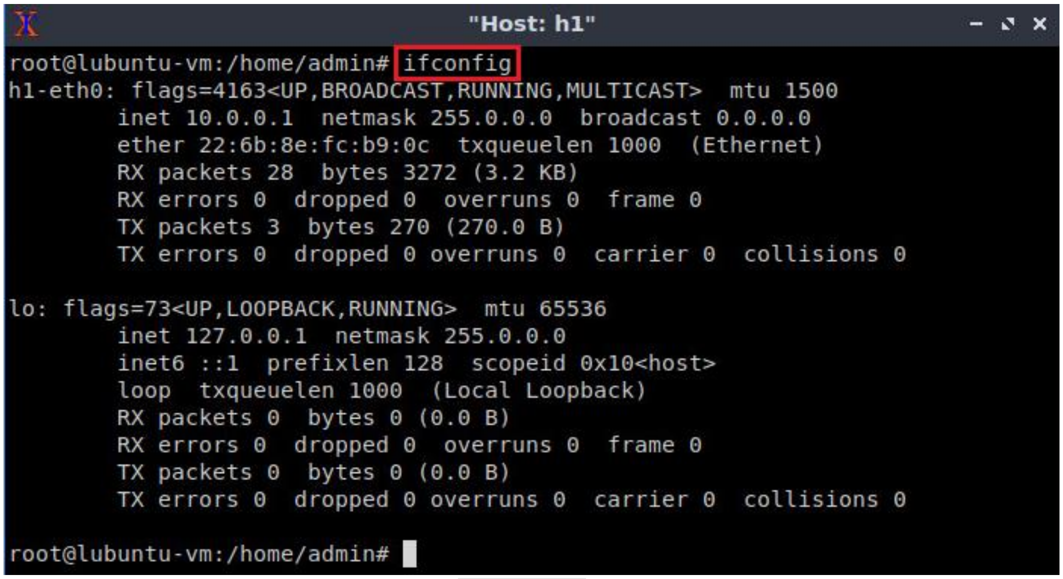
```
root@lubuntu-vm:/home/admin# 
```

```
root@lubuntu-vm:/home/admin# 
```

Figure 20. Terminals at host h1 and host h2.

Step 3. On host h1’s terminal, type the command shown below to display its assigned IP addresses. The interface *h1-eth0* at host h1 should be configured with the IP address 10.0.0.1 and subnet mask 255.0.0.0.

```
ifconfig
```



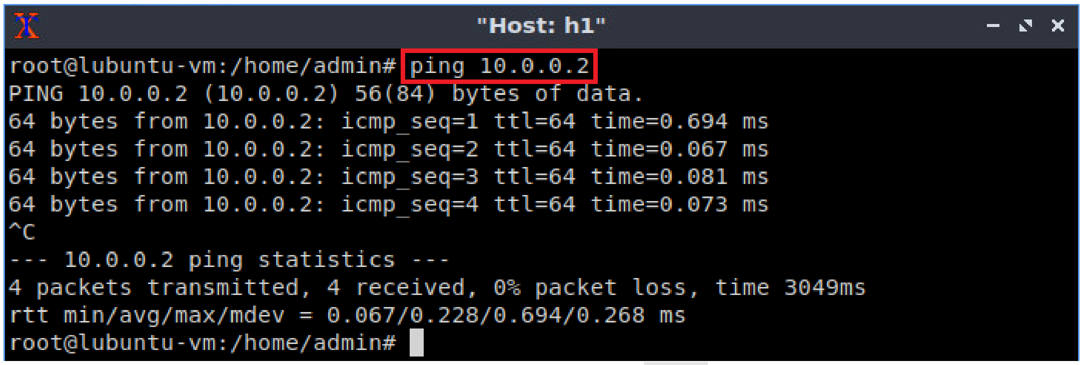
```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 22:6b:8e:fc:b9:0c  txqueuelen 1000  (Ethernet)
        RX packets 28  bytes 3272 (3.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin# 
```

Figure 21. Output of `ifconfig` command on host h1.

Repeat Step 3 on host h2. Its interface *h2-eth0* should be configured with IP address 10.0.0.2 and subnet mask 255.0.0.0.

Step 4. On host h1's terminal, type the command shown below. This command tests the connectivity between host h1 and host h2. To stop the test, press `Ctrl+c`. The figure below shows a successful connectivity test. Host h1 (10.0.0.1) sent six packets to host h2 (10.0.0.2) and successfully received the expected responses.

```
ping 10.0.0.2
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.694 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.073 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.067/0.228/0.694/0.268 ms
root@lubuntu-vm:/home/admin# 
```

Figure 22. Connectivity test using `ping` command.

Step 5. Stop the emulation by clicking on the *Stop* button.

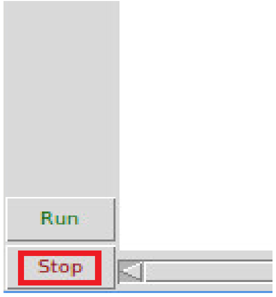
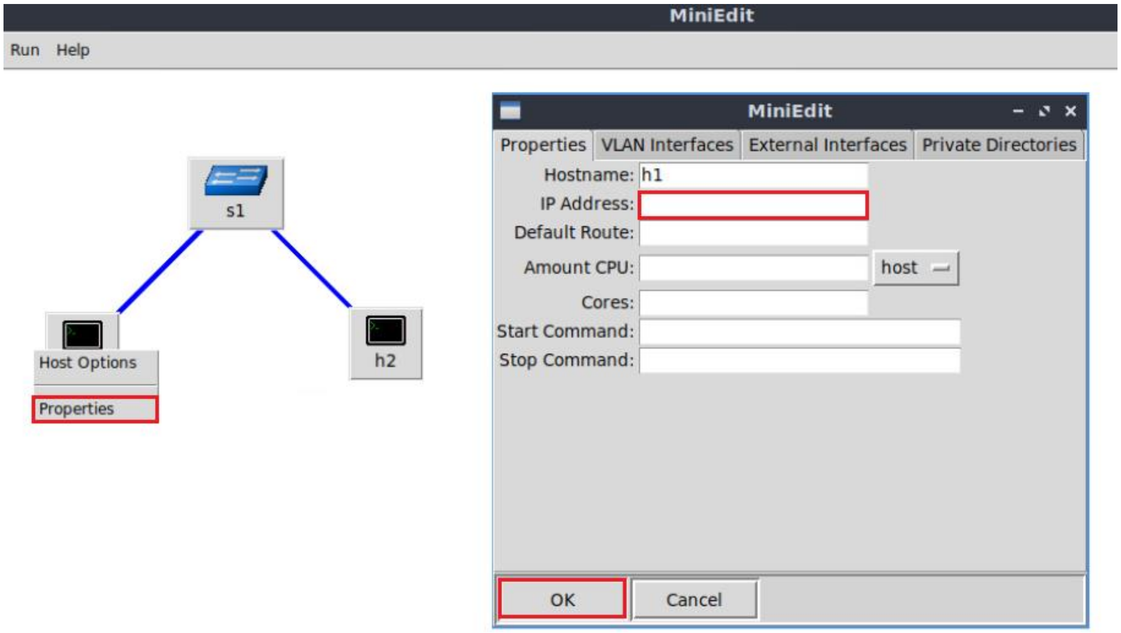


Figure 23. Stopping the emulation.

3.3 Automatic assignment of IP addresses

In the previous section, you manually assigned IP addresses to host h1 and host h2. An alternative is to rely on Mininet for an automatic assignment of IP addresses (by default, Mininet uses automatic assignment), which is described in this section.

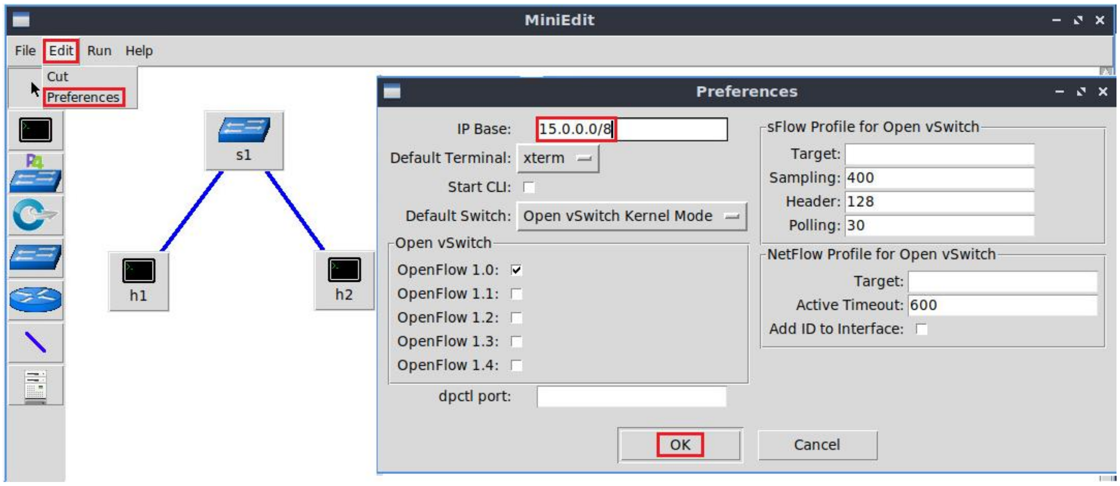
Step 1. Remove the manually assigned IP address from host h1. Right-click on host h1 and select *Properties*. Delete the IP address, leaving it unassigned, and press the *OK* button as shown below. Repeat the procedure on host h2.



Hostname: h1
IP Address:
Default Route:
Amount CPU:
Cores:
Start Command:
Stop Command:

Figure 24. Host h1 properties.

Step 2. In the MiniEdit application, navigate to *Edit > Preferences*. The default IP base is 10.0.0.0/8. Modify this value to 15.0.0.0/8, and then press the *OK* button.



IP Base: 15.0.0.0/8
Default Terminal: xterm
Start CLI: □
Default Switch: Open vSwitch Kernel Mode

Open vSwitch
OpenFlow 1.0: ☑
OpenFlow 1.1: □
OpenFlow 1.2: □
OpenFlow 1.3: □
OpenFlow 1.4: □

dpctl port:

sFlow Profile for Open vSwitch
Target:
Sampling: 400
Header: 128
Polling: 30

NetFlow Profile for Open vSwitch
Target:
Active Timeout: 600
Add ID to Interface: □

Figure 25. Modification of the IP Base (network address and prefix length).

Step 3. Run the emulation again by clicking on the *Run* button. The emulation will start and the buttons of the MiniEdit panel will be disabled.

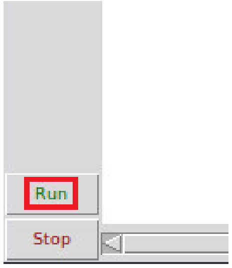


Figure 26. Starting the emulation.

Step 4. Open a terminal by right-clicking on host h1 and select *Terminal*.

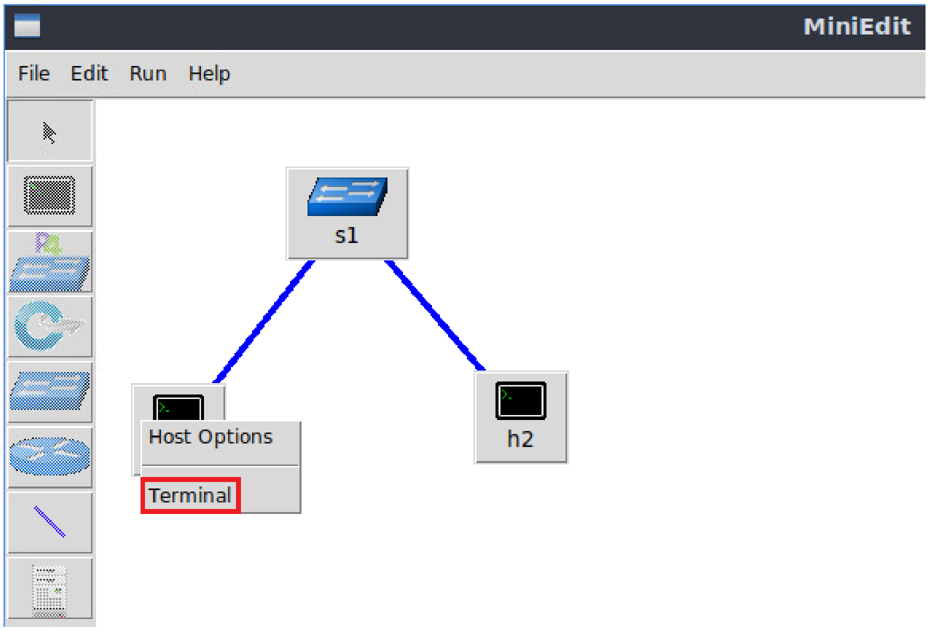
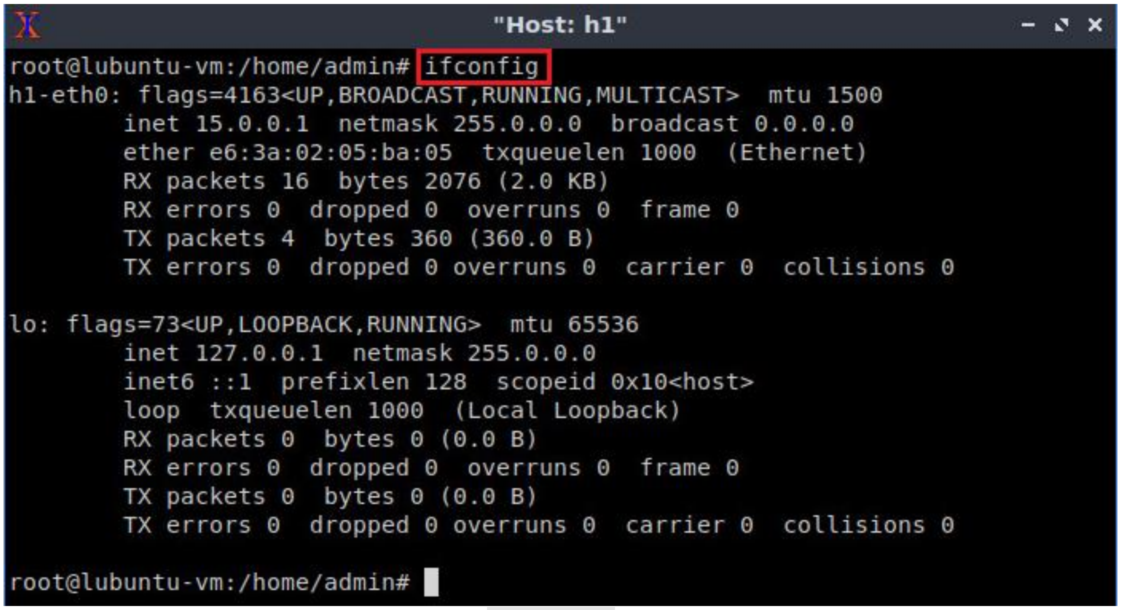


Figure 27. Opening a terminal on host h1.

Step 5. Type the command shown below to display the IP addresses assigned to host h1. The interface *h1-eth0* at host h1 now has the IP address 15.0.0.1 and subnet mask 255.0.0.0.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 15.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether e6:3a:02:05:ba:05  txqueuelen 1000  (Ethernet)
        RX packets 16  bytes 2076 (2.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 360 (360.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin# 
```

Figure 28. Output of `ifconfig` command on host h1.

You can also verify the IP address assigned to host h2 by repeating Steps 4 and 5 on host h2's terminal. The corresponding interface *h2-eth0* at host h2 has now the IP address 15.0.0.2 and subnet mask 255.0.0.0.

Step 6. Stop the emulation by clicking on *Stop* button.

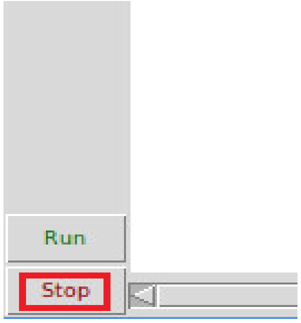


Figure 29. Stopping the emulation.

3.4 Save and load a Mininet topology

In this section you will save and load a Mininet topology. It is often useful to save the network topology, particularly when its complexity increases. MiniEdit enables you to save the topology to a file.

Step 1. In the MiniEdit application, save the current topology by clicking *File*. Provide a name for the topology and notice *myTopology* as the topology name. Ensure you are in the *lab1* folder and click *Save*.

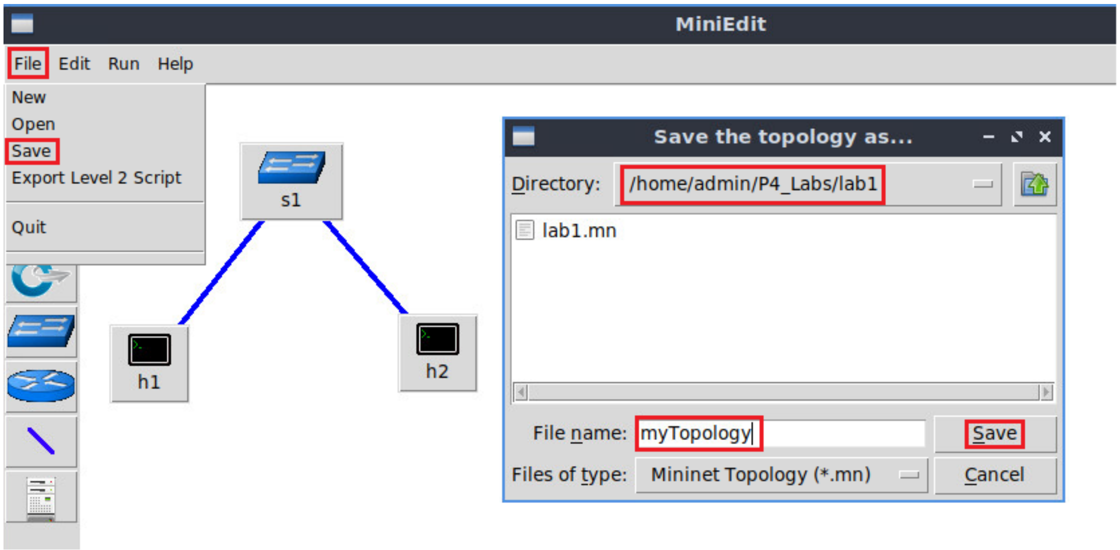


Figure 30. Saving the topology.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab1* folder and search for the topology file called *lab1.mn* and click on Open. A new topology will be loaded to MiniEdit.

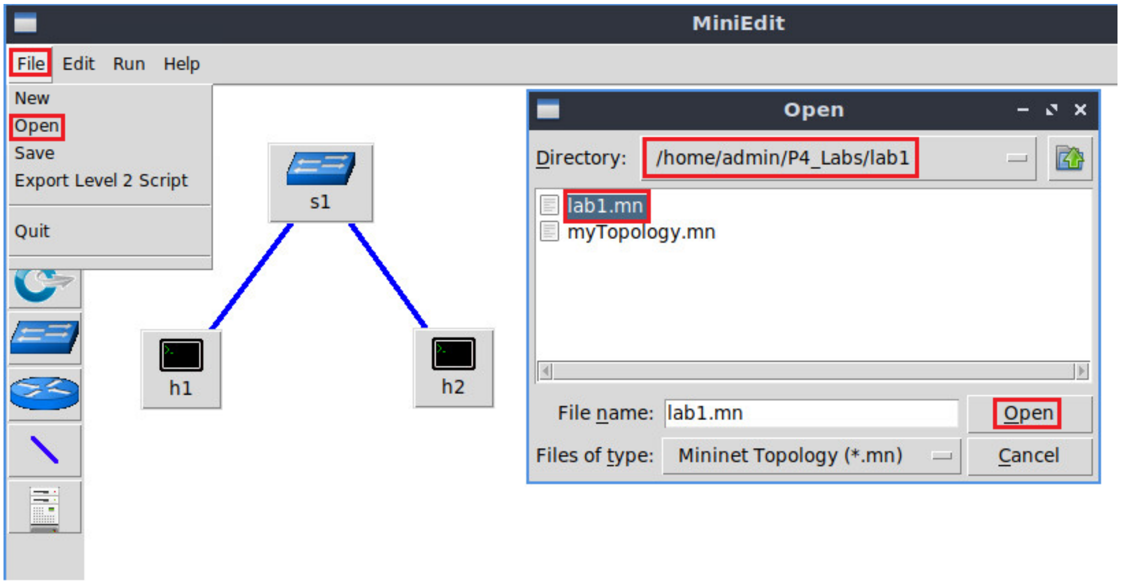


Figure 31. Opening a topology.

This concludes lab 1. Stop the emulation and then exit out of MiniEdit and the Linux terminal.

References

1. Mininet walkthrough. [Online]. Available: http://Mininet.org.
2. Mckeown N., Anderson T., Balakrishnan H., Parulkar G., Peterson L., Rexford J., Shenker S., Turner J., "*OpenFlow*," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, p. 69, 2008.

3. Esch J., “*Prolog to, software-defined networking: a comprehensive survey*,” Proceedings of the IEEE, vol. 103, no. 1, pp. 10–13, 2015.
4. Dordal P., “*An Introduction to computer networks*,”. [Online]. Available: https://intronetworks.cs.luc.edu/.
5. Lantz B., Gee G. “*MiniEdit: a simple network editor for Mininet.*” 2013. [Online]. Available: https://github.com/Mininet/Mininet/blob/master/examples.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 2: Introduction to P4 and BMv2

Document Version: 01-25-2022

Contents

Overview

This lab introduces programmable data plane switches and their role in the Software-defined Networking (SDN) paradigm. The lab introduces the Programming Protocol-independent Packet Processors (P4), the de facto programming language used to describe the behavior of the data planes of programmable switches. The focus of this lab is to provide a high-level overview of the general lifecycle of programming, compiling, and running a P4 program on a software switch.

Objectives

By the end of this lab, students should be able to:

1. Define the need for SDN and data plane programmability.
2. Understand the structure of a P4 program.
3. Compile a simple P4 program and deploy it to a software switch.
4. Start the switch daemon and allocate virtual interfaces to the switch.
5. Perform a connectivity test to verify the correctness of the program.

Lab settings

Table 1 contains the credentials of the virtual machine used for this lab.

Table 1**.** Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Loading the P4 program.
4. Section 4: Configuring switch s1.

1 Introduction

Since the emergence of the world wide web and 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 in the flexibility of protocol design caused by standardized requirements, which cannot be easily removed to enable protocol changes, has perpetuated the status quo. This protocol ossification[1, 2] has been characterized by a slow innovation pace at the hand of few network vendors. As an example, after being initially conceived by Cisco and VMware[3], the Application Specific Integrated Circuit (ASIC) implementation of the Virtual Extensible LAN (VXLAN)[4], a simple frame encapsulation protocol, took several years, a process that could have been reduced to weeks by software implementations. The design cycle of switch ASICs has been characterized by a lengthy, closed, and proprietary process that usually takes years. Such process contrasts with the agility of the software industry.

The programmable forwarding can be viewed as a natural 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[5], which stands for Programming Protocol-independent Packet Processors. Essentially, P4 programmable switches have removed the entry barrier to network design, previously reserved to network vendors.

1.1 Workflow of a P4 program

Programming a P4 switch, whether a hardware or a software target, requires a software development environment that includes a compiler. Consider Figure 1. The compiler maps the target-independent P4 source code (P4 program) to the specific platform. The compiler, the architecture model, and the target device are vendor specific and are provided by the vendor. The P4 source code on the other hand is supplied by the user.

The compiler generates two artifacts after compiling the P4 program. First, it generates a data plane configuration (Data plane runtime) that implements the forwarding logic specified in the P4 input program. This configuration includes the instructions and resource mappings for the target. Second, it generates runtime APIs that are used by the control plane / user to interact with the data plane. Examples include adding/removing entries from match-action tables and reading/writing the state of extern objects (e.g., counters, meters, registers). The APIs contain the information needed by the control plane to manipulate tables and objects in the data plane, such as the identifiers of the tables, fields used for matches, keys, action parameters, and others.

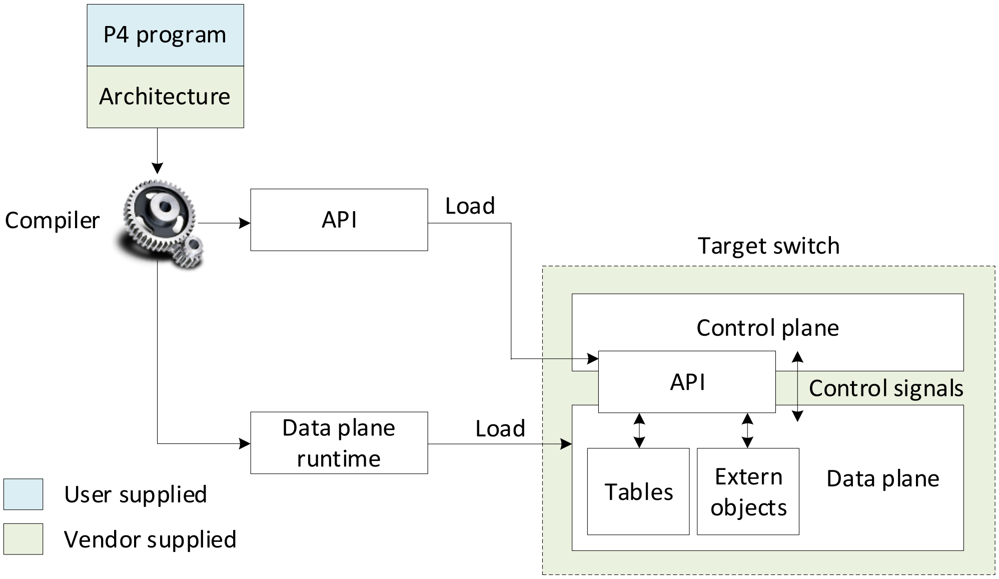


Figure 1. Generic workflow design. The compiler, the architecture model, and the target switch are provided by the vendor of the device. The P4 source code is customized by the user. The compiler generates a data plane runtime to be loaded into the target, and the APIs used by the control plane to communicate with the data plane at runtime.

1.2 Workflow used in this lab series

This section demonstrates the P4 workflow that will be used in this lab series. Consider Figure 2. We will use the Visual Studio Code (VS Code) as the editor to modify the *basic.p4* program. Then, we will use the *p4c* compiler with the V1Model architecture to compile the user supplied P4 program (*basic.p4*). The compiler will generate a JSON output (i.e., *basic.json*) which will be used as the data plane program by the switch daemon (i.e., simple_switch). Finally, we will use the `simple_switch_CLI` at runtime to populate and manipulate table entries in our P4 program. The target switch (vendor supplied) used in this lab series for testing and debugging P4 programs is the behavioral model version 2 (BMv2)[6].

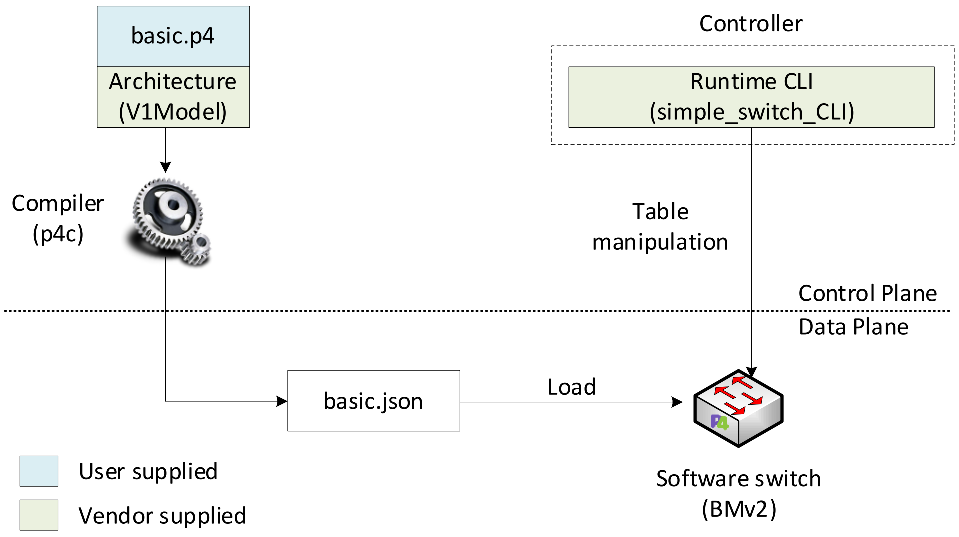


Figure 2. Workflow used in this lab series.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

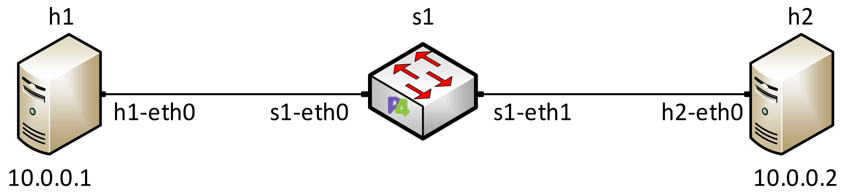


Figure 3. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

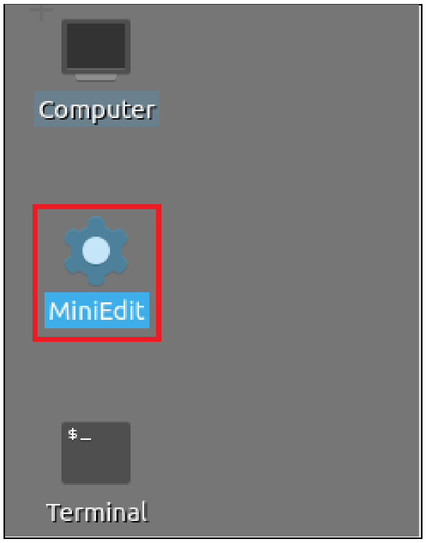


Figure 4. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab2*, select the file *lab2.mn,* and click on *Open*.

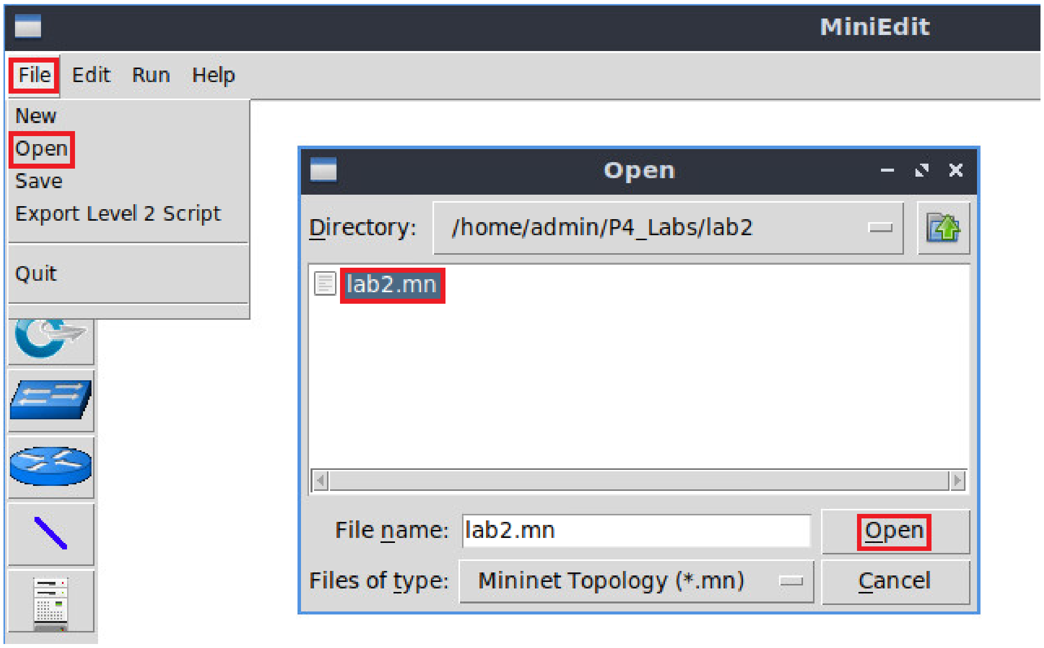


Figure 5. Opening a topology in MiniEdit.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

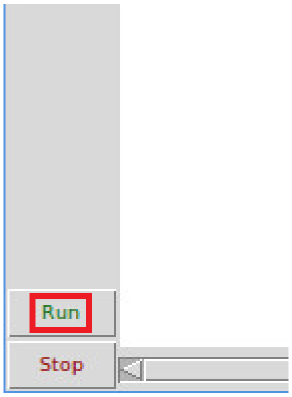


Figure 6. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

Step 1. Hold the right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

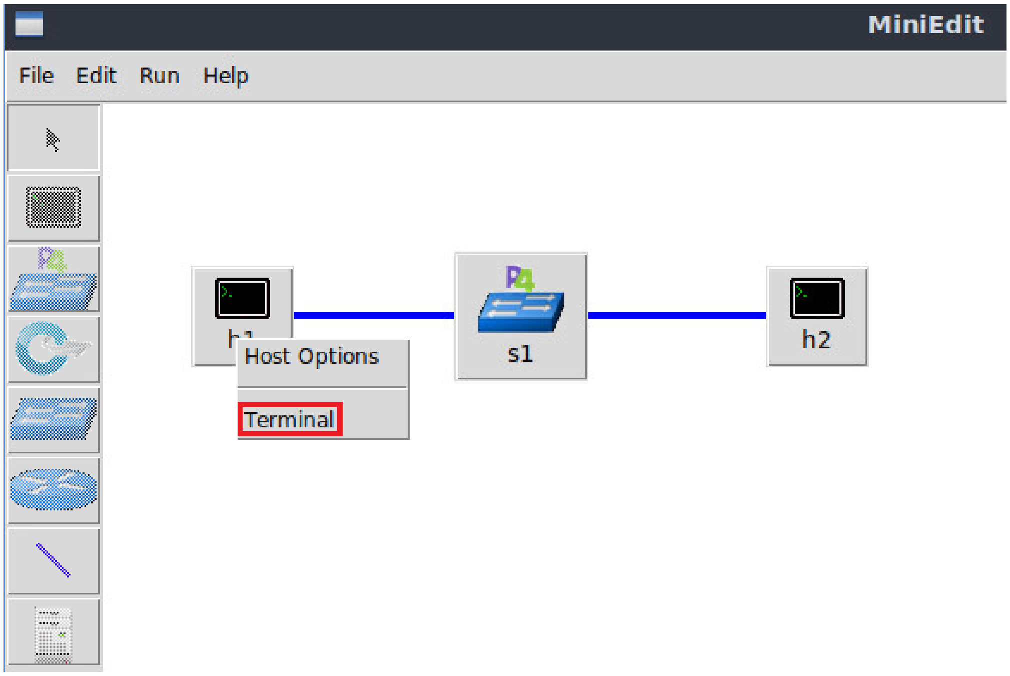
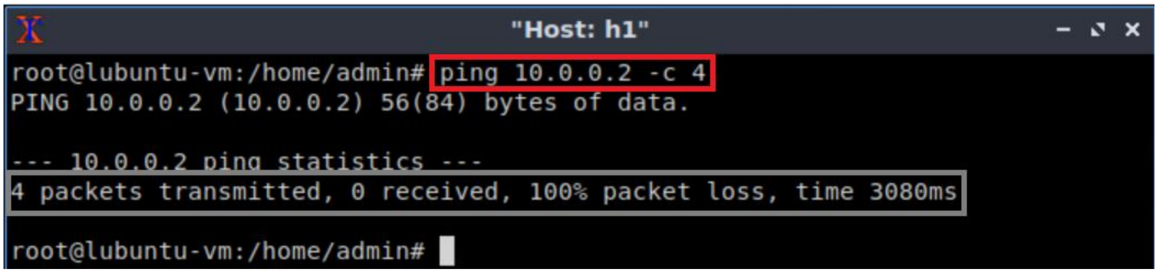


Figure 7. Opening a terminal on host h1.

Step 2. Test the connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3080ms

root@lubuntu-vm:/home/admin# 
```

Figure 8. Performing a connectivity test between host h1 and host h2.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded into the switch.

3 Loading the P4 program

This section shows the steps required to implement a P4 program. It describes the editor that will be used to modify the P4 program and the P4 compiler that will produce a data plane program for the software switch.

VS Code will be used as the editor to modify P4 programs. It highlights the syntax of P4 and provides an integrated terminal where the P4 compiler will be invoked. The P4 compiler that will be used is *p4c*, the reference compiler for the P4 programming language. *p4c* supports both P4$_{14}$ and P4$_{16}$, but in this lab series we will only focus on P4$_{16}$ since it is the newer version and is currently being supported by major programming ASIC manufacturers[7].

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the Linux terminal icon located on the desktop.

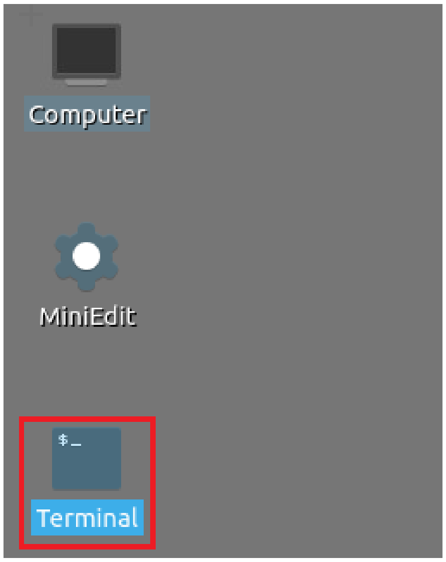
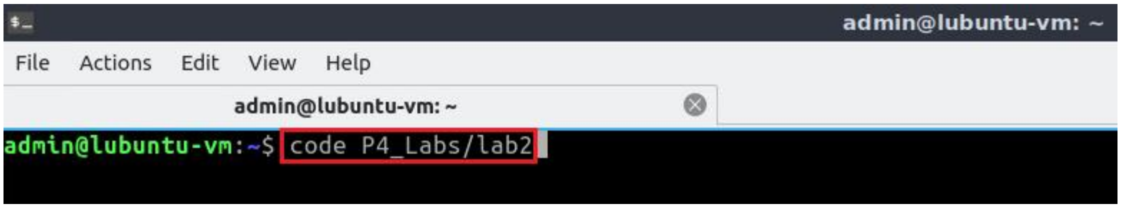


Figure 9. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the VS Code and opens the directory where the P4 program for this lab is located.

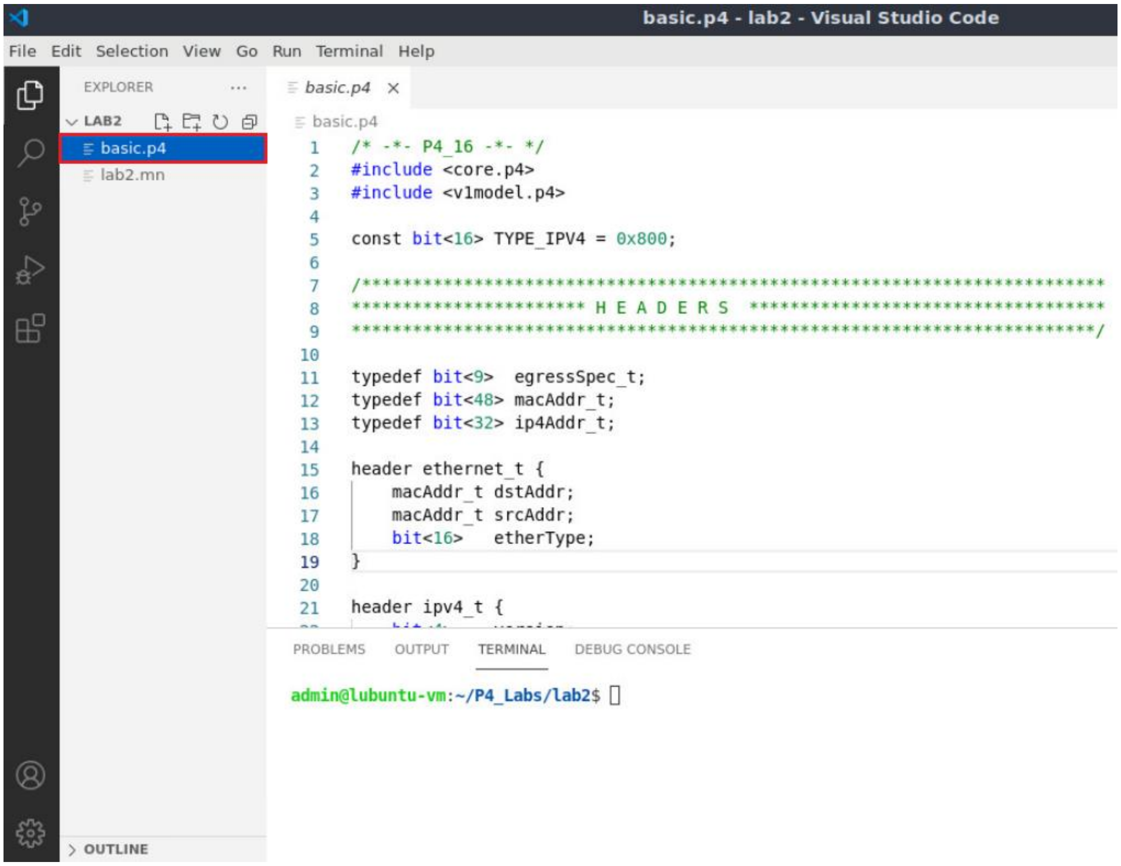
```
code P4_Labs/lab2
```



```
admin@lubuntu-vm:~$ code P4_Labs/lab2
```

Figure 10. Launching the editor and opening the lab2 directory.

Step 3. Once the previous command is executed, VS Code will start. Click on *basic.p4* in the file explorer panel on the left hand side to open the P4 program in the editor.

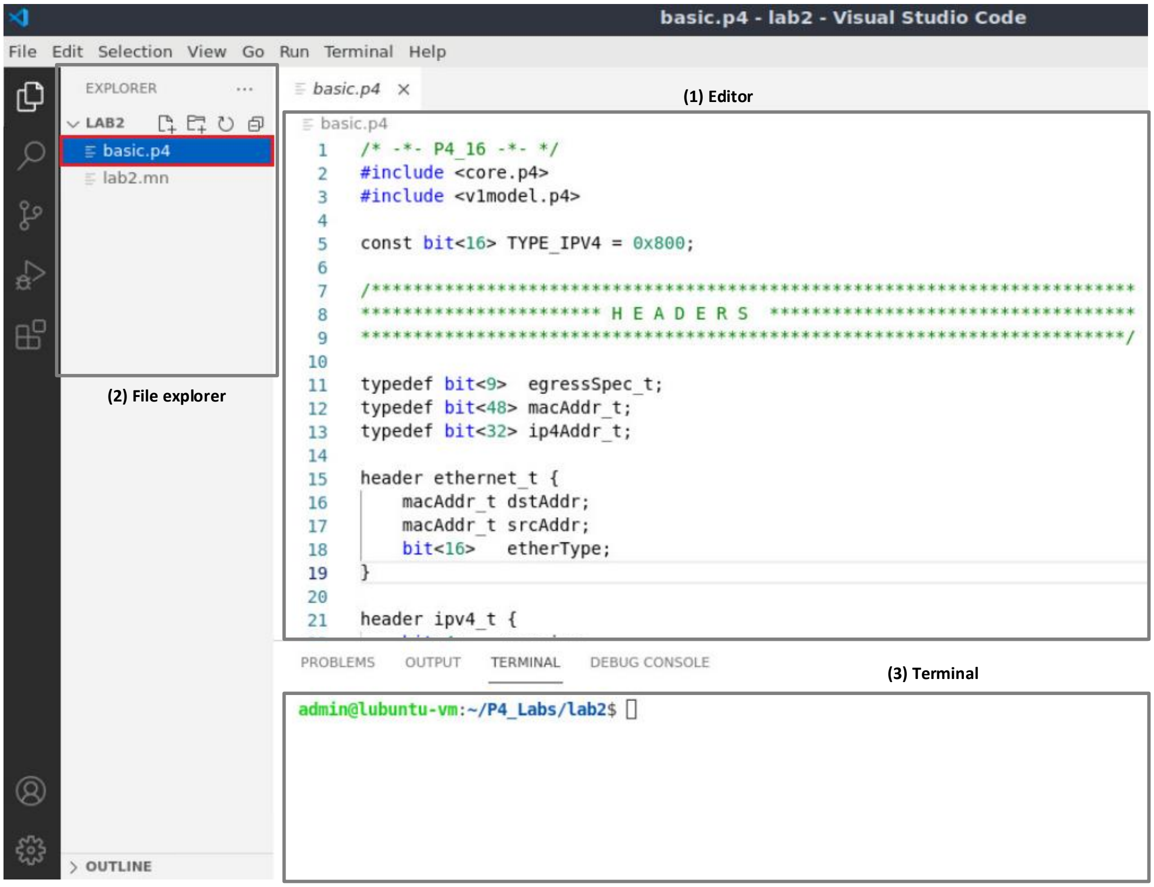

basic.p4 - lab2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB2
basic.p4
lab2.mn
OUTLINE
basic.p4
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** 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 ipv4_t {
```

```
admin@lubuntu-vm:~/P4_Labs/lab2$
```

Figure 11. Opening the programming environment in VS Code.

Step 4. Identify the components of VS Code highlighted in the grey boxes.


basic.p4 - lab2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB2
basic.p4
lab2.mn
OUTLINE
basic.p4
(1) Editor
(2) File explorer
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
(3) Terminal

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** 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 ipv4_t {
```

```
admin@lubuntu-vm:~/P4_Labs/lab2$
```

Figure 12. VS Code graphical interface components.

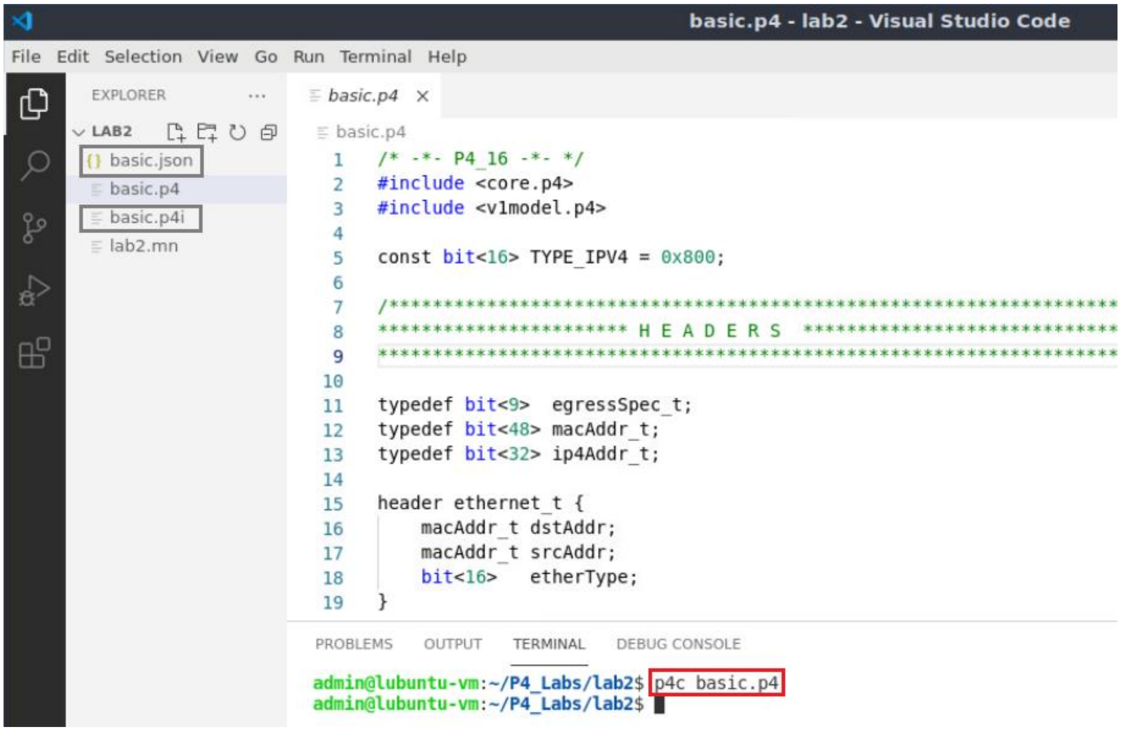
The VS Code interface consists of three main panels:

1. Editor: the editor panel will display the content of the file selected in the file explorer. In the figure above, the *basic.p4* program is shown in the Editor.
2. File explorer: this panel contains all the files in the current directory. You will see the *basic.p4* file which contains the P4 program that will be used in this lab, and the topology file for the current lab (i.e., *lab2.mn*).
3. Terminal: this is a regular Linux terminal integrated in the VS Code. This is where the compiler (*p4c*) is invoked to compile the P4 program and generate the output for the switch.

3.2 Compiling and loading the P4 program to switch s1

Step 1. In this lab, we will not modify the P4 code. Instead, we will just compile it and download it to the switch s1. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```



```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** 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;
}
```

```
admin@lubuntu-vm:~/P4_Labs/lab2$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab2$
```

Figure 13. Compiling the P4 program using the VS Code terminal.

The command above invokes the *p4c* compiler to compile the *basic.p4* program. After executing the command, if there are no messages displayed in the terminal, then the P4 program was compiled successfully. You will see in the file explorer that two files were generated in the current directory:

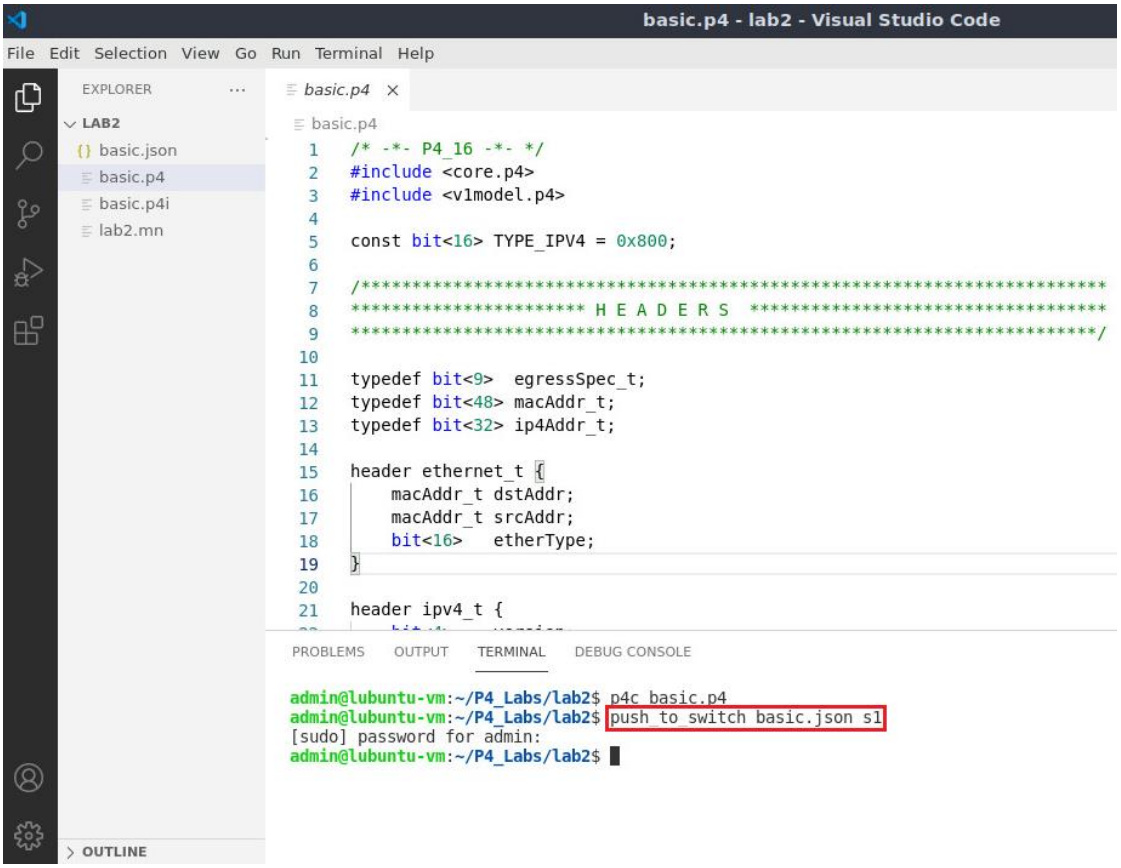
- *basic.json*: this file is generated by the *p4c* compiler if the compilation is successful. This file will be used by the software switch to describe the behavior of the data plane. You can think of this file as the binary or the executable to run on the switch data plane. The file type here is JSON because we are using the software switch. However, in hardware targets, most probably this file will be a binary file.
- *basic.p4i*: the output from running the preprocessor of the compiler on your P4 program.

At this point, we will only be focusing on the *basic.json* file.

Now that we have compiled our P4 program and generated the JSON file, we can download the program to the switch and start the switch daemon.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. The script accepts as input the JSON output of the *p4c* compiler, and the target switch name (e.g., s1). If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



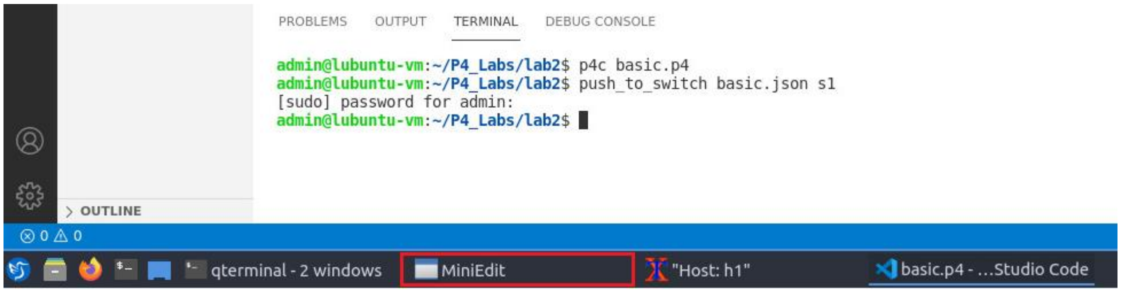
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** 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 ipv4_t {
```

```
admin@lubuntu-vm:~/P4_Labs/lab2$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab2$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab2$
```

Figure 14. Downloading the compiled program to switch s1.

3.3 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.



```
admin@lubuntu-vm:~/P4_Labs/lab2$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab2$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab2$
```

Figure 15. Maximizing the MiniEdit window.

Step 2. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

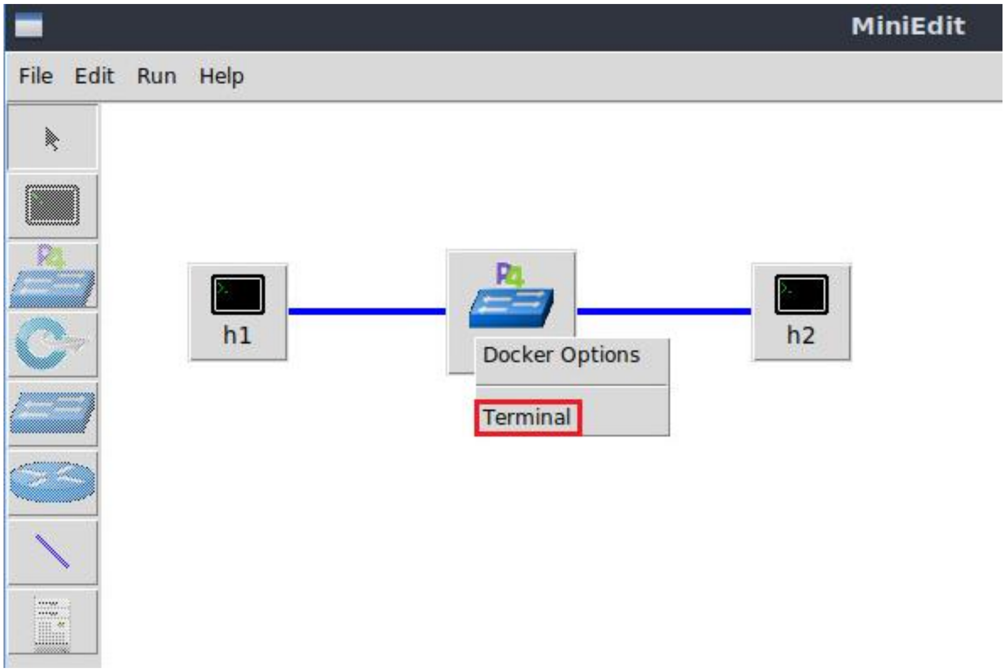
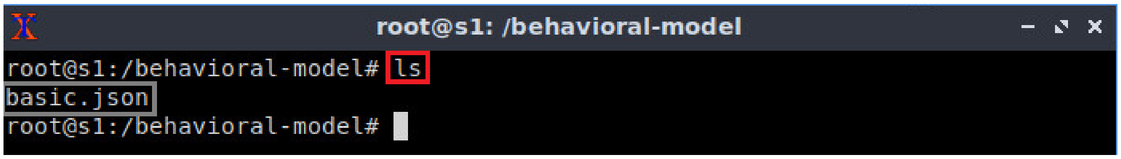


Figure 16. Starting the terminal on switch s1.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch's terminal.

Step 3. Issue the following command to list the files in the current directory.

```
ls
```



root@s1: /behavioral-model

```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model#
```

Figure 17. Displaying the contents of the current directory in the switch s1.

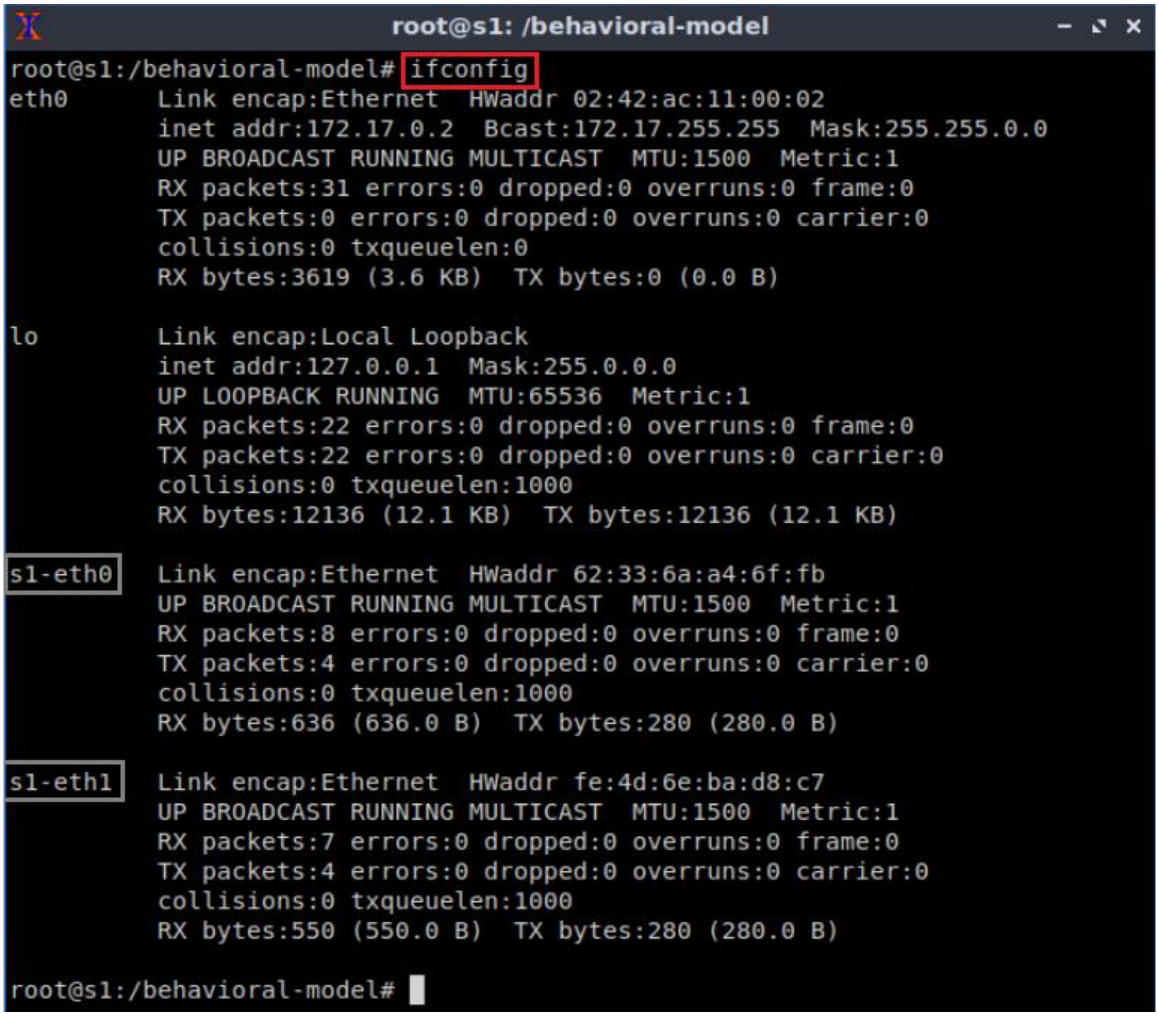
We can see that the switch contains the *basic.json* file that was downloaded after compiling the P4 program.

4 Configuring switch s1

4.1 Mapping P4 program's ports

Step 1. Issue the following command to display the interfaces in switch s1.

```
ifconfig
```



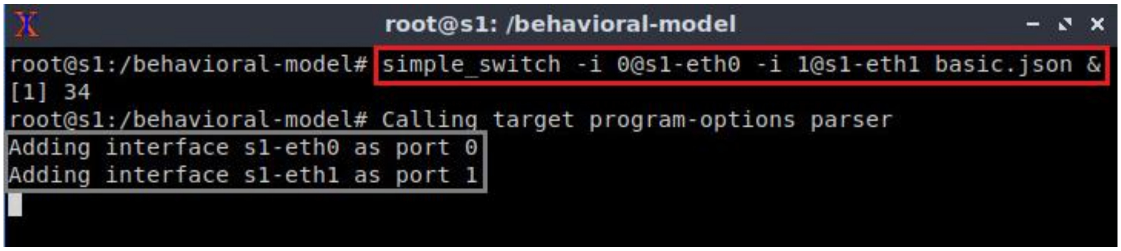
```
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3619 (3.6 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12136 (12.1 KB)  TX bytes:12136 (12.1 KB)

s1-eth0   Link encap:Ethernet  HWaddr 62:33:6a:a4:6f:fb
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:636 (636.0 B)  TX bytes:280 (280.0 B)

s1-eth1   Link encap:Ethernet  HWaddr fe:4d:6e:ba:d8:c7
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:550 (550.0 B)  TX bytes:280 (280.0 B)

root@s1:/behavioral-model# 
```

Figure 18. Displaying switch s1 interfaces.

We can see that the switch has the interfaces *s1-eth0* and *s1-eth1*. The interface *s1-eth0* on the switch s1 connects to the host h1. The interface *s1-eth1* on the switch s1 connects to the host h2.

Step 2. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
[1] 34
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 19. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

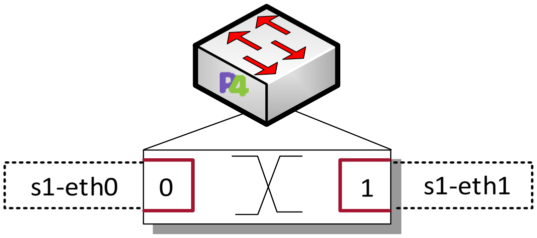
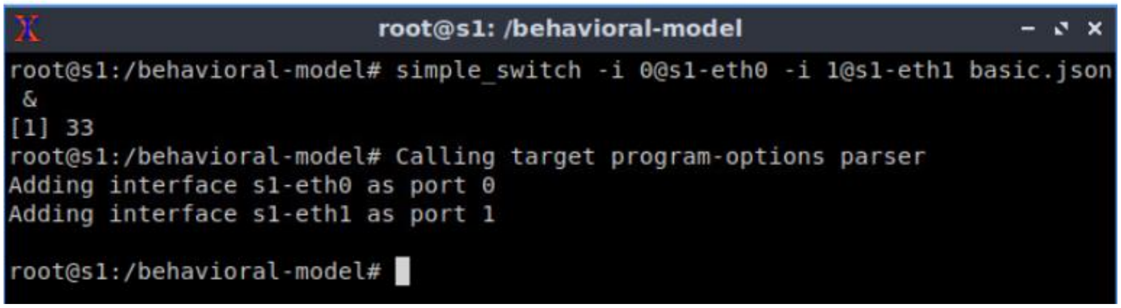


Figure 20. Ports 0 and 1 are mapped to the interfaces *s1-eth0* and *s1-eth1* of switch s1.

4.2 Loading the rules to the switch

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

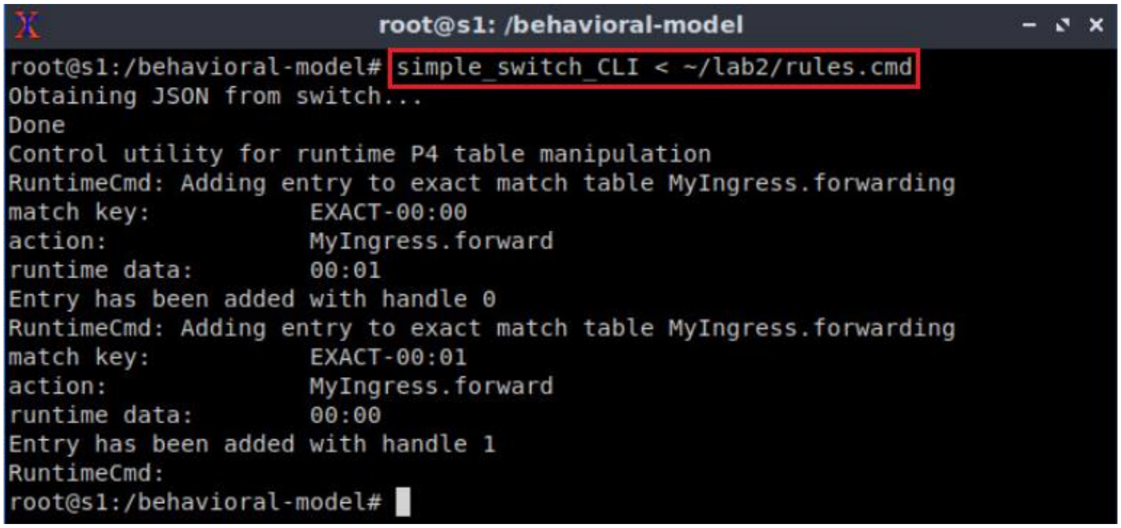


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 21. Returning to switch s1 CLI.

Step 2. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab2/rules.cmd
```



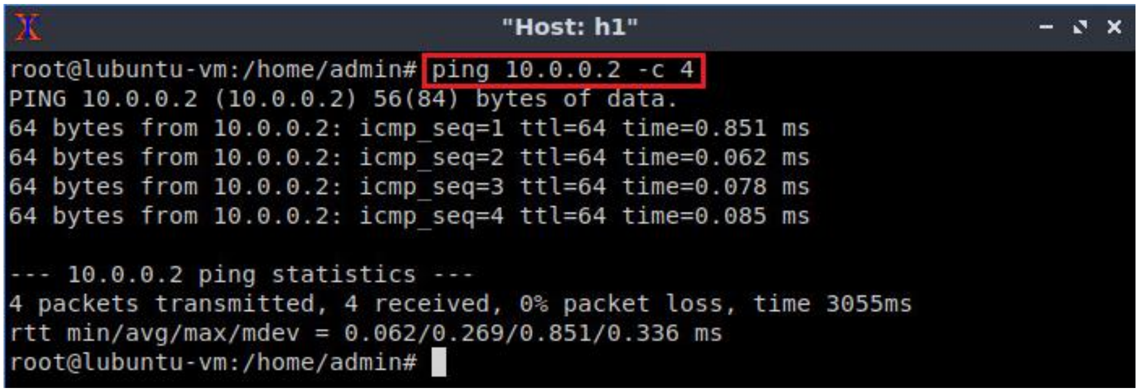
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab2/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# 
```

Figure 22. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

Step 3. Go back to host h1 terminal to test the connectivity between host h1 and host h2 by issuing the following command.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.851 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.062 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.078 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.085 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3055ms
rtt min/avg/max/mdev = 0.062/0.269/0.851/0.336 ms
root@lubuntu-vm:/home/admin# 
```

Figure 23. Performing a connectivity test between host h1 and host h2.

Now that the switch has a program with tables properly populated, the hosts can ping each other.

This concludes lab 2. Stop the emulation and then exit out of MiniEdit.

References

1. B. Trammell, M. Kuehlewind. “*RFC 7663: Report from the IAB workshop on stack evolution in a middlebox internet (SEMI).*” 2015. [Online]. Available: https://tools.ietf.org/html/rfc7663.
2. G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, S. Mangiante. ‘‘*De-ossifying the internet transport layer: A survey and future perspectives*,’’ IEEE Communications. Surveys and Tutorials., 2017.
3. The Register. “*VMware, Cisco stretch virtual LANs across the heavens*.” 2011. [Online]. Available: https://tinyurl.com/y6mxhqzn.
4. M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, “*Virtual eXtensible Local Area Network (VXLAN): a framework for overlaying virtualized layer 2 networks over layer 3 networks*,” RFC7348. [Online]. Available: http://www. rfc-editor.org/rfc/rfc7348.txt
5. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, ‘‘*P4: Programming protocol-independent packet processors,*’’ ACM SIGCOMM Computer Communications. 2014.
6. P4lang. “*Behavioral model*”. [Online]. Available: https://github.com/p4lang/behavioral-model.
7. V. Gurevich, A. Fingerhut, “*P4$_{16}$ for Intel Tofino^{TM} using Intel P4 Studio^{TM}*”. 2021 P4 Workshop, ONF. [Online]. Available: https://tinyurl.com/yckzkybf.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 3: P4 Program Building Blocks

Document Version: 01-25-2022

Contents

Overview

This lab describes the building blocks and the general structure of a P4 program. It maps the program's components to the Protocol-Independent Switching Architecture (PISA), a programmable pipeline used by modern whitebox switching hardware. The lab also demonstrates how to track an incoming packet as it traverses the pipeline of the switch. Such capability is very useful to debug and troubleshoot a P4 program.

Objectives

By the end of this lab, students should be able to:

1. Understand the PISA architecture.
2. Understand on high-level the main building blocks of a P4 program.
3. Map the P4 program components to the components of the programmable pipeline.
4. Trace the lifecycle of a packet as it traverses the pipeline.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client machine.

Device	**Account**	**Password**
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: The PISA architecture.
2. Section 2: Lab topology.
3. Section 3: Navigating through the components of a basic P4 program.
4. Section 4: Loading the P4 program.
5. Section 5: Configuring switch s1.
6. Section 6: Testing and verifying the P4 program.

1 The PISA architecture

1.1 The PISA architecture

The Protocol Independent Switch Architecture (PISA)[1] is a packet processing model that includes the following elements: programmable parser, programmable match-action pipeline, and programmable deparser, see Figure 1. The programmable parser permits the programmer to define the headers (according to custom or standard protocols) and to parse them. The parser can be represented as a state machine. The programmable match-action pipeline executes the operations over the packet headers and intermediate results. A single match-action stage has multiple memory blocks (e.g., tables, registers) and Arithmetic Logic Units (ALUs), which allow for simultaneous lookups and actions. Since some action results may be needed for further processing (e.g., data dependencies), stages are arranged sequentially. The programmable deparser assembles the packet headers back and serializes them for transmission. A PISA device is protocol independent. The P4 program defines the format of the keys used for lookup operations. Keys can be formed using packet header's information. The control plane populates table entries with keys and action data. Keys are used for matching packet information (e.g., destination IP address) and action data is used for operations (e.g., output port).

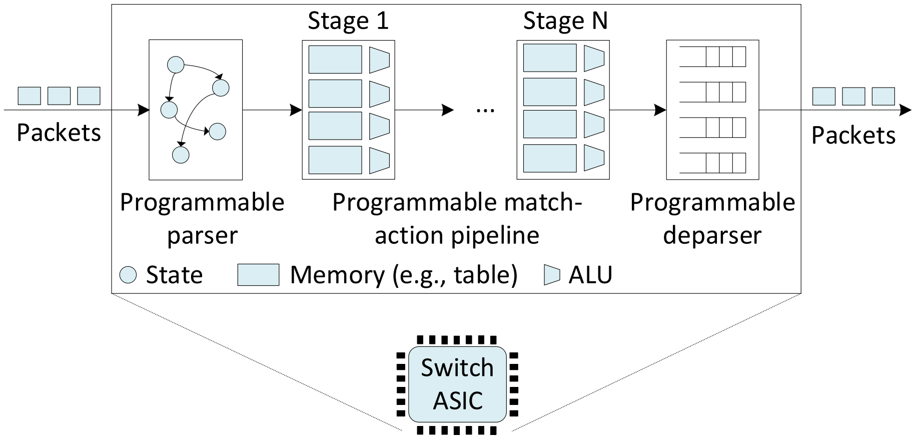


Figure 1. A PISA-based data plane.

Programmable switches do not introduce performance penalty. On the contrary, they may produce better performance than fixed-function switches. When compared with general purpose CPUs, ASICs remain faster at switching, and the gap is only increasing.

1.2 Programmable parser

The programmable parser permits the programmer to define the headers (according to custom or standard protocols) and to describe how the switch should process those headers. The parser de-encapsulates the headers, converting the original packet into a parsed representation of the packet. The programmer declares the headers that must be recognized and their order in the packet. The parser can be represented as a state machine without cycles (direct acyclic graph), with one initial state (start) and two final states (accept or reject).

1.3 Programmable match-action pipeline

The match-action pipeline implements the processing occurring at a switch. The pipeline consists of multiple identical stages (N stages are shown in Figure 1). Practical implementations may have 10/15 stages on the ingress and egress pipelines. Each stage contains multiple match-action units (4 units per stage in Figure 1). A match-action unit has a match phase and an action phase. During the match phase, a table is used to match a header field of the incoming packet against entries in the table (e.g., destination IP address). Note that there are multiple tables in a stage (4 tables per stage in Figure 1), which permit the switch to perform multiple matches in parallel over different header fields. Once a match occurs, a corresponding action is performed by the ALU. Examples of actions include: modify a header field, forward the packet to an egress port, drop the packet, and others. The sequential arrangement of stages allows for the implementation of serial dependencies. For example, if the result of an operation is needed prior to perform a second operation, then the compiler would place the first operation at an earlier stage than the second operation.

1.4 Programmable deparser

The deparser assembles back the packet and serializes it for transmission. The programmer specifies the headers to be emitted by the deparser. When assembling the packet, the deparser emits the specified headers followed by the original payload of the packet.

1.5 The V1Model

Figure 2 depicts the V1Model[2] architecture components. The V1Model architecture consists of a programmable parser, an ingress match-action pipeline, a traffic manager, an egress match-action pipeline, and a programmable deparser. The traffic manager schedules packets between input ports and output ports and performs packet replication (e.g., replication of a packet for multicasting). The V1Model architecture is implemented on top BMv2's simple_switch target[3].

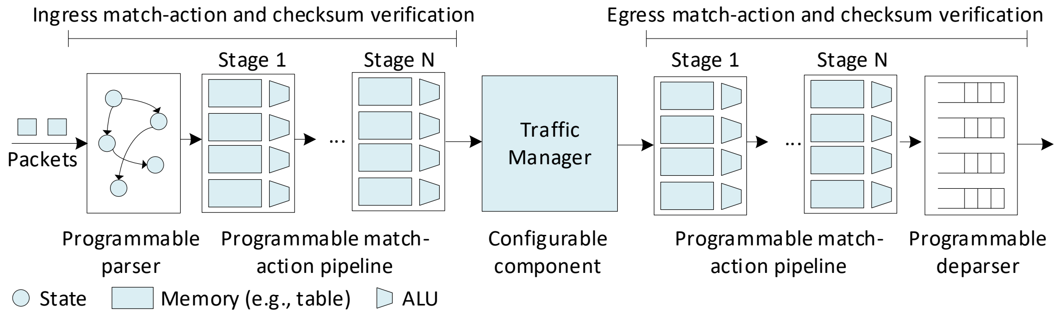

Ingress match-action and checksum verification
Egress match-action and checksum verification
Stage 1
Stage N
Stage 1
Stage N
Packets
Traffic Manager
...
...
Programmable parser
Programmable match-action pipeline
Configurable component
Programmable match-action pipeline
Programmable deparser
State
Memory (e.g., table)
ALU

Figure 2. The V1Model architecture.

1.6 P4 program mapping to the V1Model

The P4 program used in this lab is separated into different files. Figure 3 shows the V1Model and its associated P4 files. These files are as follows:

- *headers.p4*: this file contains the packet headers' and the metadata's definitions.
- *parser.p4*: this file contains the implementation of the programmable parser.
- *ingress.p4:* this file contains the ingress control block that includes match-action tables.
- *egress.p4*: this file contains the egress control block.
- *deparser.p4*: this file contains the deparser logic that describes how headers are emitted from the switch.
- *checksum.p4:* this file contains the code that verifies and computes checksums.
- *basic.p4:* this file contains the starting point of the program (main) and invokes the other files. This file must be compiled.

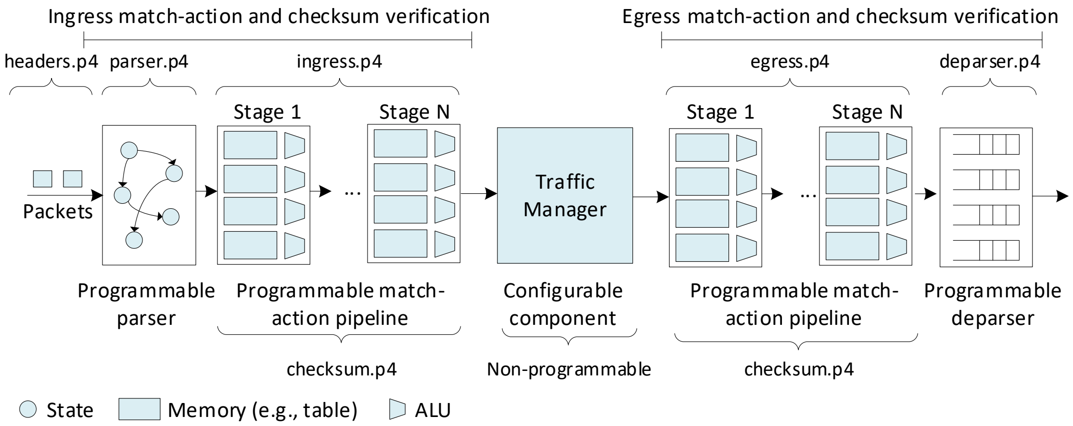


Figure 3. Mapping of P4 files to the V1Model's components.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

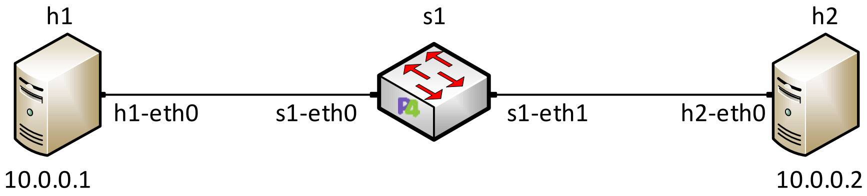


Figure 4. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

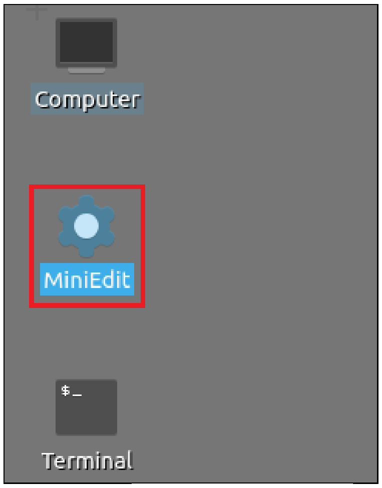
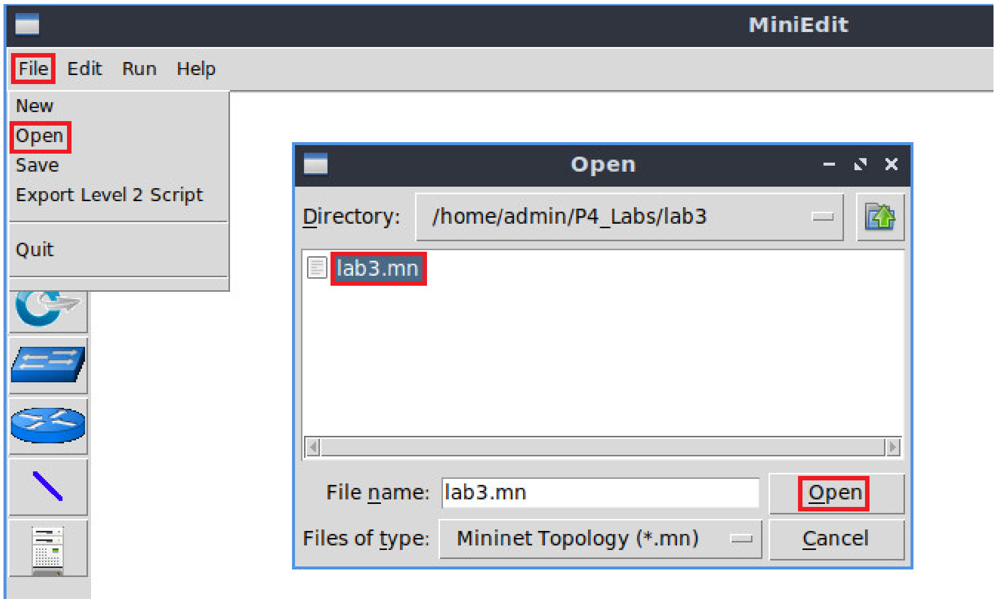


Figure 5. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the lab3 folder and search for the topology file called *lab3.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4_Labs/lab3

lab3.mn

File name: lab3.mn

Figure 6. Opening a topology in MiniEdit.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

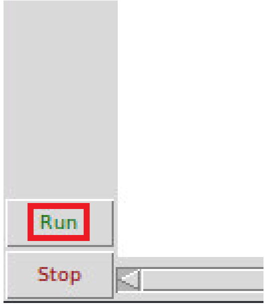


Figure 7. Running the emulation.

2.1 Starting host h1 and host h2

Step 1. Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

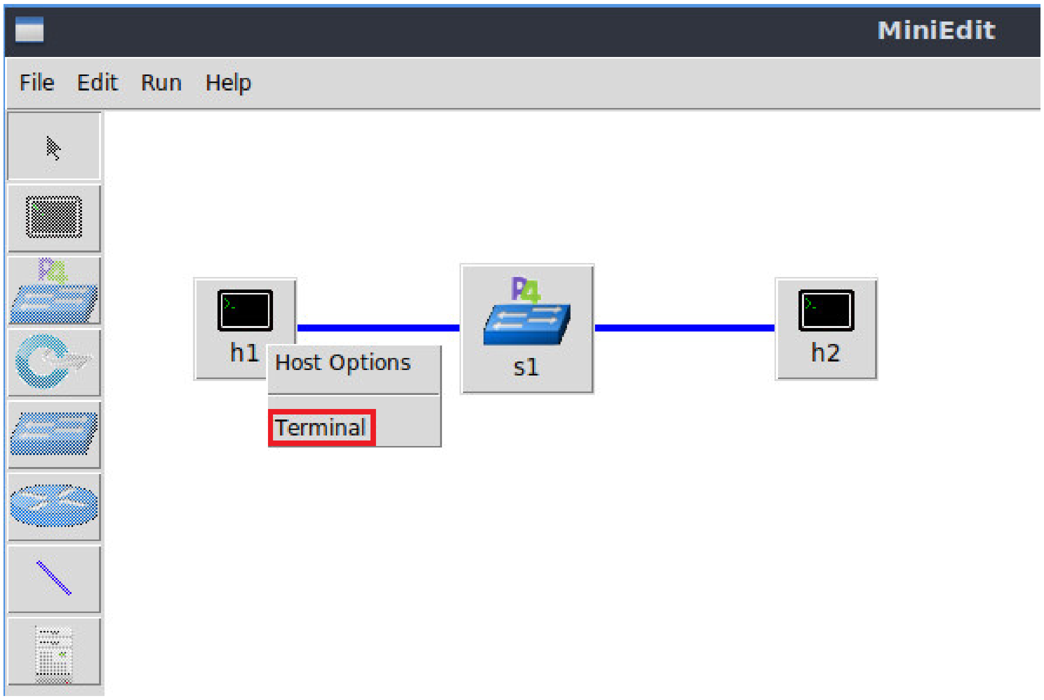
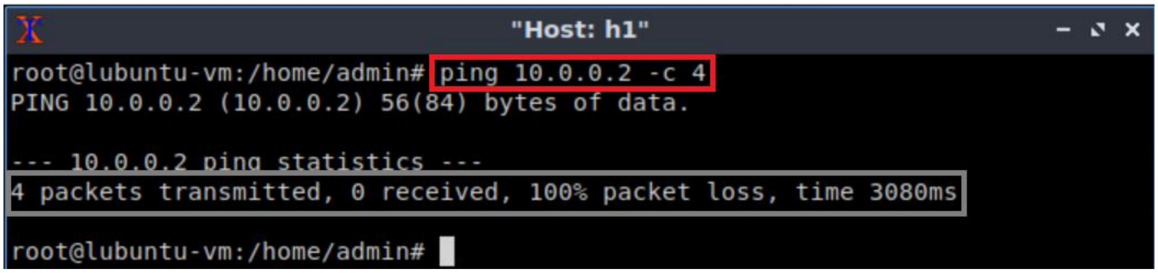


Figure 8. Opening a terminal on host h1.

Step 2. Test the connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3080ms

root@lubuntu-vm:/home/admin#
```

Figure 9. Performing a connectivity test between host h1 and host h2.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded on the switch.

3 Navigating through the components of a basic P4 program

This section shows the steps required to compile the P4 program. It illustrates the editor that will be used to modify the P4 program, and the P4 compiler that will produce a data plane program for the software switch.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the icon located on the desktop.

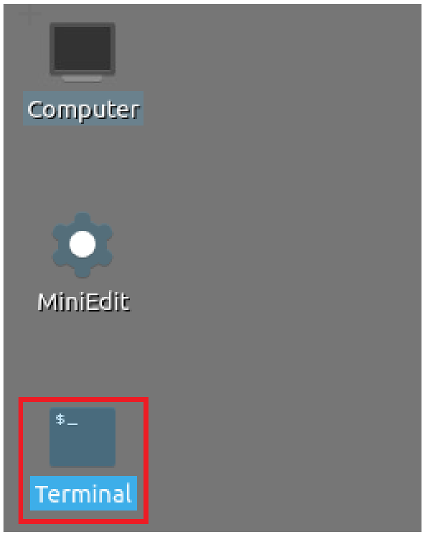
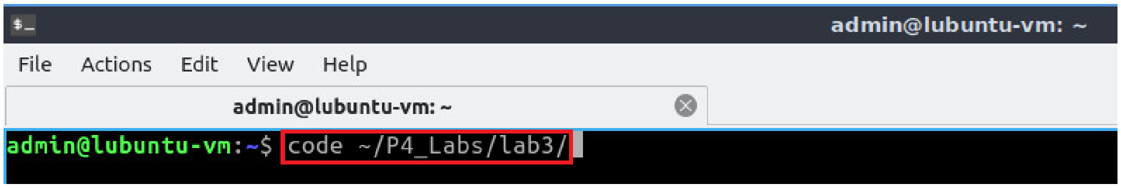


Figure 10. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab3/
```

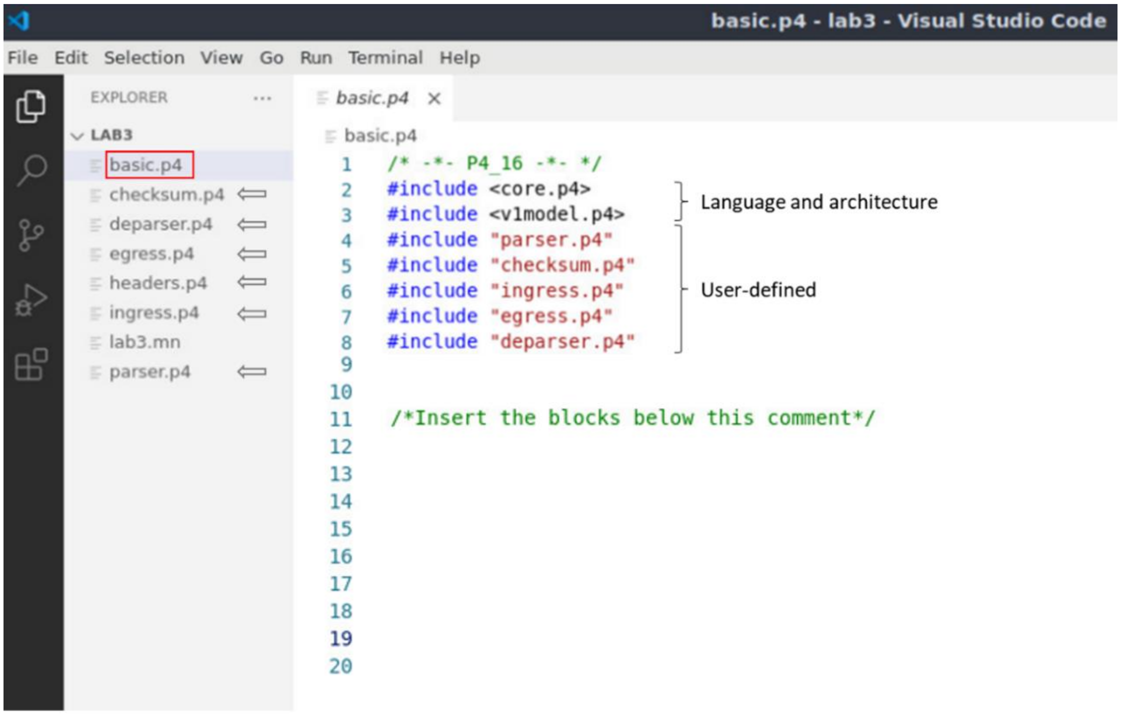


```
admin@lubuntu-vm:~$ code ~/P4_Labs/lab3/
```

Figure 11. Launching the editor and opening the lab3 directory.

3.2 Describing the components of the P4 program

Step 1. Once the previous command is executed, VS Code will start. Click on *basic.p4* in the file explorer panel on the left hand side to open the P4 program in the editor.



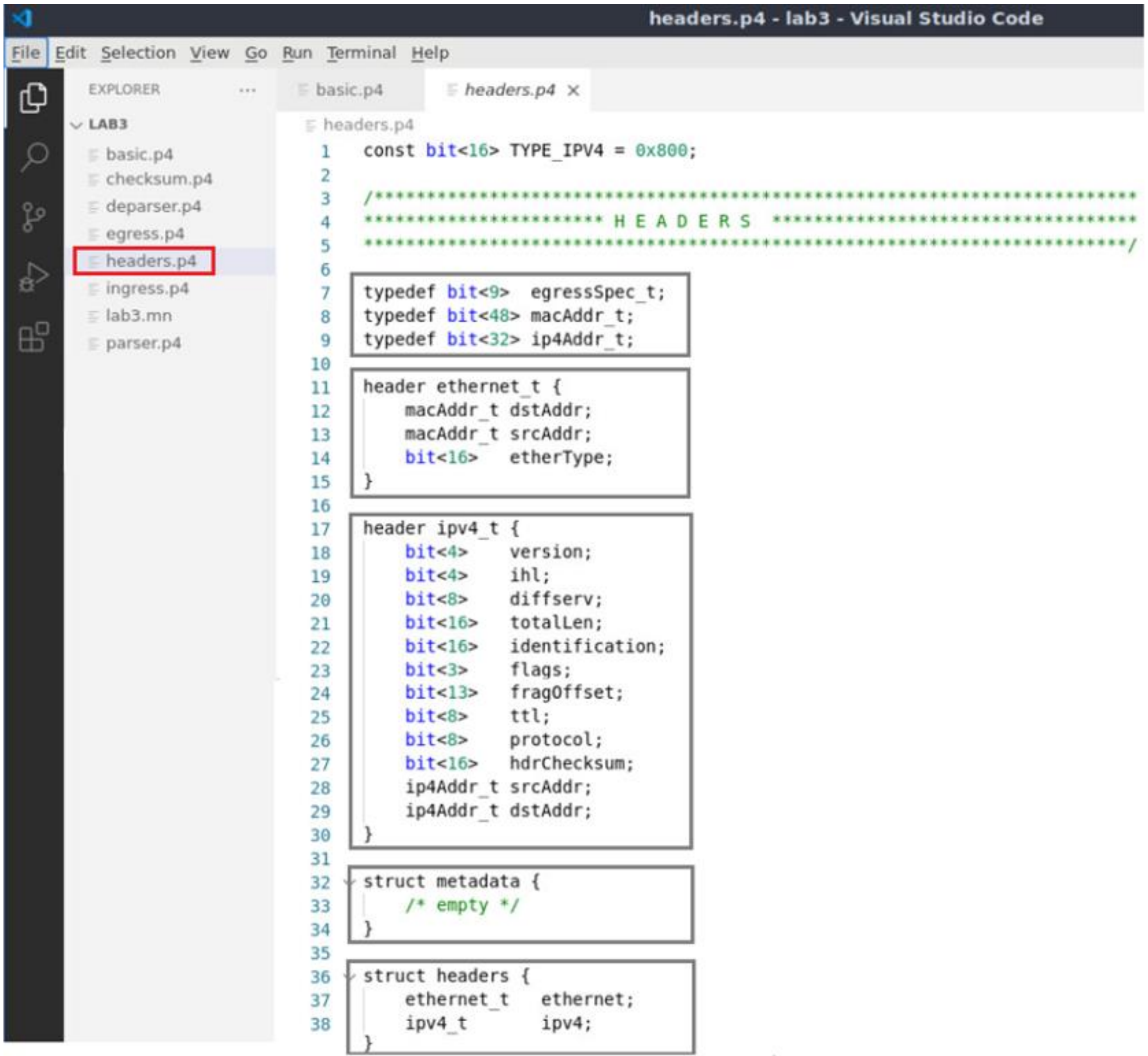
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/
```

Figure 12. The main P4 file and how it includes other user-defined files.

The *basic.p4* file includes the starting point of the P4 program and other files that are specific to the language (*core.p4*) and to the architecture (*v1model.p4*). To make the P4 program easier to read and understand, we separated the whole program into different files. Note how the files in the explorer panel correspond to the components of the V1Model. To use those files, the main file (*basic.p4*) must include them first. For example, to use the parser, we need to include the *parser.p4* file (`#include “parser.p4”`).

We will navigate through the files in sequence as they appear in the architecture.

Step 2. Click on the *headers.p4* file to display the content of the file.

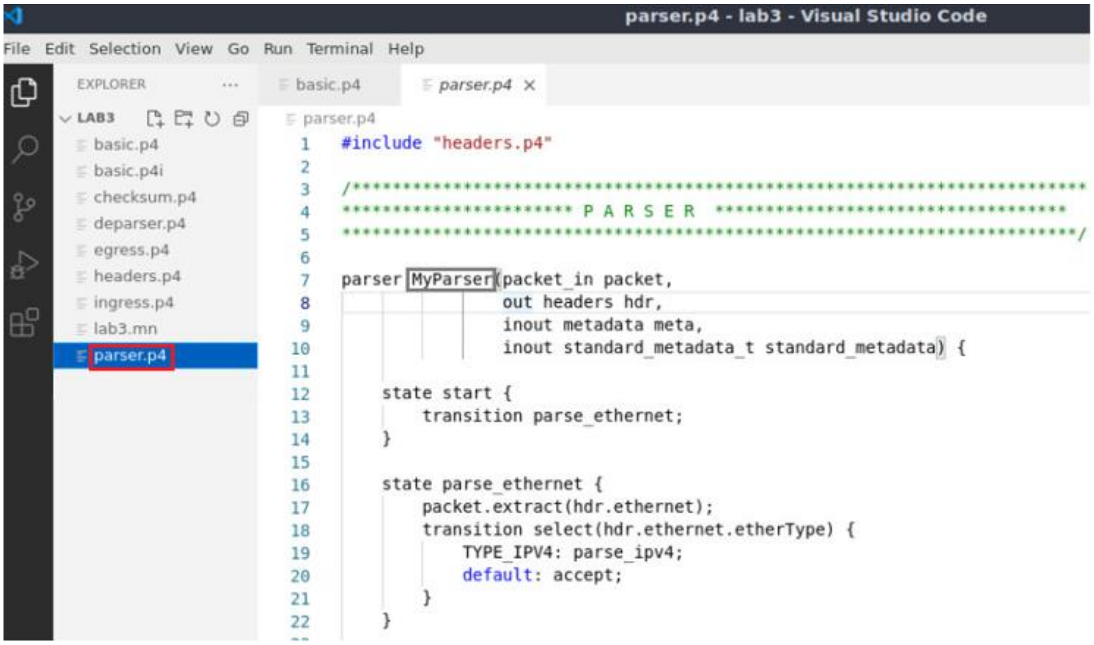


```
const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** 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 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;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

Figure 13. The defined headers.

The *headers.p4* above shows the headers that will be used in our pipeline. We can see that the ethernet and the IPv4 headers are defined. We can also see how they are grouped into a structure (`struct headers`). The `headers` name will be used throughout the program when referring to the headers. Furthermore, the file shows how we can use `typedef` to provide an alternative name to a type.

Step 3. Click on the *parser.p4* file to display the content of the parser.

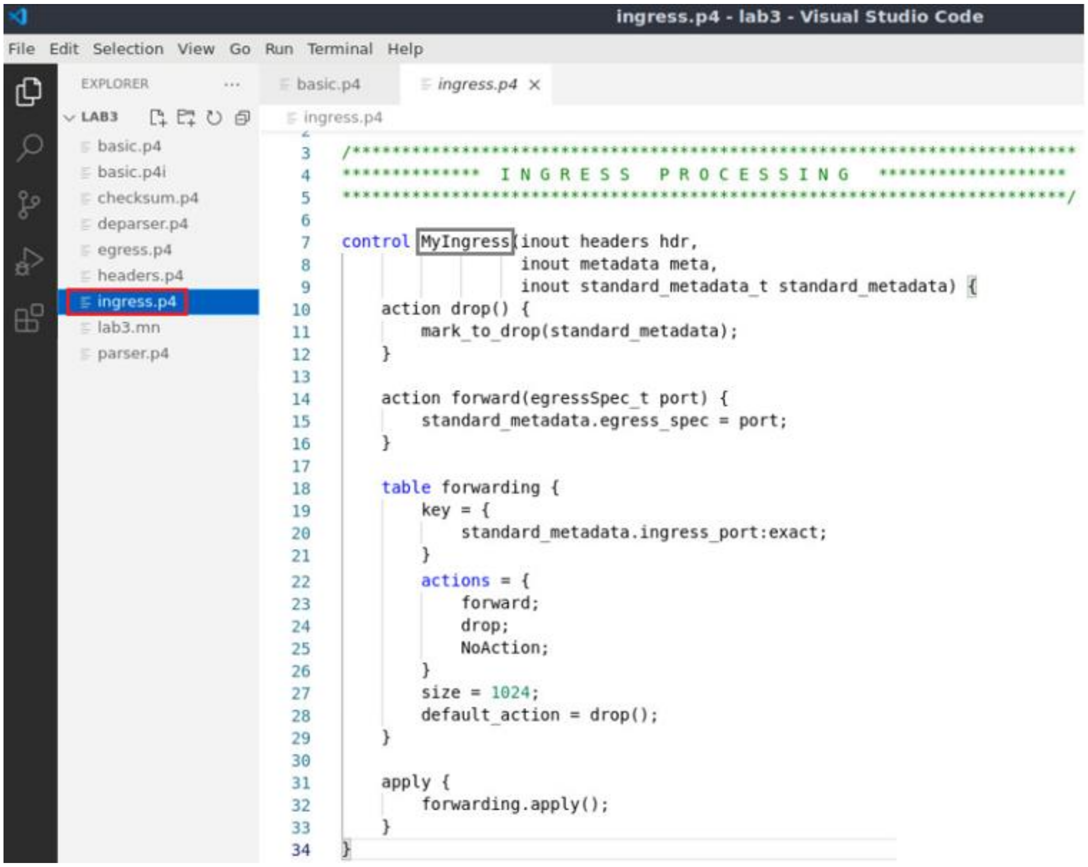


```
#include "headers.p4"

/*************************************************************************
*********************** 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 {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
```

Figure 14. The parser implementation.

The figure above shows the content of the parser.p4 file. We can see that the parser is already written with the name MyParser. This name will be used when defining the pipeline sequence.

Step 4. Click on the *ingress.p4* file to display the content of the file.

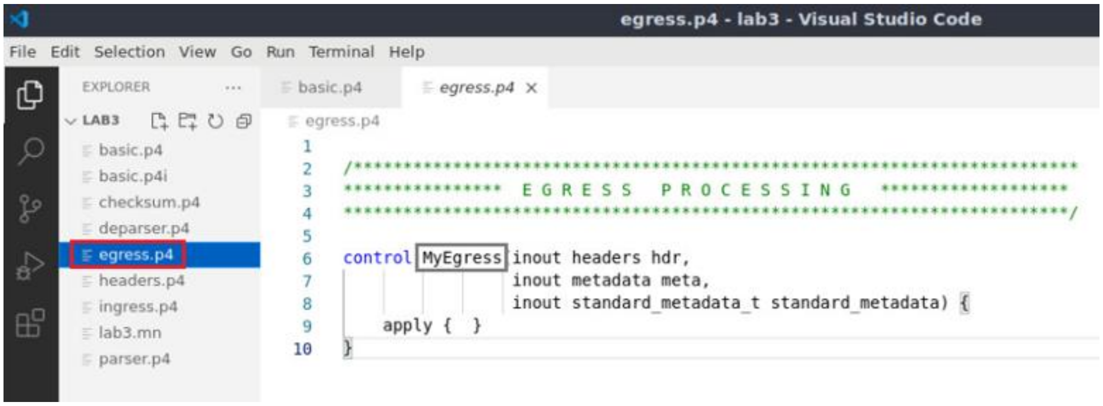


```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

    action forward(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }

    table forwarding {
        key = {
            standard_metadata.ingress_port:exact;
        }
        actions = {
            forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        forwarding.apply();
    }
}
```

Figure 15. The ingress component.

The figure above shows the content of the *ingress.p4* file. We can see that the ingress is already written with the name *MyIngress*. This name will be used when defining the pipeline sequence.

Step 5. Click on the *egress.p4* file to display the content of the file.

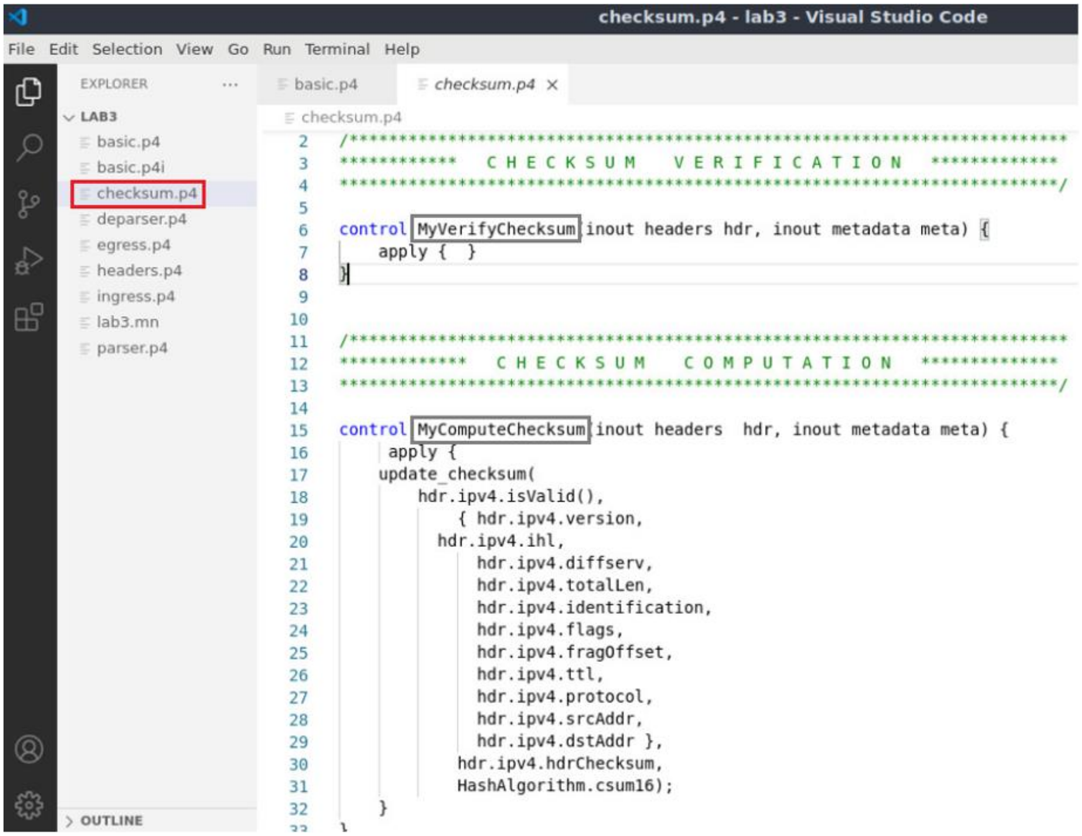


```
/******************************************************************************
*************** E G R E S S   P R O C E S S I N G ****************
******************************************************************************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}
```

Figure 16. The egress component.

The figure above shows the content of the *egress.p4* file. We can see that the egress is already written with the name *MyEgress*. This name will be used when defining the pipeline sequence.

Step 6. Click on the *checksum.p4* file to display the content of the file.

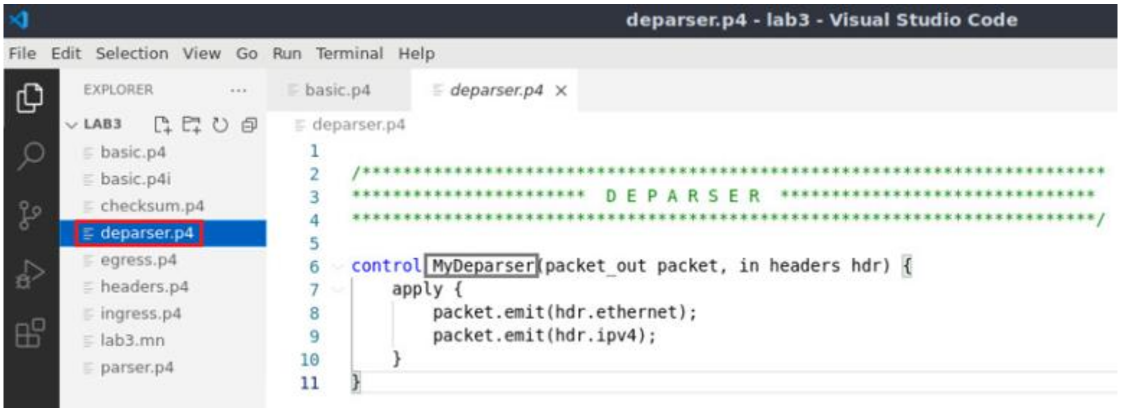


```
/******************************************************************************
************   C H E C K S U M   V E R I F I C A T I O N   *************
******************************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {  }
}


/******************************************************************************
*************   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 17. The checksum component.

The figure above shows the content of the *checksum.p4* file. We can see that the checksum is already written with two control blocks: `MyVerifyChecksum` and `MyComputeChecksum`. These names will be used when defining the pipeline sequence. Note that `MyVerifyChecksum` is empty since no checksum verification is performed in this lab.

Step 7. Click on the *deparser.p4* file to display the content of the file.



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

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

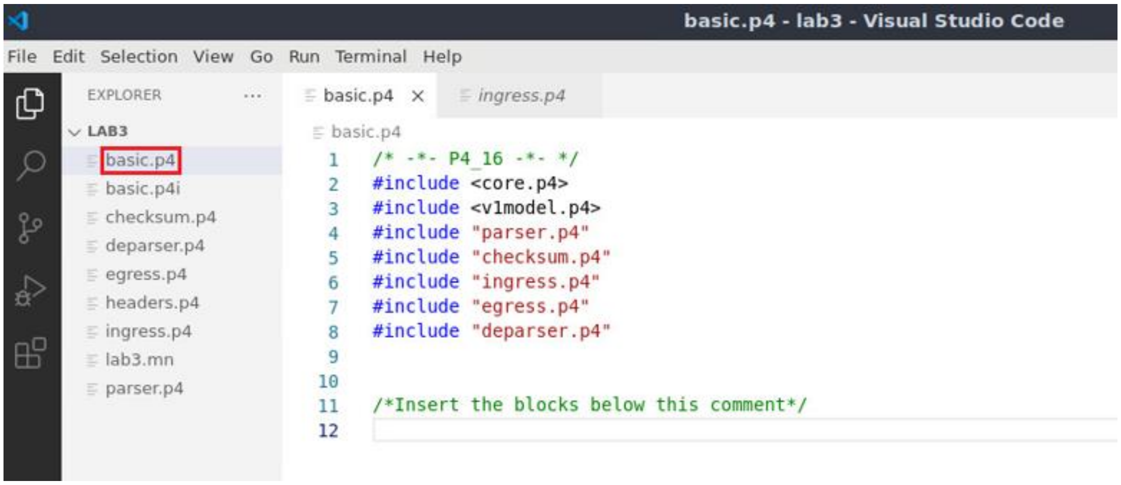
Figure 18. The deparser component.

The figure above shows the content of the *deparser.p4* file. We can see that the deparser is already written with two instructions that reassemble the packet.

3.3 Programming the pipeline sequence

Now it is time to write the pipeline sequence in the *basic.p4* program.

Step 1. Click on the *basic.p4* file to display the content of the file.

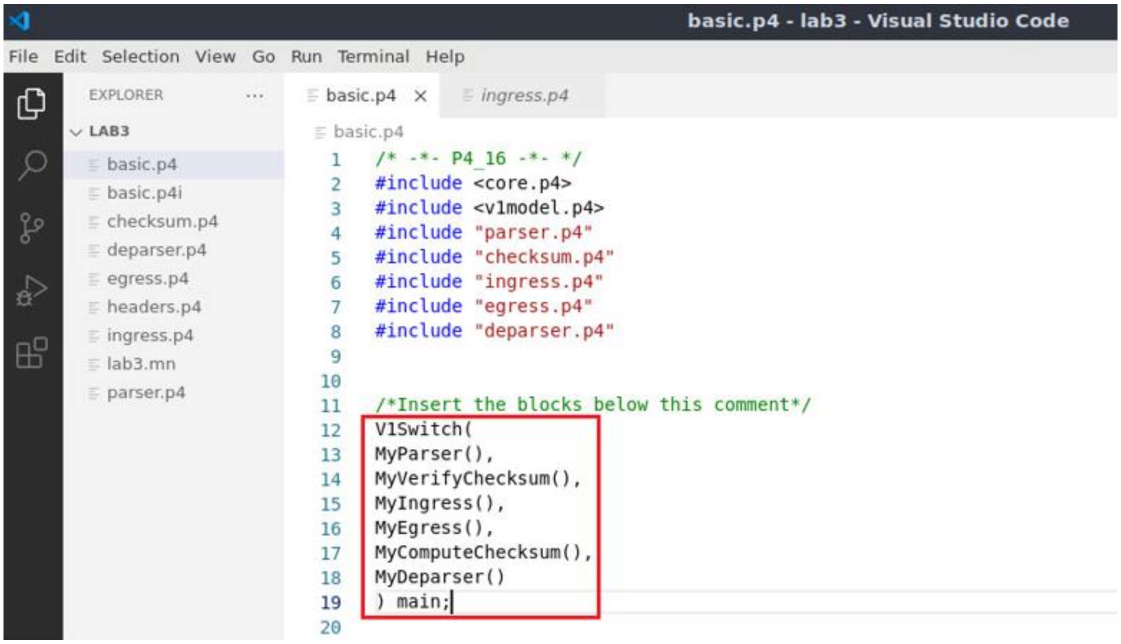


```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/

```

Figure 19. Selecting the *basic.p4* file.

Step 2. Write the following block of code at the end of the file

```
V1Switch (
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```



```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Figure 20. Writing the pipeline sequence in the *basic.p4* program

We can see here that we are defining the pipeline sequence according to the V1Model architecture. First, we start by the parser, then we verify the checksum. Afterwards, we specify the ingress block and the egress block, and we recompute the checksum. Finally, we specify the deparser.

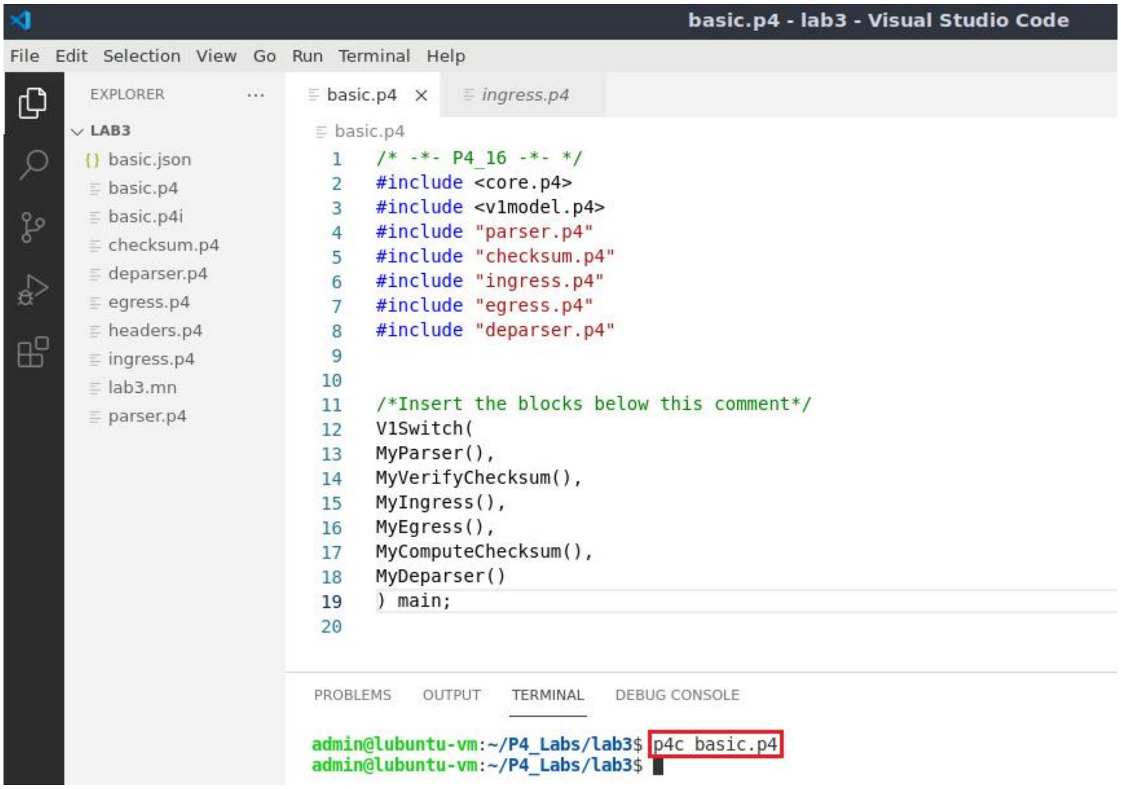
Step 3. Save the changes by pressing `Ctrl+s`.

4 Loading the P4 program

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```



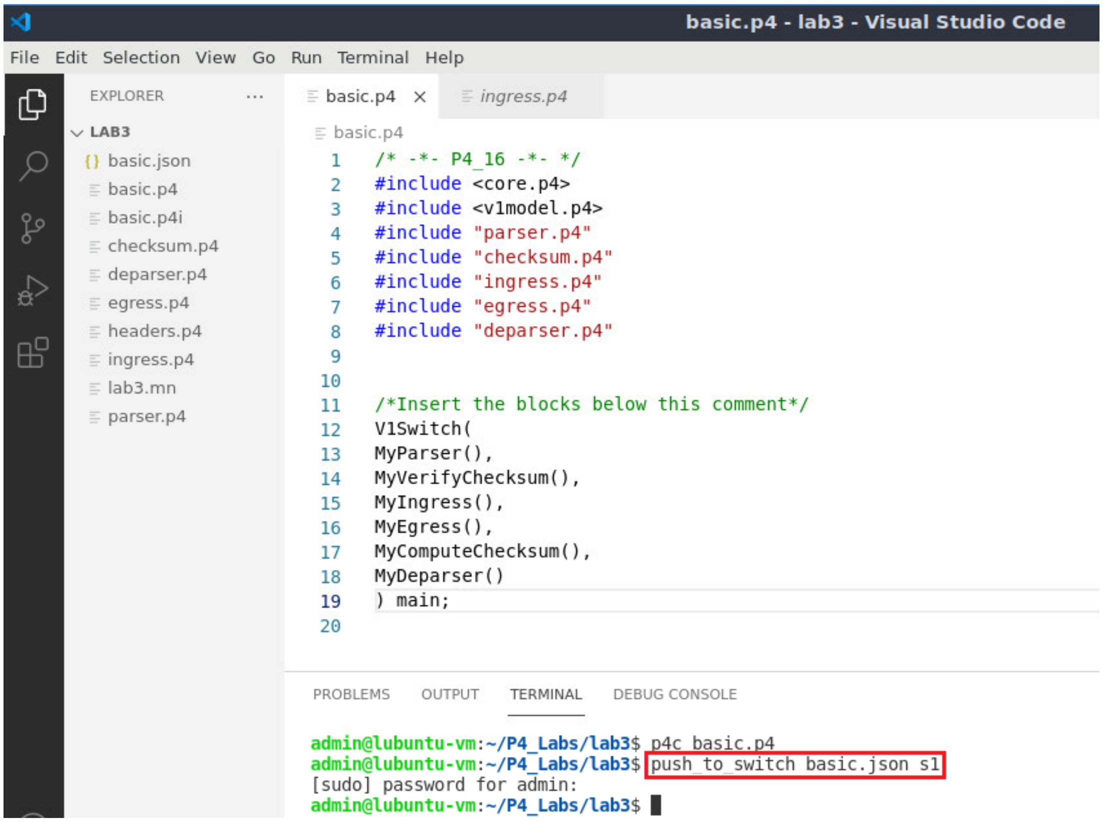
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

```
admin@lubuntu-vm:~/P4_Labs/lab3$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab3$ 
```

Figure 21. Compiling a P4 program.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the *p4c* compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


/*Insert the blocks below this comment*/
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

```
admin@lubuntu-vm:~/P4_Labs/lab3$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab3$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab3$ 
```

Figure 22. Downloading the P4 program to switch s1.

4.2 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.

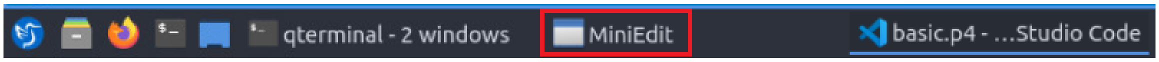


Figure 23. Maximizing the MiniEdit window.

Step 2. In MiniEdit, right-click on the P4 switch icon and start the *Terminal*.

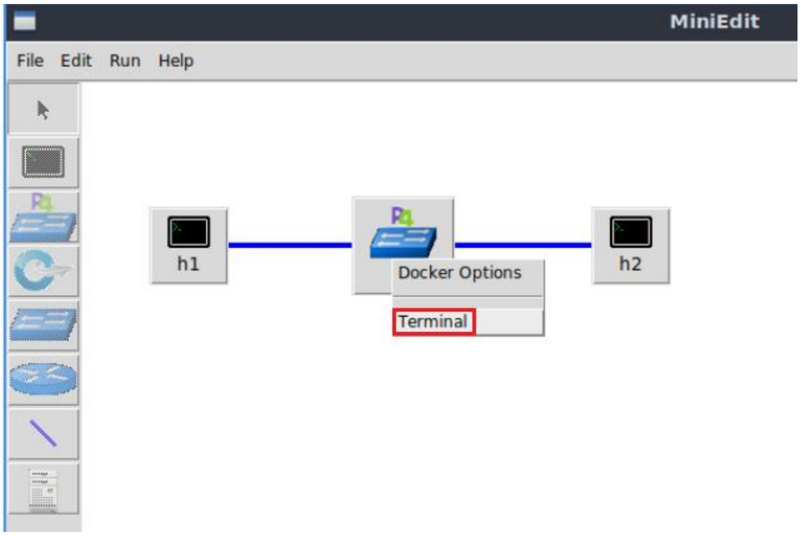
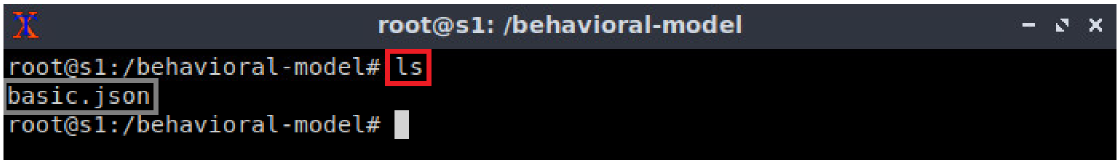


Figure 24. Starting the terminal on the switch.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

Step 3. Issue the command `ls` on the terminal of the switch s1 that was opened in the previous step.

```
ls
```



```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

Figure 25. Displaying the contents of the current directory in the switch s1.

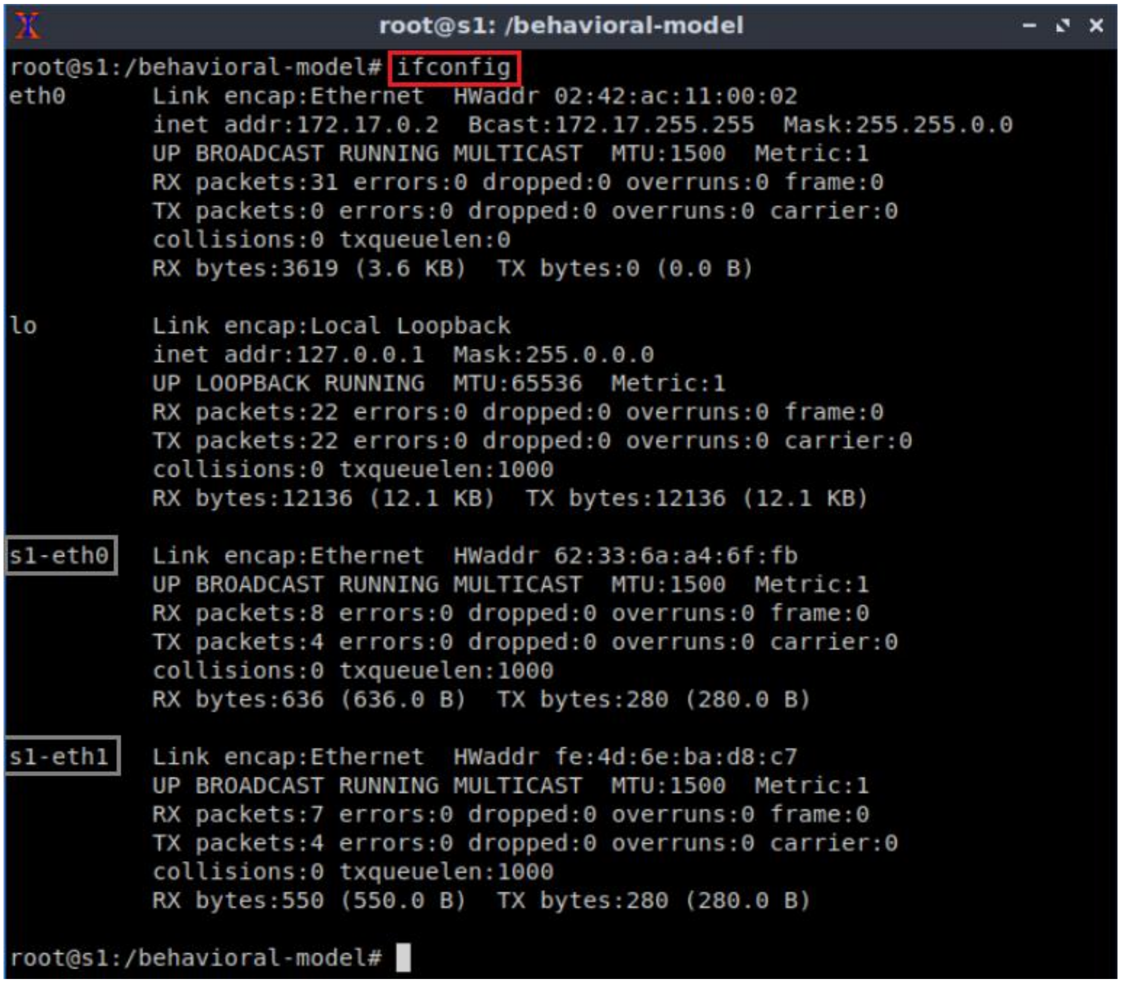
We can see that the switch contains the *basic.json* file that was downloaded to switch s1 after compiling the P4 program.

5 Configuring switch s1

5.1 Mapping the P4 program’s ports

Step 1. Issue the following command to display the interfaces on the switch s1.

```
ifconfig
```



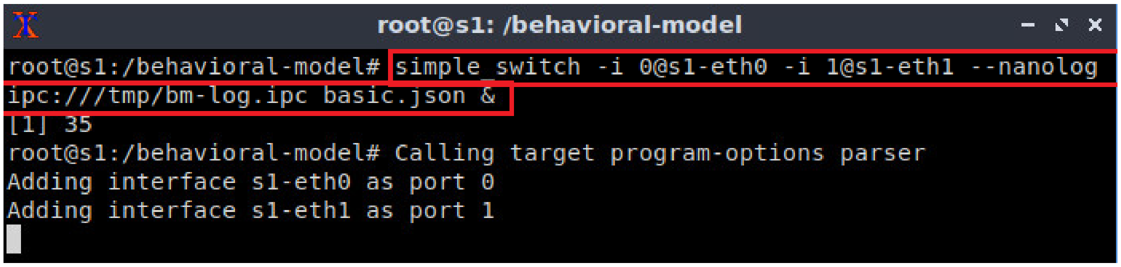
```
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3619 (3.6 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12136 (12.1 KB)  TX bytes:12136 (12.1 KB)

s1-eth0   Link encap:Ethernet  HWaddr 62:33:6a:a4:6f:fb
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:636 (636.0 B)  TX bytes:280 (280.0 B)

s1-eth1   Link encap:Ethernet  HWaddr fe:4d:6e:ba:d8:c7
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:550 (550.0 B)  TX bytes:280 (280.0 B)

root@s1:/behavioral-model#
```

Figure 26. Displaying switch s1 interfaces.

We can see that the switch has the interfaces *s1-eth0* and *s1-eth1*. The interface *s1-eth0* on the switch s1 connects host h1. The interface *s1-eth1* on the switch s1 connects host h2.

Step 2. Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog ipc:///tmp/bm-log.ipc
basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 35
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 27. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

The `--nanolog` option is used to instruct the switch daemon that we want to see the logs of the switch.

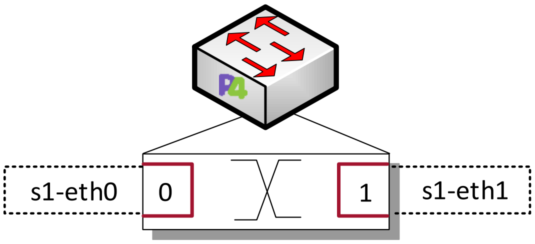
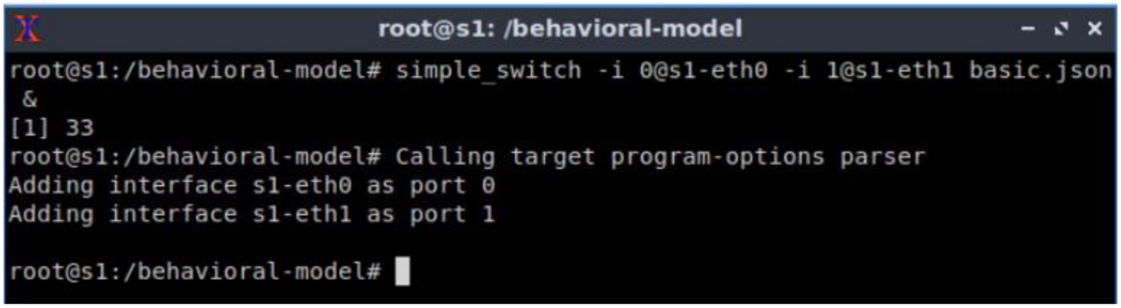


Figure 28. Mapping of the logical interface numbers (0, 1) to the Linux interfaces (*s1-eth0*, *s1-eth1*).

5.2 Loading the rules to the switch

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

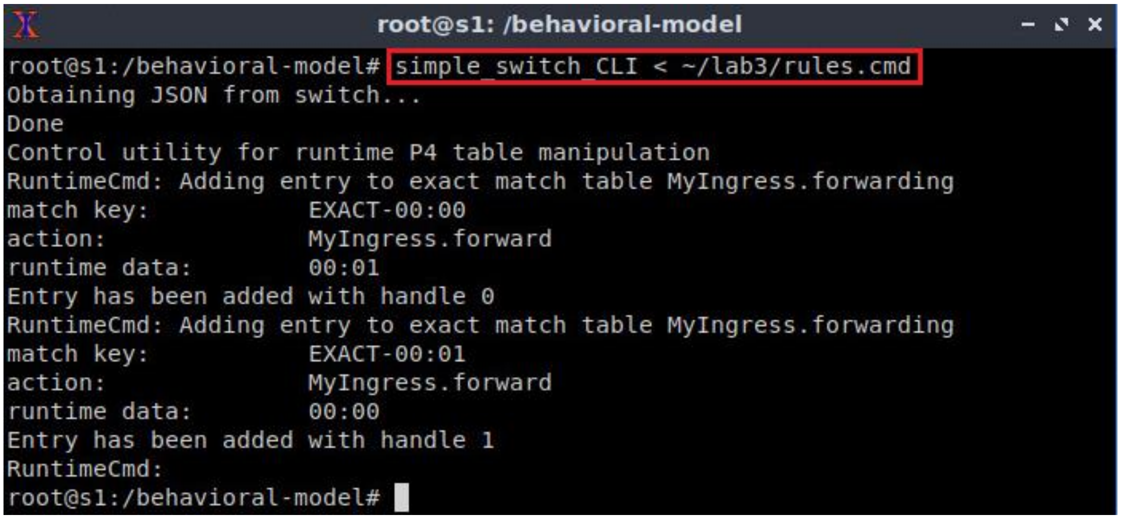


```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 29. Returning to switch s1 CLI.

Step 2. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab3/rules.cmd
```



```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab3/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# 
```

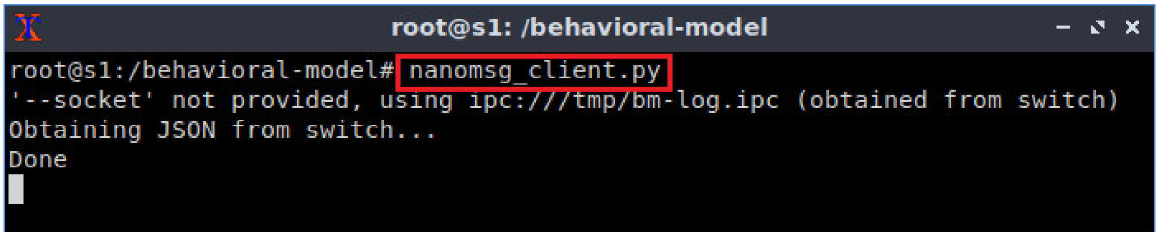
Figure 30. Loading the forwarding table entries into switch s1.

Now the forwarding table in the switch is populated.

6 Testing and verifying the P4 program

Step 1. Type the following command to initiate the `nanolog` client that will display the switch logs.

```
nanomsg_client.py
```

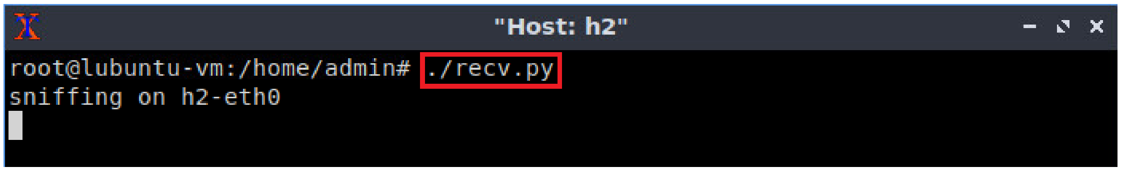


```
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 31. Displaying switch s1 logs.

Step 2. On host h2's terminal, type the command below so that the host starts listening for incoming packets.

```
./recv.py
```

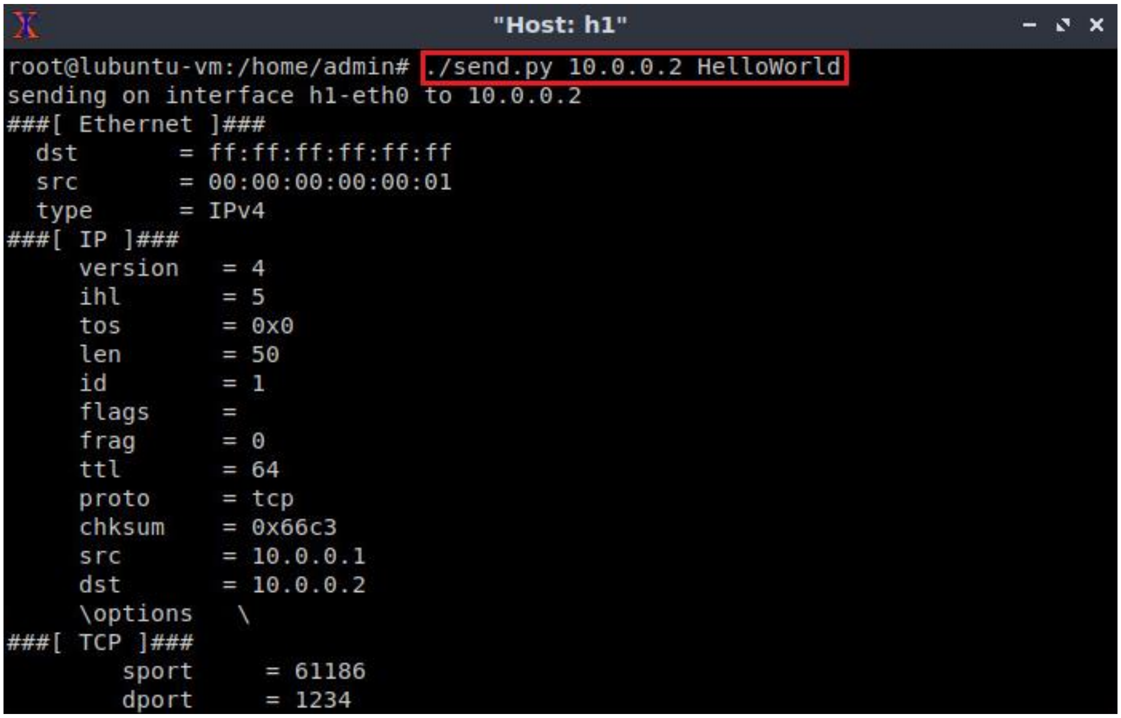


```
root@lubuntu-vm:/home/admin# ./recv.py
sniffing on h2-eth0
```

Figure 32. Listening for incoming packets in host h2.

Step 3. On host h1's terminal, type the following command to send a packet to host h2.

```
./send.py 10.0.0.2 HelloWorld
```

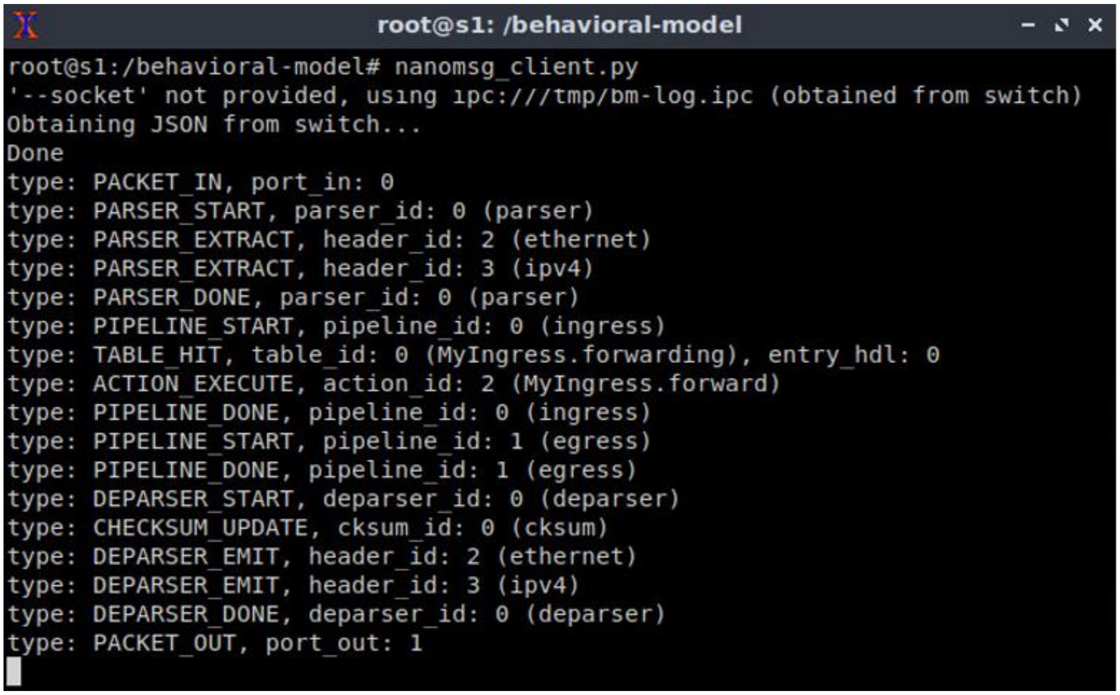


```
root@lubuntu-vm:/home/admin# ./send.py 10.0.0.2 HelloWorld
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 50
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x66c3
     src       = 10.0.0.1
     dst       = 10.0.0.2
     \options   \
###[ TCP ]###
        sport     = 61186
        dport     = 1234
```

Figure 33. Sending a test packet from host h1 to host h2.

Now that the switch has a program with tables properly populated, the hosts are able to reach each other.

Step 4. Go back to switch s1 terminal and inspect the logs.



```
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 34. Inspecting the logs in switch s1.

The figure above shows the processing logic as the packet enters switch s1. The packet arrives on port 0 (`port_in: 0`), then the parser starts extracting the headers. After the parsing is done, the packet is processed in the ingress and in the egress pipelines. Then, the checksum update is executed and the deparser reassembles and emits the packet using port 1 (`port_out: 1`).

Step 5. Verify that the packet was received on host h2.

This concludes lab 3. Stop the emulation and then exit out of MiniEdit.

References

1. C. Cascaval, D. Daly. "P4 Architectures." [Online]. Available: https://tinyurl.com/3zk8vs6a.
2. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.
3. P4lang/behavioral-model github repository*. “The BMv2 Simple Switch target*.” [Online]. Available: https://tinyurl.com/vrasamm.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 4: Parser Implementation

Document Version: 01-25-2022

Contents

Overview

This lab starts by describing how to define custom headers in a P4 program. It then explains how to implement a simple parser that parses the defined headers. The lab further shows how to track the parsing states of a packet inside the software switch.

Objectives

By the end of this lab, students should be able to:

1. Define custom headers in a P4 program.
2. Understand how the parser transitions between states and how it extracts the headers from the packets.
3. Implement a simple parser in P4.
4. Trace the parsed states when a packet enters to the switch.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client machine.

Device	**Account**	**Password**
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Defining the headers.
4. Section 4: Parser implementation.
5. Section 5: Loading the P4 program.
6. Section 6: Configuring switch s1.
7. Section 7: Testing and verifying the P4 program.
8. Section 8: Augmenting the P4 program to parse IPv6.
9. Section 9: Testing and verifying the augmented P4 program.

1 Introduction

1.1 Program headers and definitions

For several decades, the networking industry operated in a bottom-up approach. At the bottom of the system are the fixed-function Application Specific Integrated Circuits (ASICs), which enforce protocols, features, and processes available in the switch. Programmers and operators are limited to these capabilities when building their systems. Consequently, systems have features defined by ASIC vendors that are rigid and may not fit the network operators’ needs. Programmable switches and P4 represent a disruption of the networking industry by enabling a top-down approach for the design of network applications. With this approach, the programmer or network operator can precisely describe features and how packets are processed in the ASIC, using a high-level language, P4.

With the Protocol Independent Switch Architecture (PISA)[1], the programmer defines the headers and corresponding parser as well as actions executed in the match-action pipeline and the deparser. The programmer has the flexibility of defining custom headers (i.e., a header not standardized). Such capability is not available in non-programmable devices.

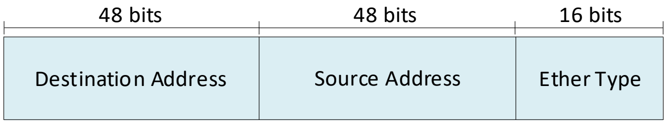


Figure 1. Ethernet header.

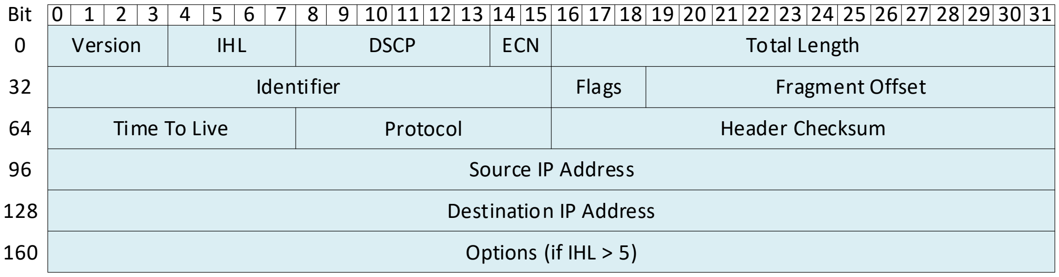


Figure 2. IPv4 header.

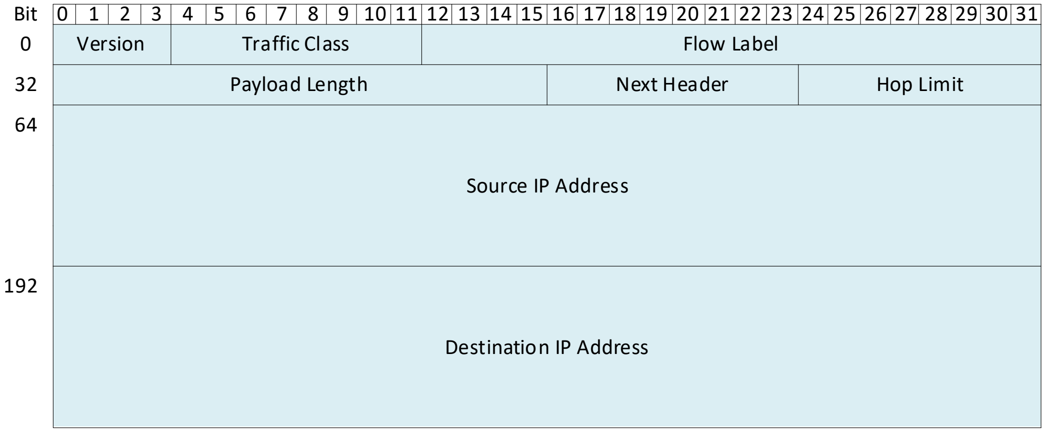
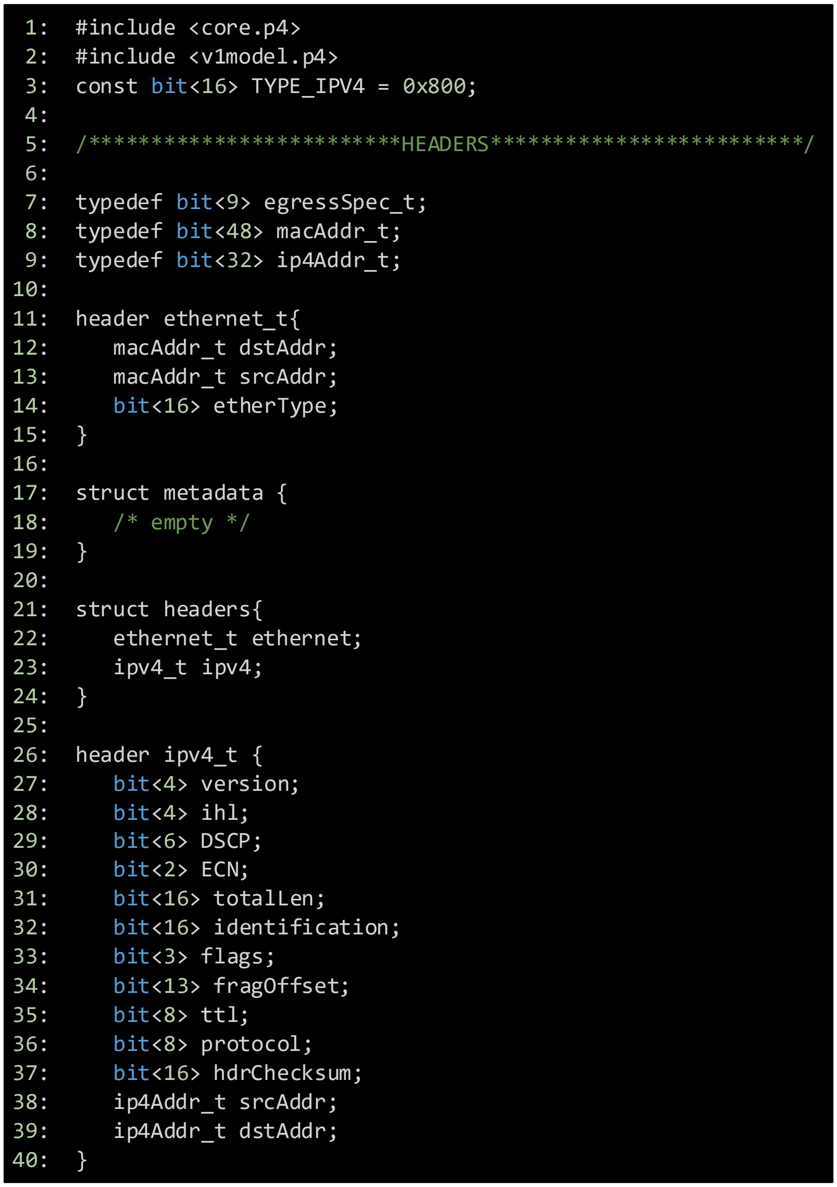


Figure 3. IPv6 header.

Figure 4 shows an excerpt of a P4 program where the headers are defined. This is typically written at the top of the program before the parsing starts. We can see that the programmer defined a header corresponding to Ethernet (lines 11-15). The Ethernet header fields are shown in Figure 1.

The programmer also defined an IPv4 header (lines 26-40). The IPv4 header format is shown in Figure 2 and the IPv6 header is shown in Figure 3.



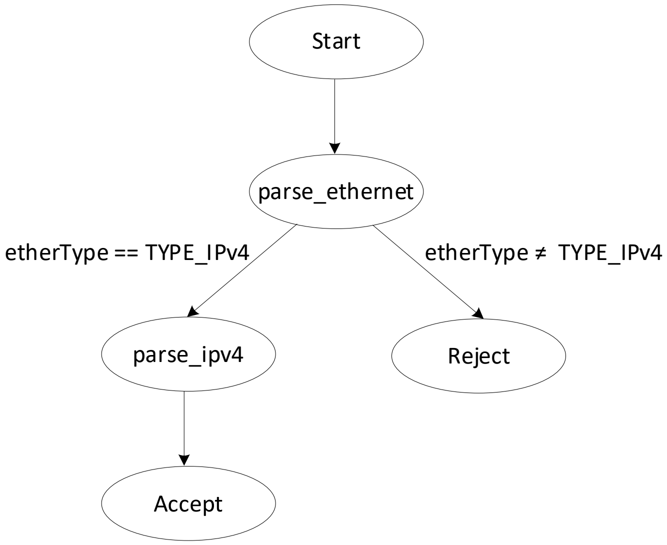
```
 1:  #include <core.p4>
 2:  #include <v1model.p4>
 3:  const bit<16> TYPE_IPV4 = 0x800;
 4:
 5:  /*************************HEADERS*************************/
 6:
 7:  typedef bit<9> egressSpec_t;
 8:  typedef bit<48> macAddr_t;
 9:  typedef bit<32> ip4Addr_t;
10:
11:  header ethernet_t{
12:     macAddr_t dstAddr;
13:     macAddr_t srcAddr;
14:     bit<16> etherType;
15:  }
16:
17:  struct metadata {
18:     /* empty */
19:  }
20:
21:  struct headers{
22:     ethernet_t ethernet;
23:     ipv4_t ipv4;
24:  }
25:
26:  header ipv4_t {
27:     bit<4> version;
28:     bit<4> ihl;
29:     bit<6> DSCP;
30:     bit<2> ECN;
31:     bit<16> totalLen;
32:     bit<16> identification;
33:     bit<3> flags;
34:     bit<13> fragOffset;
35:     bit<8> ttl;
36:     bit<8> protocol;
37:     bit<16> hdrChecksum;
38:     ip4Addr_t srcAddr;
39:     ip4Addr_t dstAddr;
40:  }
```

Figure 4. Program headers and definitions.

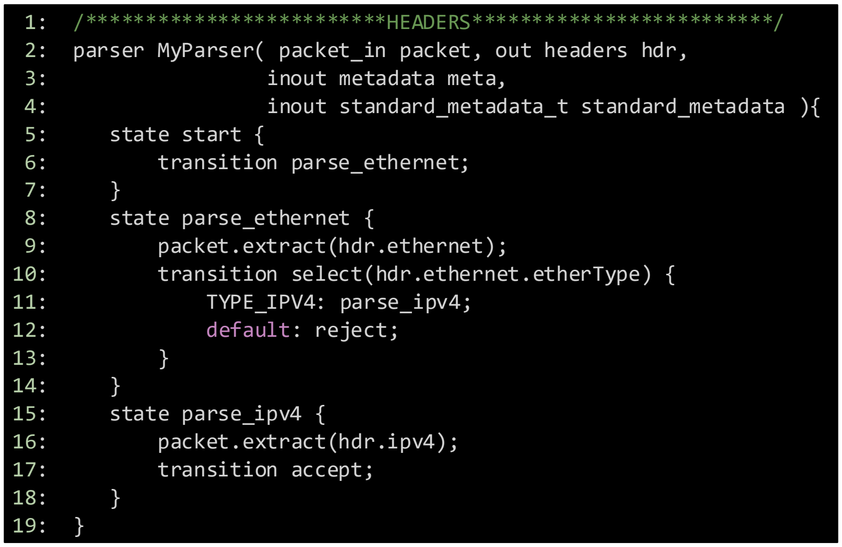
The code starts by including the *core.p4* file (line 1) which defines some common types and variables used in all P4 programs. For instance, the `packet_in` and `packet_out` extern types which represent incoming and outgoing packets, respectively, are declared in *core.p4*[2]. Next, the *v1model.p4*[3] file is included (line 2) to define the V1Model architecture[4] and all its externs used when writing P4 programs. Line 3 creates a 16-bit constant TYPE_IPV4 with the value 0x800. This means that TYPE_IPV4 can be used later in the P4 program to reference the value 0x800. The typedef declarations (lines 7 - 9) are used to assign alternative names to types. Subsequently, the headers and the metadata structs that will be used in the program are defined. These headers are customized depending on how the programmer wants the packets to be parsed. The program in Figure 4 defines the Ethernet header (lines 11-15) and the IPv4 header (lines 26-40). The declarations inside each header are usually written after referring to the standard specifications of the protocol. Note in the ethernet_t header the macAddr_t is used rather than using a 48-bit field. Lines 17 - 19 show how to declare user-defined metadata, which are passed from one block to another as the packet propagates through the architecture. For simplicity, this program does not require any user metadata.

1.2 Programmable parser

The programmable parser permits the programmer to describe how the switch will process the packet. The parser de-encapsulates the headers, converting the original packet into a parsed representation of the packet. The parser can be represented as a state machine without cycles (direct acyclic graph), with one initial state (start) and two final states (accept or reject).



(a)



```
 1:  /***********************HEADERS***********************/
 2:  parser MyParser( packet_in packet, out headers hdr,
 3:                  inout metadata meta,
 4:                  inout standard_metadata_t standard_metadata ){
 5:     state start {
 6:         transition parse_ethernet;
 7:     }
 8:     state parse_ethernet {
 9:         packet.extract(hdr.ethernet);
10:         transition select(hdr.ethernet.etherType) {
11:             TYPE_IPV4: parse_ipv4;
12:             default: reject;
13:         }
14:     }
15:     state parse_ipv4 {
16:         packet.extract(hdr.ipv4);
17:         transition accept;
18:     }
19:  }
```

(b)

Figure 5. Example of a parser. (a) Graphical representation of the parser. (b) In P4, the parser always starts with the initial state called `start`. First, we transition unconditionally to `parse_ethernet`. Then, we can create some conditions to direct the parser. Finally, when we transition to the `accept` state, the packet is moved to the ingress block of the pipeline. A packet that reaches the `reject` state will be dropped.

Figure 5a shows the graphical representation of the parser and Figure 5b its corresponding P4 code. Note that packet is an instance of the `packet_in` extern (specific to V1Model) and is passed as a parameter to the parser. 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 `parse_ipv4` state if the `etherType` field corresponds to IPv4. The state transitions to the `reject` if it is not an IPv4 header, as shown in the figure above (Line 12). In the `parse_ipv4` state, the IPv4 header is extracted, and the program unconditionally transitions to the `accept` state.

2 Lab topology

Let’s get started with creating a simple Mininet topology using MiniEdit.

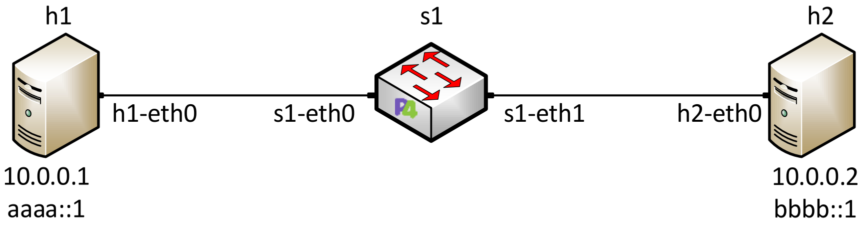


Figure 6. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

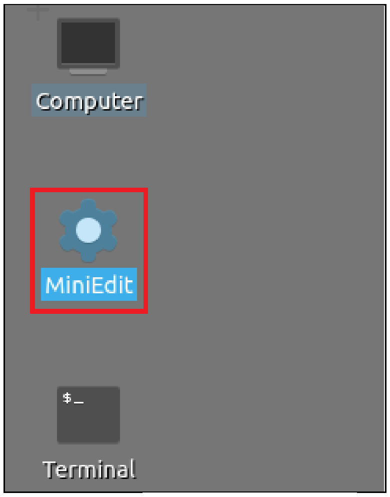
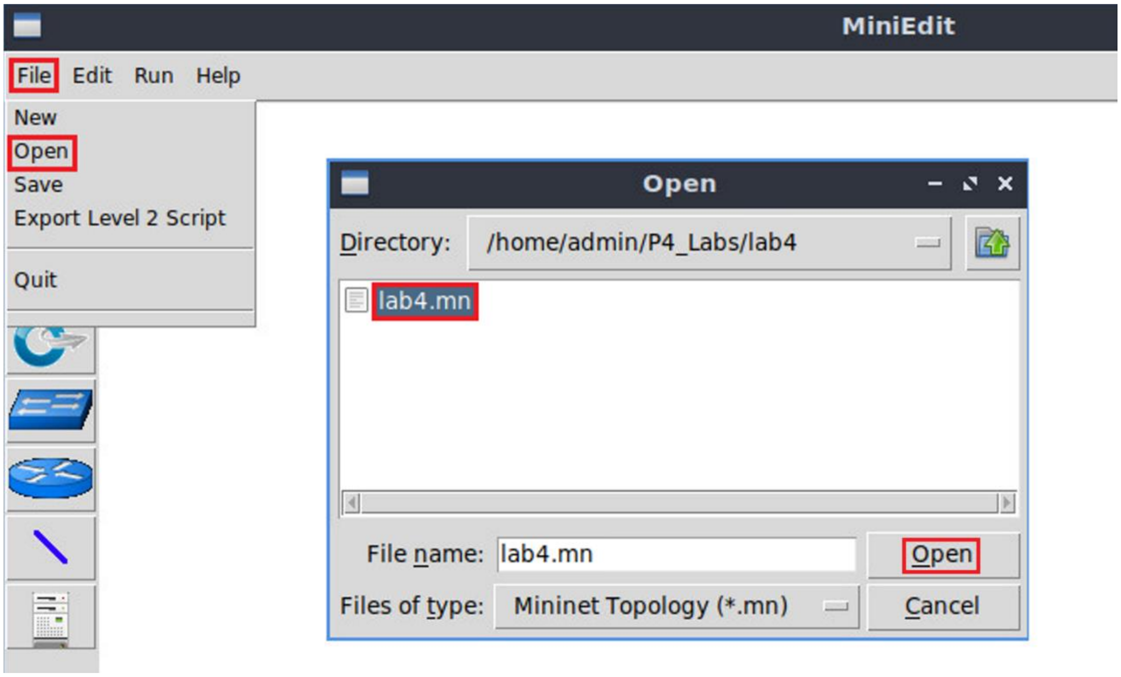


Figure 7. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab4* folder and search for the topology file called *lab4.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4_Labs/lab4

lab4.mn

File name: lab4.mn

Files of type: Mininet Topology (*.mn)

Figure 8. MiniEdit's *Open* dialog.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

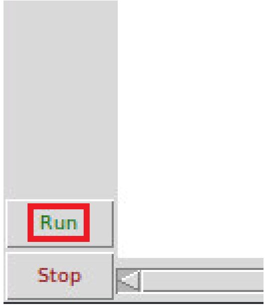


Figure 9. Running the emulation.

2.1 Starting host h1 and host h2

Step 1. Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

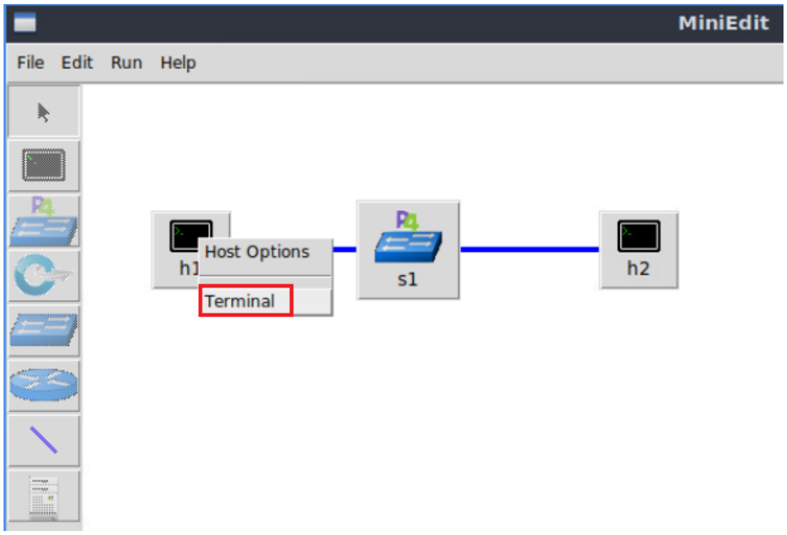


Figure 10. Opening a terminal on host h1.

3 Defining the program's headers

This section demonstrates how to define custom headers in a P4 program. It also shows how to use constants and typedefs to make the program more readable.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the icon located on the desktop.

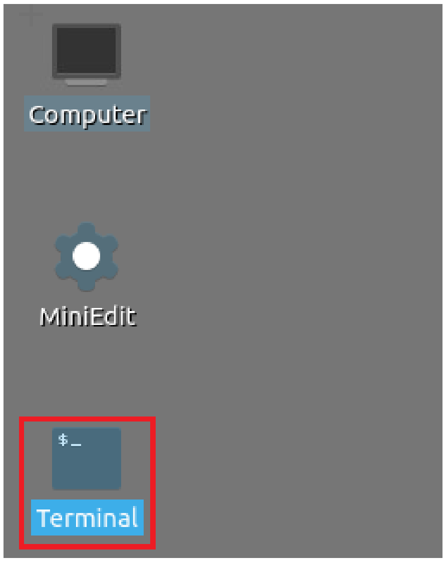
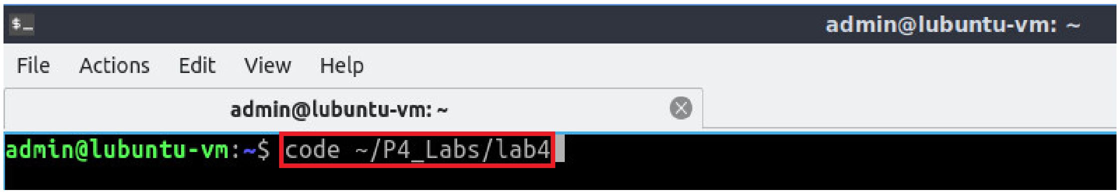


Figure 11. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). A CLI is a program that takes commands from the keyboard and sends them to the operating system to perform.

Step 2. In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab4
```

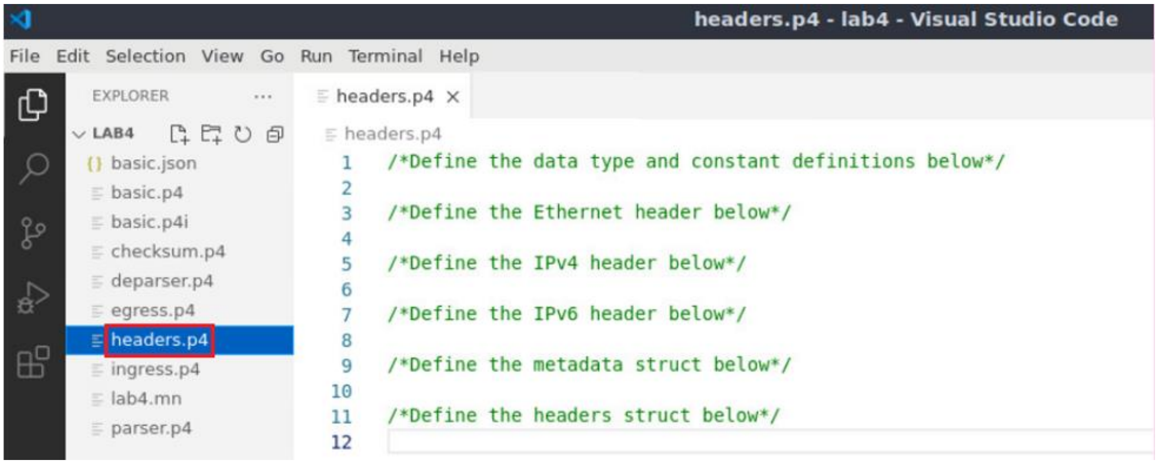


```
admin@lubuntu-vm:~$ code ~/P4_Labs/lab4
```

Figure 12. Launching the editor and opening the lab4 directory.

3.2 Coding header’s definitions into the *headers.p4* file

Step 1. Click on the *headers.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.



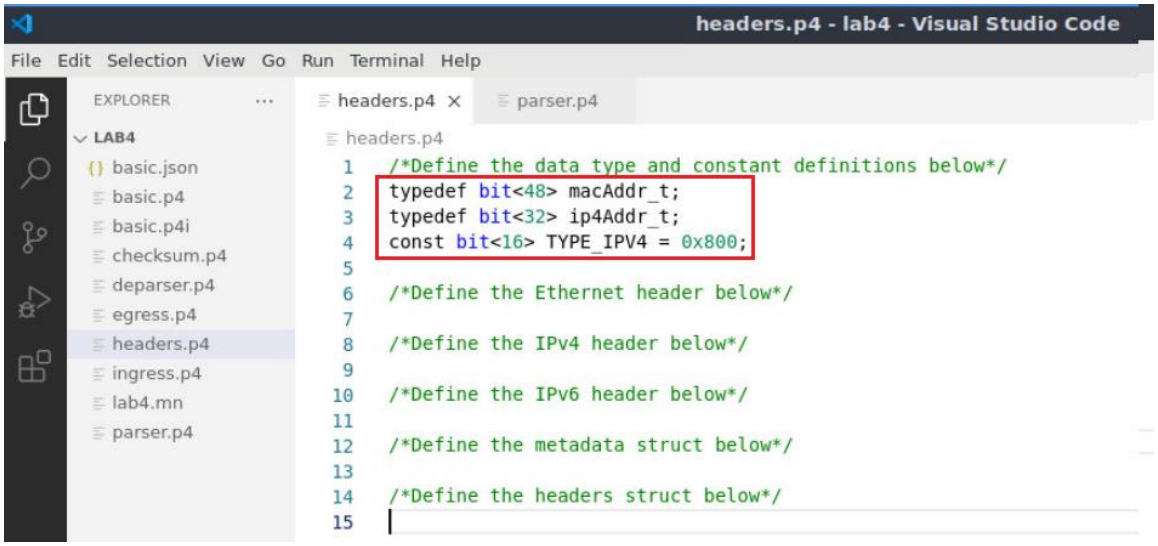
```
/*Define the data type and constant definitions below*/

/*Define the Ethernet header below*/

/*Define the IPv4 header below*/

/*Define the IPv6 header below*/

/*Define the metadata struct below*/

/*Define the headers struct below*/
```

Figure 13. Inspecting the *headers.p4* file.

We can see that the *headers.p4* is empty and we have to fill it.

Step 2. We will start by defining some typedefs and constants. Write the following in the *headers.p4* file.

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;
```



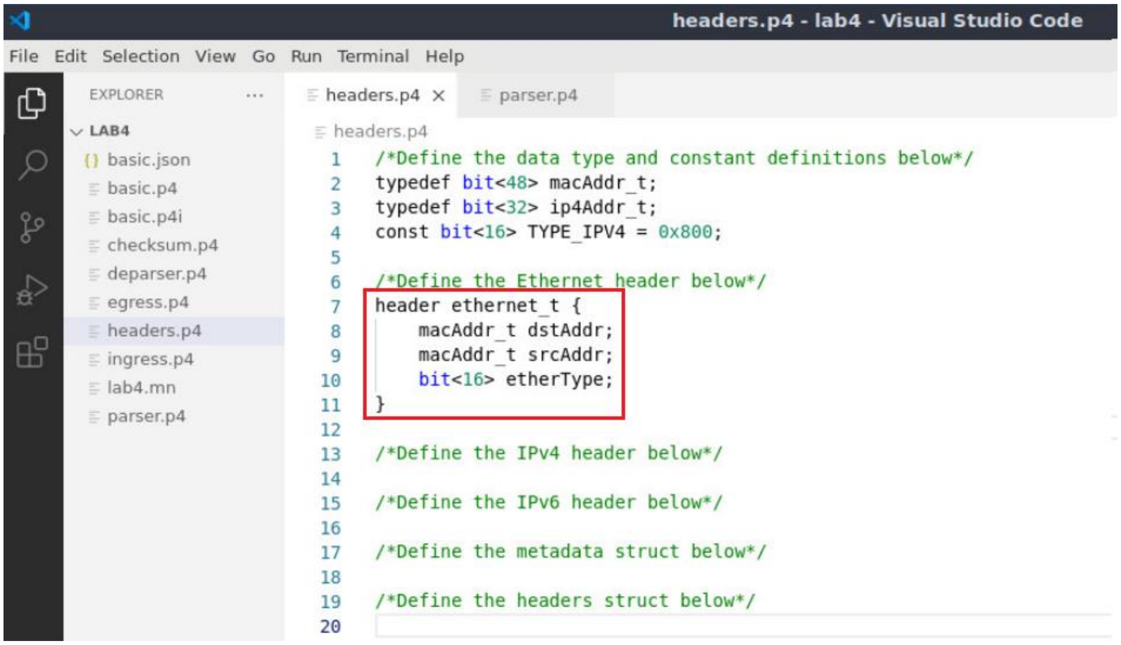
```
1  /*Define the data type and constant definitions below*/
2  typedef bit<48> macAddr_t;
3  typedef bit<32> ip4Addr_t;
4  const bit<16> TYPE_IPV4 = 0x800;
5
6  /*Define the Ethernet header below*/
7
8  /*Define the IPv4 header below*/
9
10 /*Define the IPv6 header below*/
11
12 /*Define the metadata struct below*/
13
14 /*Define the headers struct below*/
15
```

Figure 14. Data types and constant definitions.

In the figure above the typedef declarations used (lines 2 - 3) are used to assign alternative names to types. Here we are saying that `macAddr_t` can be used instead of `bit<48>`, and `ip4Addr_t` instead of `bit<32>`. We will use those typedefs when defining the headers. Line 4 shows how to define a constant with the name `TYPE_IPV4` and a value of `0x800`. We will use this value in the parser implementation.

Step 3. Now we will define the Ethernet header. Add the following code to *the headers.p4* file.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}
```



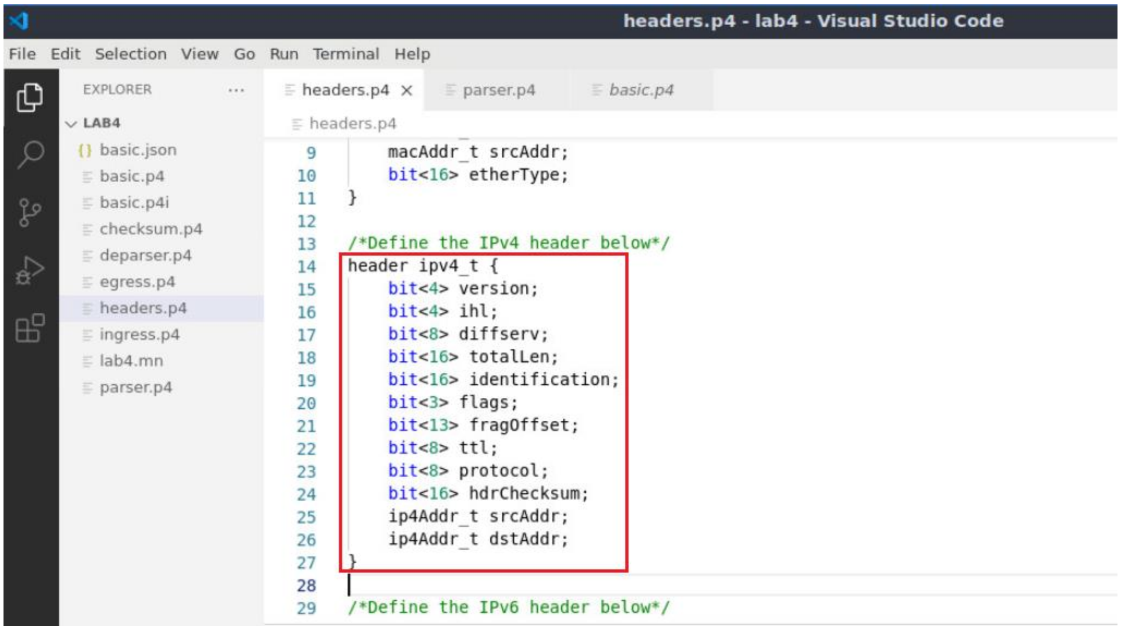
```
/*Define the data type and constant definitions below*/
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;

/*Define the Ethernet header below*/
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}

/*Define the IPv4 header below*/

/*Define the IPv6 header below*/

/*Define the metadata struct below*/

/*Define the headers struct below*/
```

Figure 15. Adding the Ethernet header definition.

Note how we used the typedef `macAddr_t` which corresponds to `bit<48>` when defining the destination MAC address field (`dstAddr`) and the source MAC address field (`srcAddr`).

Step 4. Now we will define the IPv4 header. Add the following to the *headers.p4* file.

```
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;
}
```



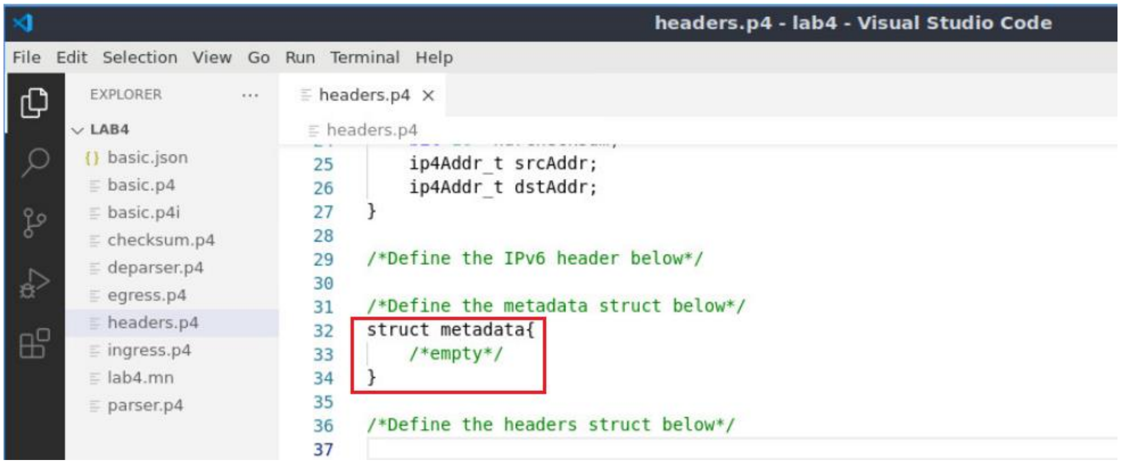
```
    macAddr_t srcAddr;
    bit<16> etherType;
}

/*Define the IPv4 header below*/
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;
}

/*Define the IPv6 header below*/
```

Figure 16. Adding the IPv4 header definition.

Consider the figure above. Note how we used the typedef `ip4Addr_t` which corresponds to `bit<32>` when defining the source IP address field (`srcAddr`) and the destination IP address field (`dstAddr`). Also, note how we are mapping the fields to those defined in the standard IPv4 header (see Figure 3).

Step 5. Now we will create a struct to represent our metadata. Metadata are passed from one block to another as the packet propagates through the architecture. For simplicity, this program does not require any user metadata, and hence we will define it as empty with no fields. Add the following to the *headers.p4* file.

```
struct metadata {
    /* empty */
}
```

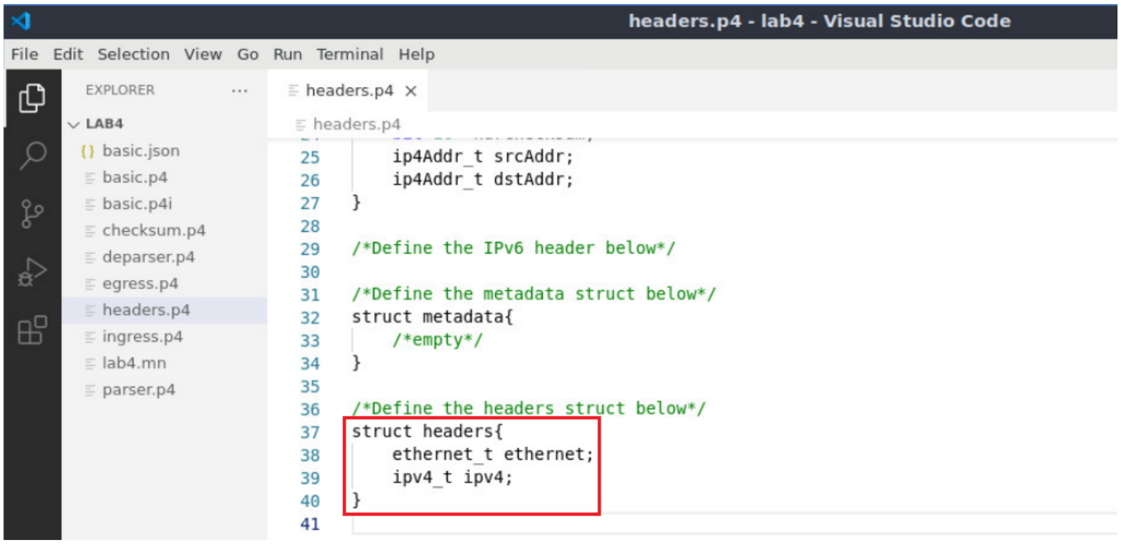


```
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/

/*Define the metadata struct below*/
struct metadata{
    /*empty*/
}

/*Define the headers struct below*/
```

Figure 17. Adding the metadata structures.

Step 6. Now we will create a struct to contain our headers (Ethernet and IPv4). Append the following code to the *headers.p4* file.

```
struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```



```
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/

/*Define the metadata struct below*/
struct metadata{
    /*empty*/
}

/*Define the headers struct below*/
struct headers{
    ethernet_t ethernet;
    ipv4_t ipv4;
}
```

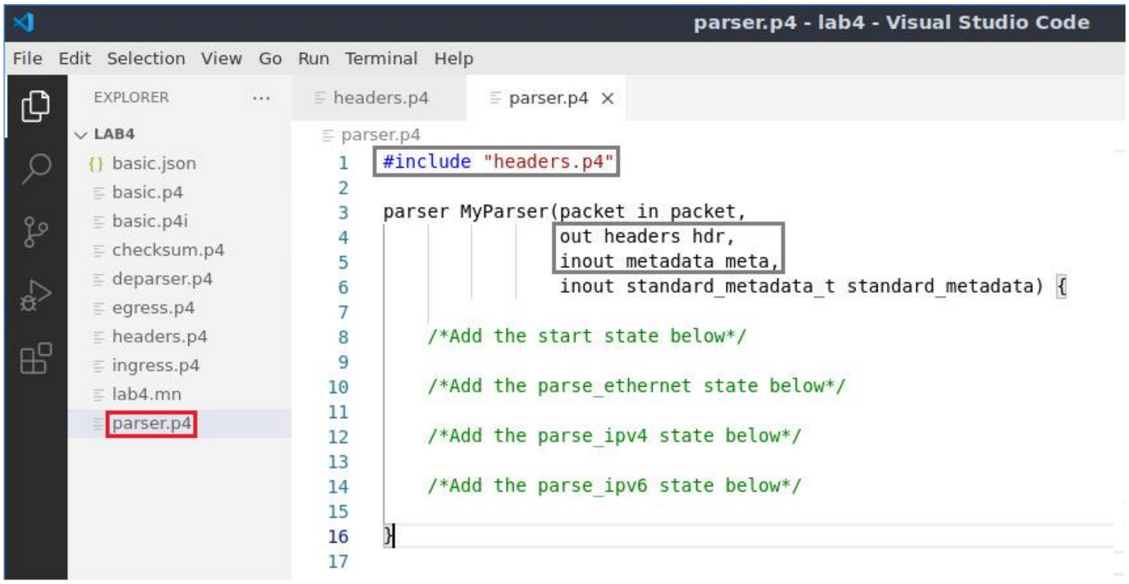
Figure 18. Appending the headers' data structure to the *headers.p4* file.

Step 7. Save the changes by pressing `Ctrl+s`.

4 Parser Implementation

Now it is time to define how the parser works.

Step 1. Click on the *parser.p4* file to display the content of the file.



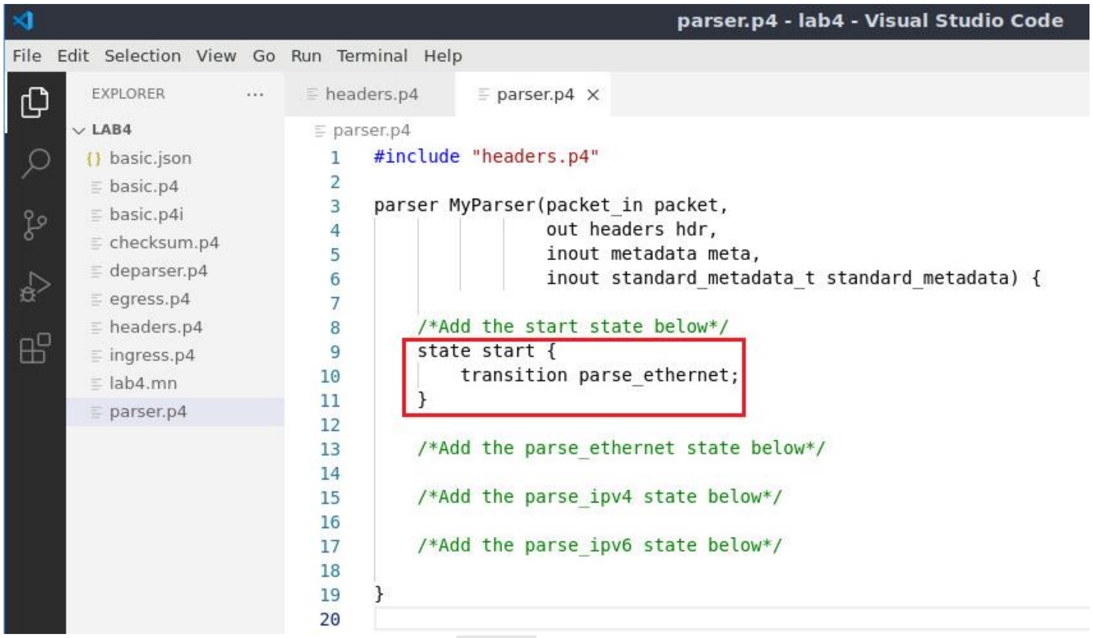
```
#include "headers.p4"

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

    /*Add the start state below*/

    /*Add the parse_ethernet state below*/

    /*Add the parse_ipv4 state below*/

    /*Add the parse_ipv6 state below*/

}
```

Figure 19. Inspecting the *parse.p4* file.

We can see that the *headers.p4* file that we just filled is included here in the parser. The file also includes a starter code which declares a parser named *MyParser*. Note how the headers and the metadata structs that we defined previously are passed as parameters to the parser.

Step 2. Add the `start` state inside the parser by inserting the following code.

```
state start {
     transition parse_ethernet;
}
```



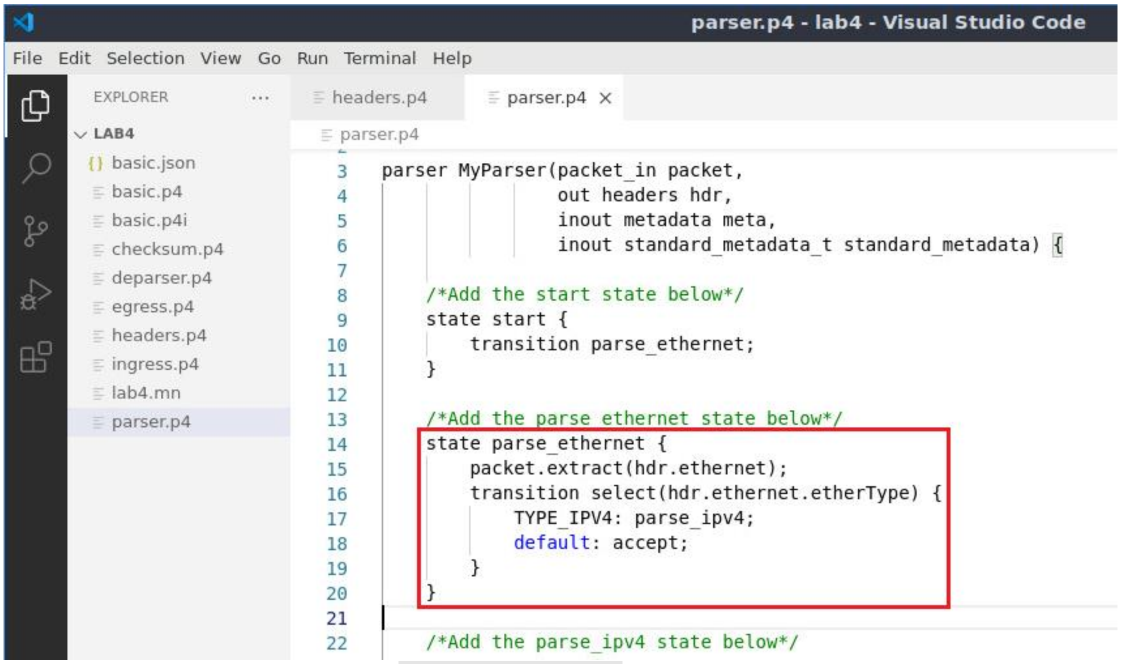
```
#include "headers.p4"

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

    /*Add the start state below*/
    state start {
        transition parse_ethernet;
    }

    /*Add the parse_ethernet state below*/

    /*Add the parse_ipv4 state below*/

    /*Add the parse_ipv6 state below*/

}
```

Figure 20. Adding `start` state to the *parser.p4* file.

The `start` state is the state where the parser begins parsing the packet. Here we are transitioning unconditionally to the `parse_ethernet` state.

Step 3. Add the `parse_ethernet` state inside the parser by inserting the following code.

```
state parse_ethernet {
   packet.extract(hdr.ethernet);
   transition select(hdr.ethernet.etherType) {
      TYPE_IPV4: parse_ipv4;
      default: accept;
   }
}
```



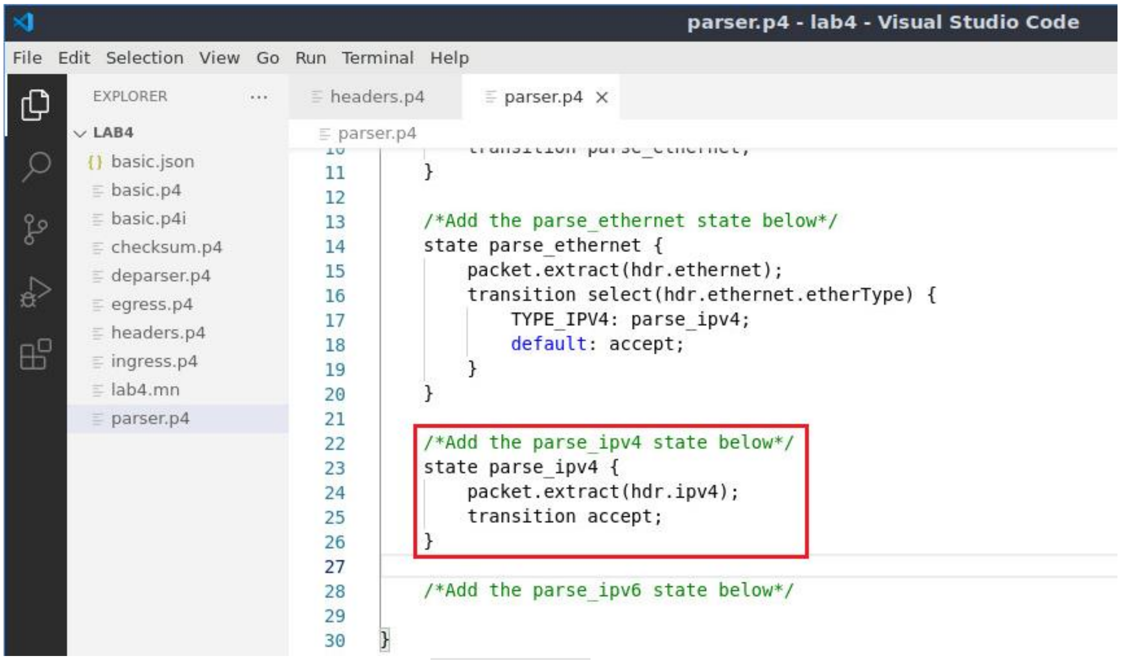
```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    /*Add the start state below*/
    state start {
        transition parse_ethernet;
    }

    /*Add the parse_ethernet state below*/
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    /*Add the parse_ipv4 state below*/
```

Figure 21. Adding `parse_ethernet` state to the *parser.p4* file.

The `parse_ethernet` state extracts the Ethernet header and checks for the value of the header field `etherType`. Note how we reference a header field by specifying the header to which that field belongs (i.e., `hdr.ethernet.etherType`). If the value of `etherType` is `TYPE_IPV4` (which corresponds to 0x800 as defined previously), the parser transitions to the `parse_ipv4` state. Otherwise, the execution of the parser terminates.

Step 4. Add the `parse_ipv4` state inside the parser by inserting the following code.

```
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```



```
        transition parse_ethernet;
    }

    /*Add the parse_ethernet state below*/
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    /*Add the parse_ipv4 state below*/
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

    /*Add the parse_ipv6 state below*/

}
```

Figure 22. Adding `parse_ipv4` state to the *parser.p4* file.

The `parse_ipv4` state extracts the IPv4 header and terminates the execution of the parser.

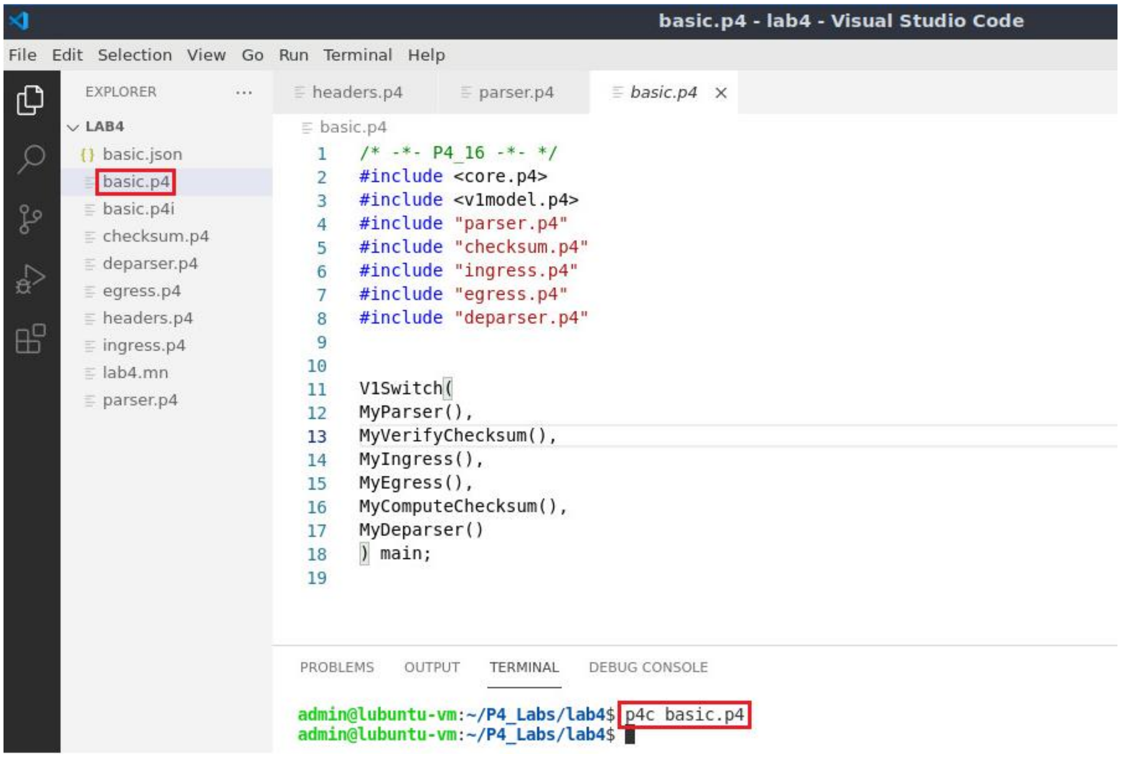
Step 5. Save the changes to the file by pressing `Ctrl + s`.

5 Loading the P4 program

5.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```



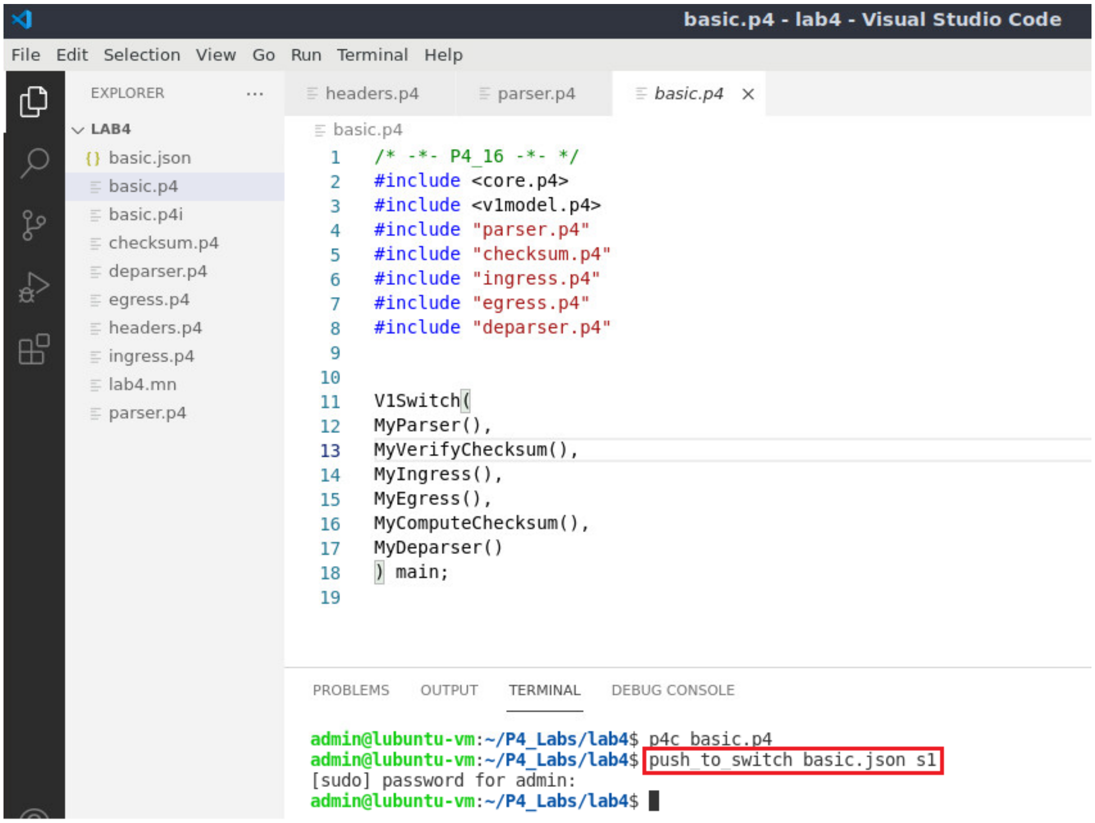
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 23. Compiling the code.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the *p4c* compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 24. Pushing the P4 program to switch s1.

5.2 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.

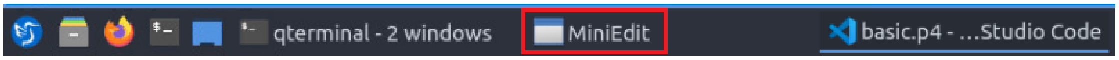


Figure 25. Maximizing the MiniEdit window.

Step 2. Right-click on the P4 switch icon in MiniEdit and start the *Terminal*.

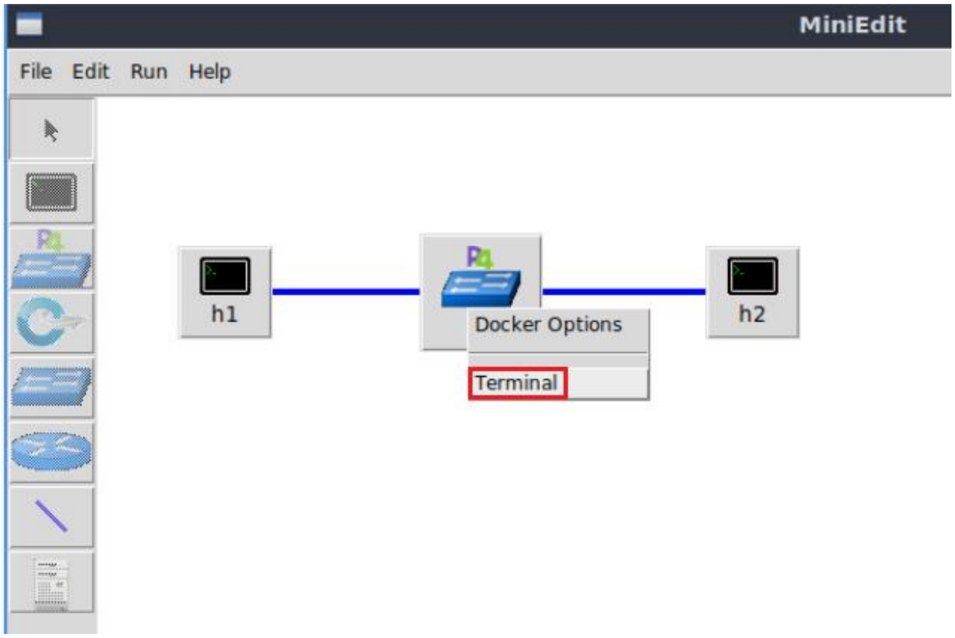
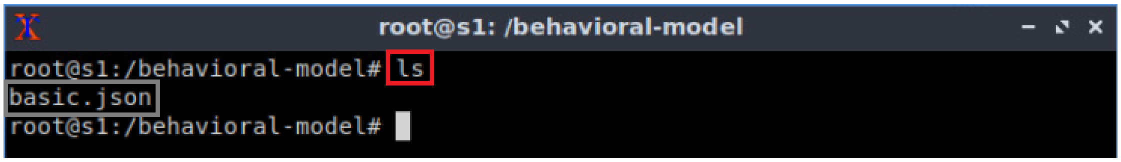


Figure 26. Starting the terminal on the switch.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

Step 3. Issue the command `ls` on the terminal of the switch s1 that was opened in the previous step.

```
ls
```



```
root@s1: /behavioral-model
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model#
```

Figure 27. Displaying the contents of the current directory in the switch s1.

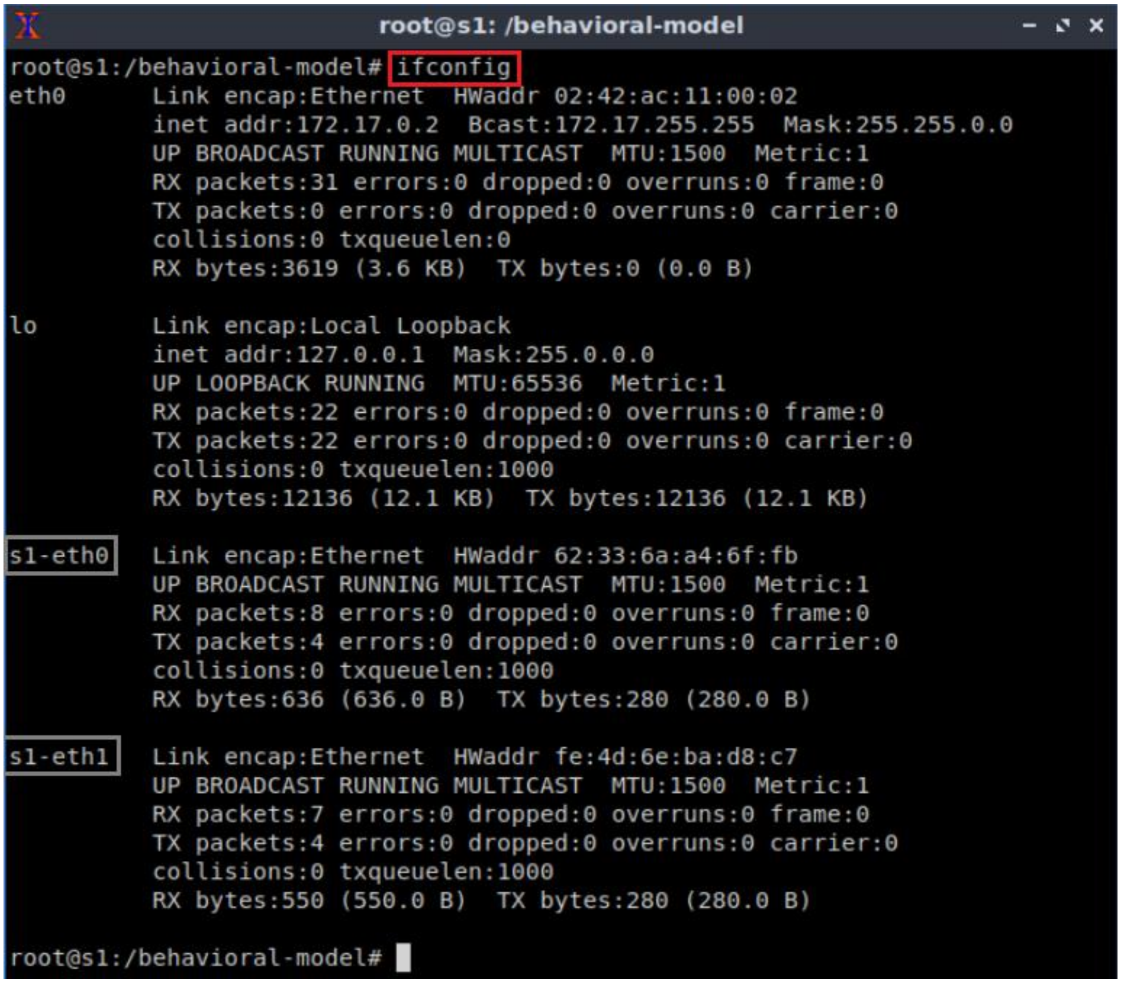
We can see that the switch contains the *basic.json* file that was pushed previously after compiling the P4 program.

6 Configuring switch s1

6.1 Mapping P4 program’s ports

Step 1. Issue the following command on switch s1 terminal to display the interfaces.

```
ifconfig
```



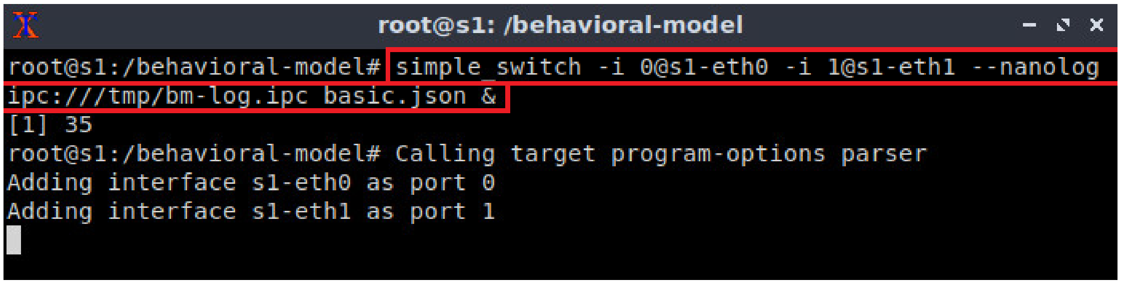
```
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3619 (3.6 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12136 (12.1 KB)  TX bytes:12136 (12.1 KB)

s1-eth0   Link encap:Ethernet  HWaddr 62:33:6a:a4:6f:fb
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:636 (636.0 B)  TX bytes:280 (280.0 B)

s1-eth1   Link encap:Ethernet  HWaddr fe:4d:6e:ba:d8:c7
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:550 (550.0 B)  TX bytes:280 (280.0 B)

root@s1:/behavioral-model# 
```

Figure 28. Displaying switch s1 interfaces.

We can see that the switch has the interfaces *s1-eth0* and *s1-eth1*. The interface *s1-eth0* on the switch s1 connects host h1. The interface *s1-eth1* on the switch s1 connects host h2.

Step 2. Start the switch daemon by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog ipc:///tmp/bm-log.ipc
basic.json &
```



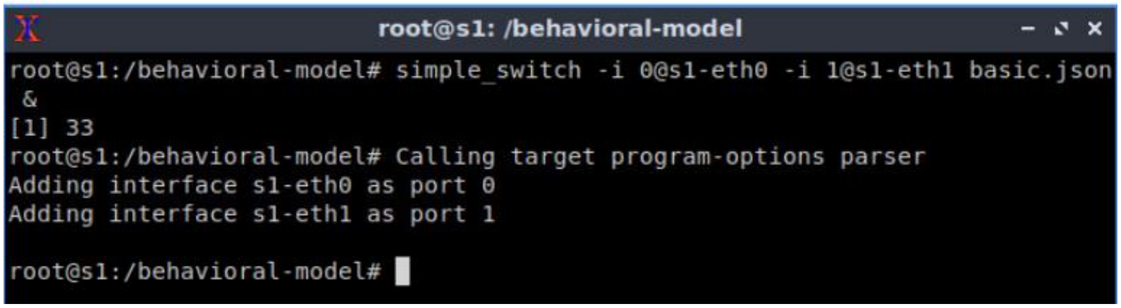
```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 35
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 29. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

The `--nanolog` parameter is used to instruct the switch daemon that we want to see the logs of the switch.

6.2 Loading the rules to the switch

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

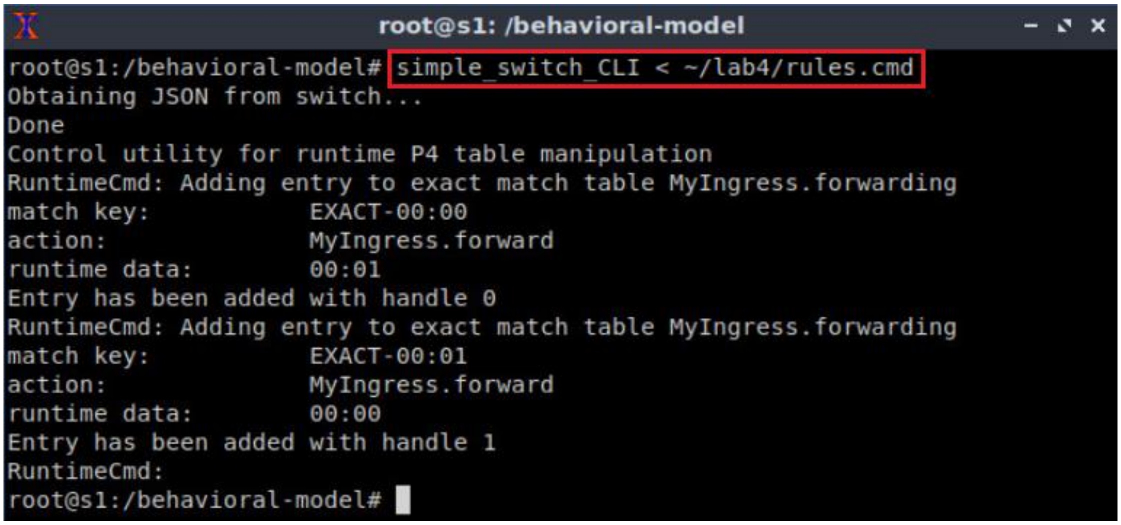


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 30. Returning to switch s1 CLI.

Step 2. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab4/rules.cmd
```



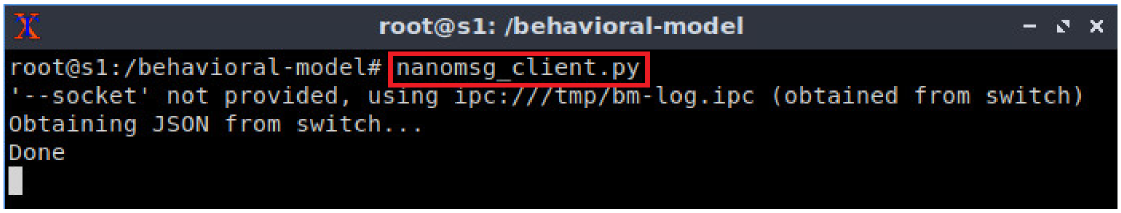
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab4/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# 
```

Figure 31. Populating the forwarding table into switch s1.

7 Testing and verifying the P4 program

Step 1. Type the following command to initiate the `nanolog` client that will display the switch logs.

```
nanomsg_client.py
```

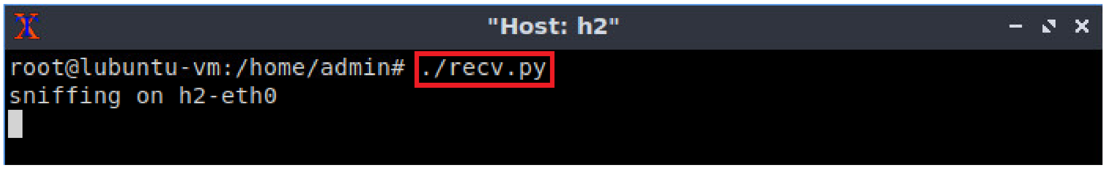


```
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 32. Displaying switch s1 logs.

Step 2. On host h2's terminal, type the command below so that the host starts listening for packets.

```
./recv.py
```

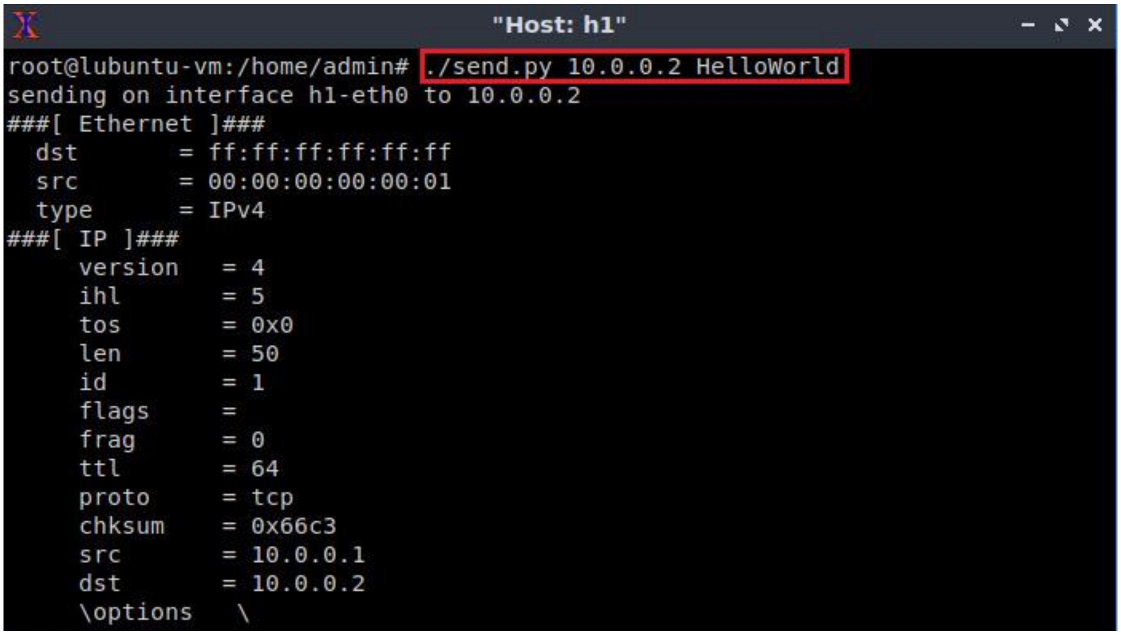


```
root@lubuntu-vm:/home/admin# ./recv.py
sniffing on h2-eth0
```

Figure 33. Listening for incoming packets in host h2.

Step 3. On host h1's terminal, type the following command to send a packet to host h2.

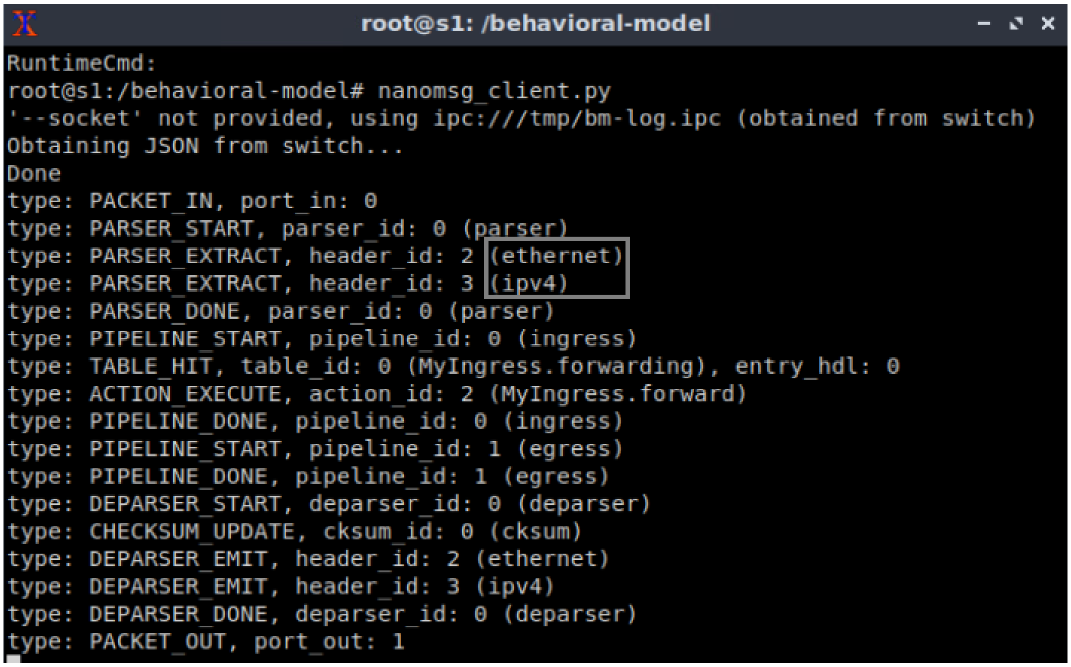
```
./send.py 10.0.0.2 HelloWorld
```



```
root@lubuntu-vm:/home/admin# ./send.py 10.0.0.2 HelloWorld
sending on interface h1-eth0 to 10.0.0.2
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 50
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x66c3
     src       = 10.0.0.1
     dst       = 10.0.0.2
     \options   \
```

Figure 34. Sending a test packet from host h1 to host h2.

Step 4. Inspect the logs on switch s1 terminal.



```
RuntimeCmd:
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 35. Inspecting the logs in switch s1.

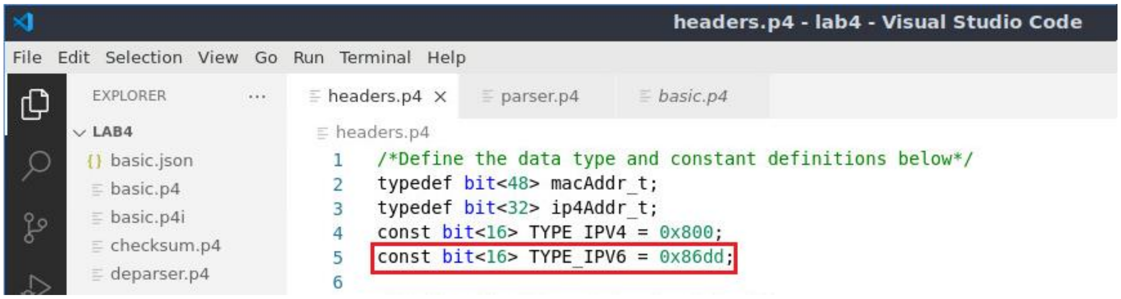
The figure above shows that the Ethernet and IPv4 header are extracted.

8 Augmenting the P4 program to parse IPv6

Now we will augment the program to parse IPv6 packets. Figure 4 shows the IPv6 header fields.

Step 1. Go back to the *headers.p4* file and add the following constant definition.

```
const bit<16> TYPE_IPV6 = 0x86dd;
```



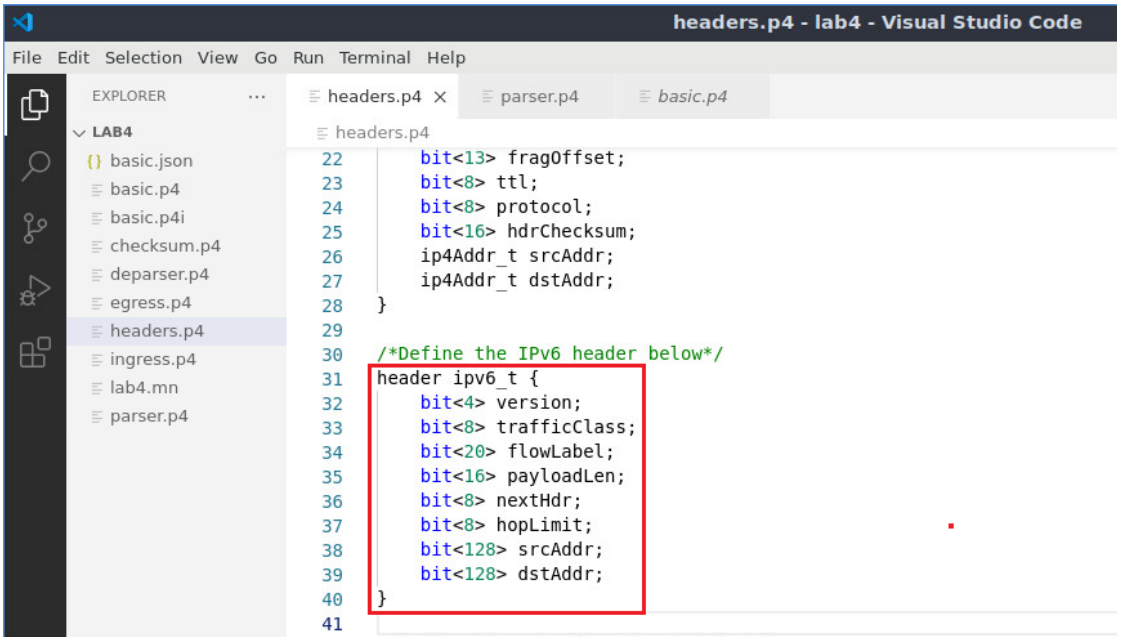
```
/*Define the data type and constant definitions below*/
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
const bit<16> TYPE_IPV4 = 0x800;
const bit<16> TYPE_IPV6 = 0x86dd;
```

Figure 36. Adding the IPv6 type definition.

Step 2. Add the IPv6 header definition as shown below.

```
header ipv6_t{
       bit<4> version;
       bit<8> trafficClass;
       bit<20> flowLabel;
       bit<16> payloadLen;
       bit<8> nextHdr;
       bit<8> hopLimit;
```

```
        bit<128> srcAddr;
        bit<128> dstAddr;
}
```

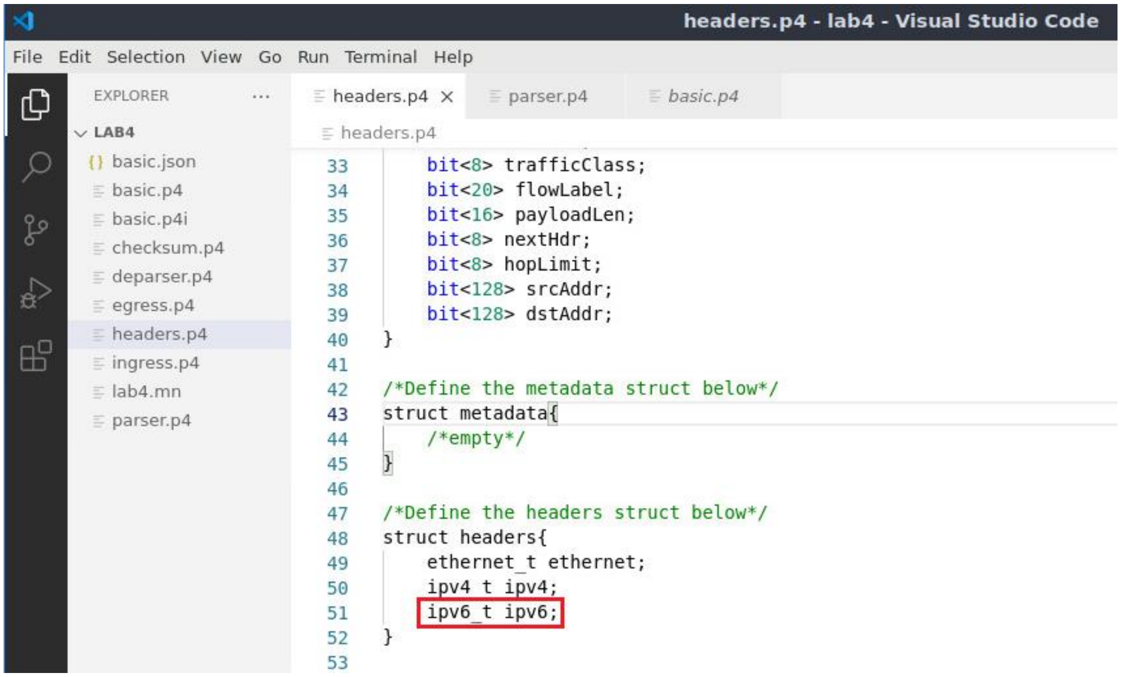


```
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

/*Define the IPv6 header below*/
header ipv6_t {
    bit<4> version;
    bit<8> trafficClass;
    bit<20> flowLabel;
    bit<16> payloadLen;
    bit<8> nextHdr;
    bit<8> hopLimit;
    bit<128> srcAddr;
    bit<128> dstAddr;
}
```

Figure 37. Adding the IPv6 header definition.

Step 3. Append the IPv6 header to the header's data structure.

```
ipv6_t ipv6;
```

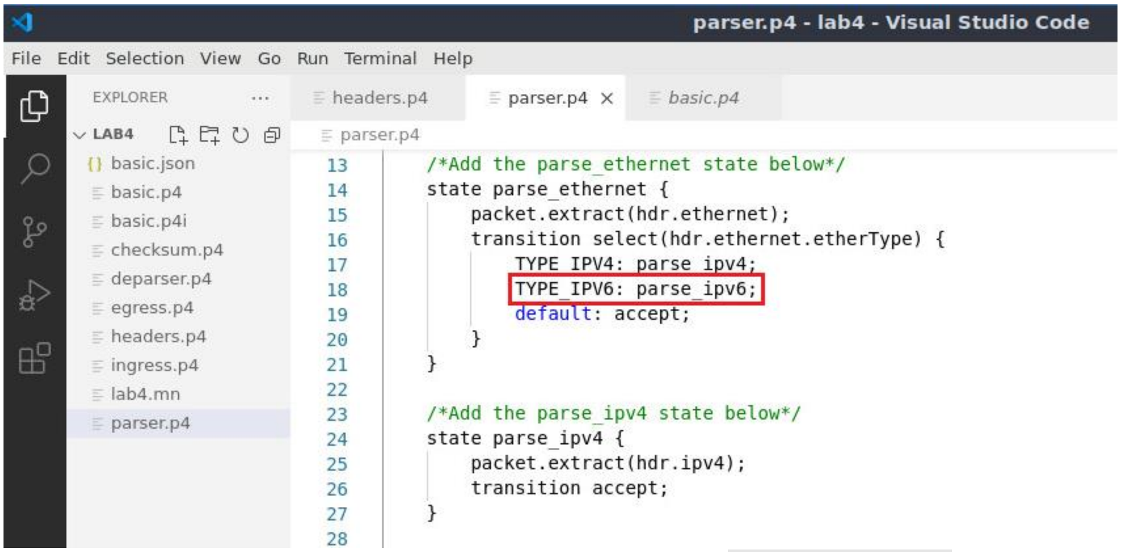


```
    bit<8> trafficClass;
    bit<20> flowLabel;
    bit<16> payloadLen;
    bit<8> nextHdr;
    bit<8> hopLimit;
    bit<128> srcAddr;
    bit<128> dstAddr;
}

/*Define the metadata struct below*/
struct metadata{
    /*empty*/
}

/*Define the headers struct below*/
struct headers{
    ethernet_t ethernet;
    ipv4_t ipv4;
    ipv6_t ipv6;
}
```

Figure 38. Adding IPv6 type to the header data structure.

Step 4. Go to the *parser.p4* file and add the following line to the `parse_ethernet` state.

```
TYPE_IPV6: parse_ipv6;
```

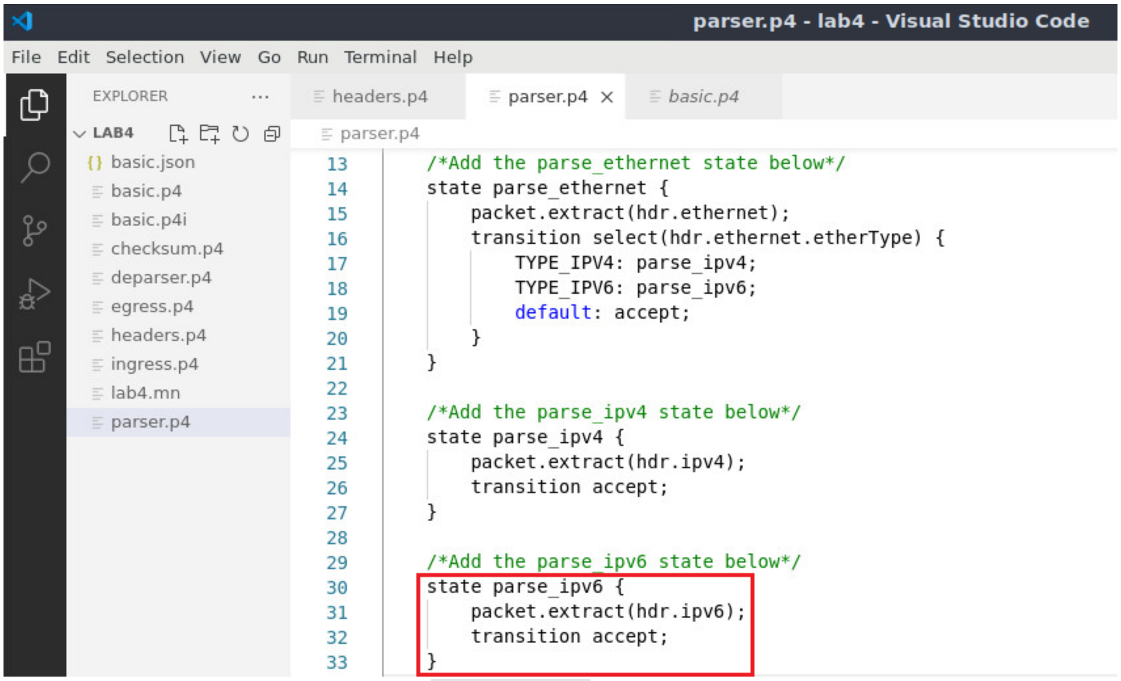


```
/*Add the parse_ethernet state below*/
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        TYPE_IPV6: parse_ipv6;
        default: accept;
    }
}

/*Add the parse_ipv4 state below*/
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```

Figure 39. Including the IPv6 state transition into the `parse_ethernet` state.

Step 5. Add the `parse_ipv6` state inside the parser by inserting the following code.

```
state parse_ipv6 {
       packet.extract(hdr.ipv6);
       transition accept;
}
```



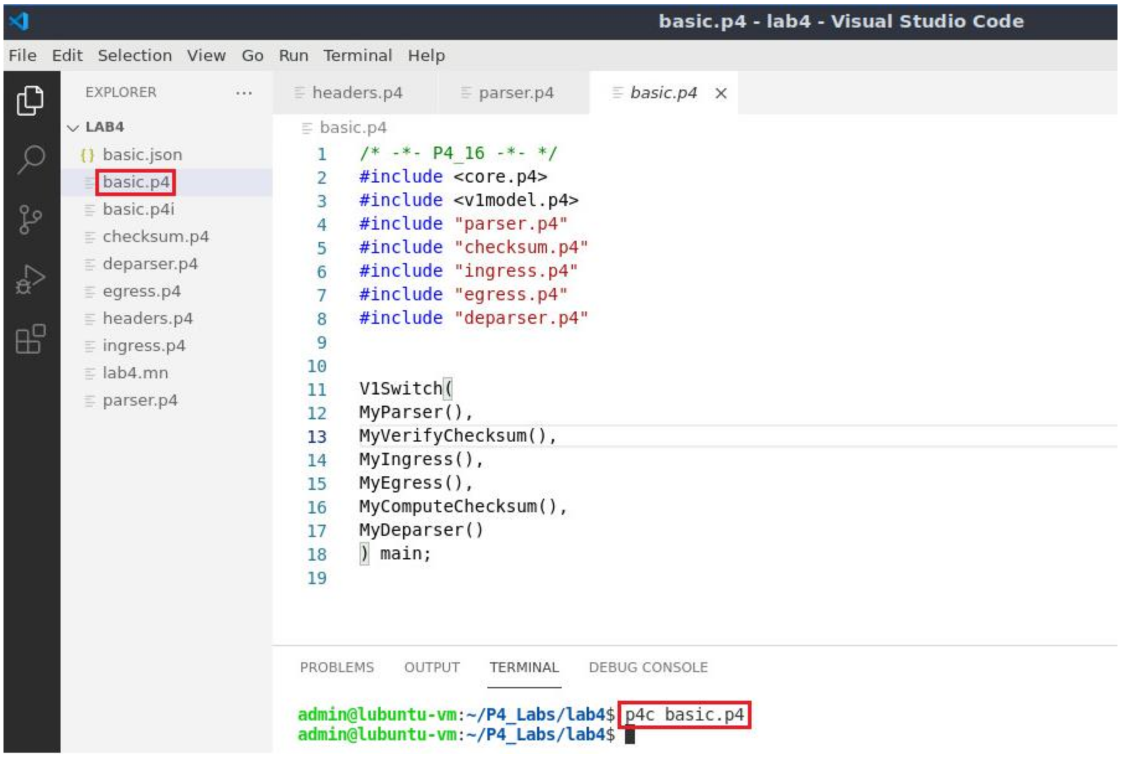
```
/*Add the parse_ethernet state below*/
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        TYPE_IPV6: parse_ipv6;
        default: accept;
    }
}

/*Add the parse_ipv4 state below*/
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}

/*Add the parse_ipv6 state below*/
state parse_ipv6 {
    packet.extract(hdr.ipv6);
    transition accept;
}
```

Figure 40. Adding `parse_ipv6` state to the *parser.p4* file.

Step 6. Save the changes by pressing `Ctrl+s`.

Step 7. Issue the following command in the terminal panel inside the Visual Studio Code to compile the program.

```
p4c basic.p4
```



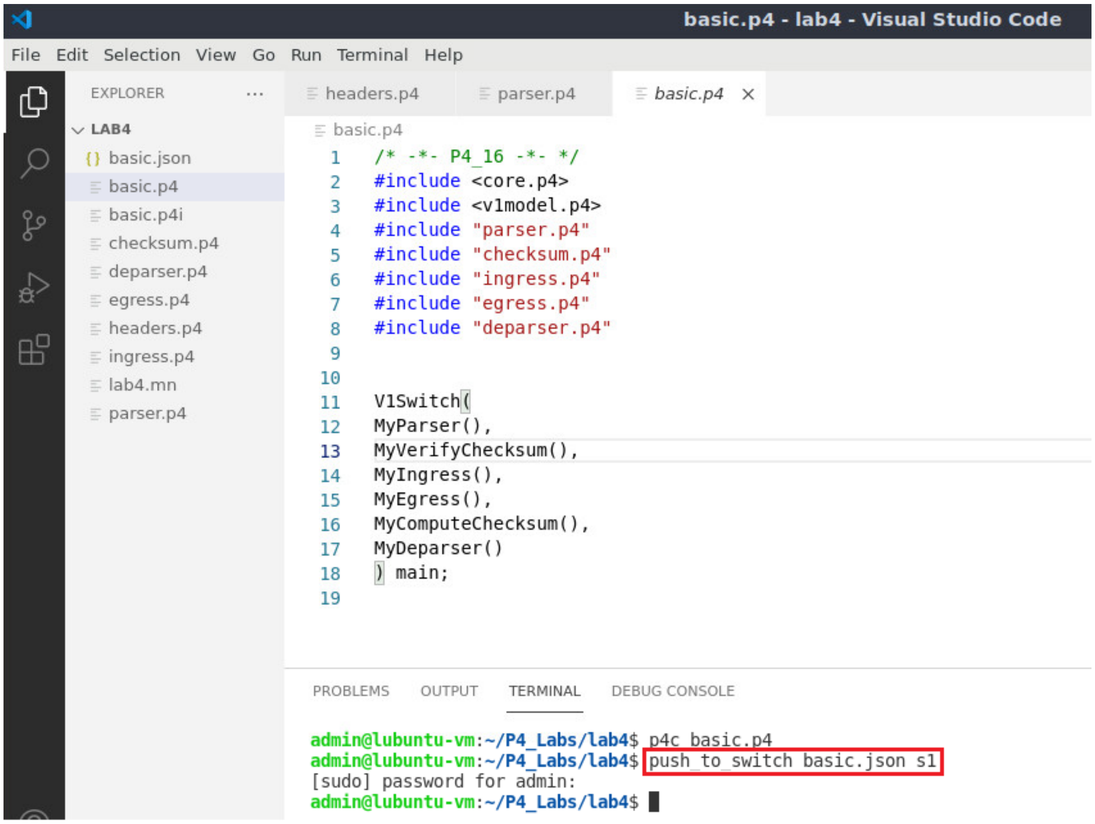
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$ 
```

Figure 41. Compiling the P4 program.

Step 8. Type the command below in the terminal panel to push the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the p4c compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



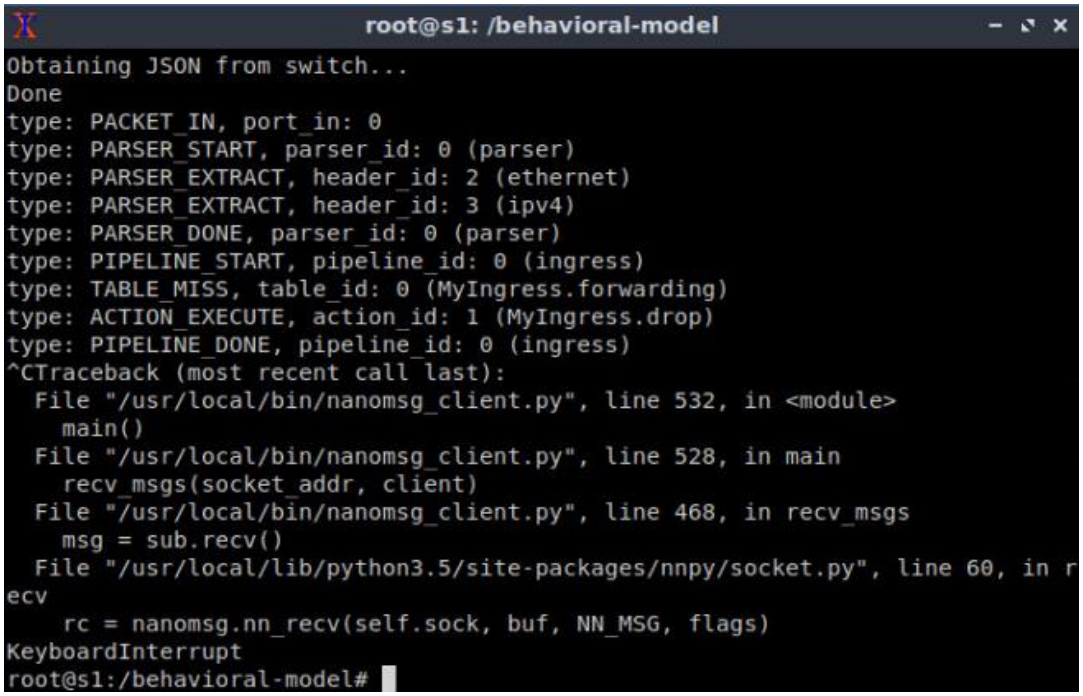
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"


V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab4$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab4$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab4$
```

Figure 42. Pushing the P4 program to switch s1.

9 Testing and verifying the augmented P4 program

Step 1. In switch s1 terminal, press `Ctrl + c` to return to the CLI. The figure below shows the output after executing the command.

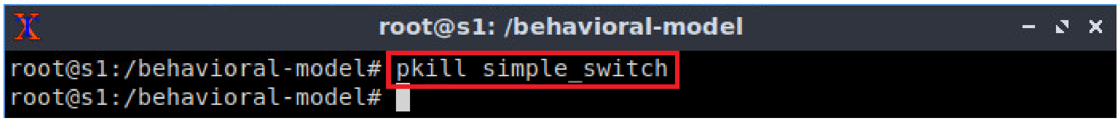


```
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_MISS, table_id: 0 (MyIngress.forwarding)
type: ACTION_EXECUTE, action_id: 1 (MyIngress.drop)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
^CTraceback (most recent call last):
  File "/usr/local/bin/nanomsg_client.py", line 532, in <module>
    main()
  File "/usr/local/bin/nanomsg_client.py", line 528, in main
    recv_msgs(socket_addr, client)
  File "/usr/local/bin/nanomsg_client.py", line 468, in recv_msgs
    msg = sub.recv()
  File "/usr/local/lib/python3.5/site-packages/nnpy/socket.py", line 60, in r
ecv
    rc = nanomsg.nn_recv(self.sock, buf, NN_MSG, flags)
KeyboardInterrupt
root@s1:/behavioral-model#
```

Figure 43. Returning to the CLI.

Step 2. Type the command below in the terminal of switch s1 to stop the running daemon.

```
pkill simple_switch
```

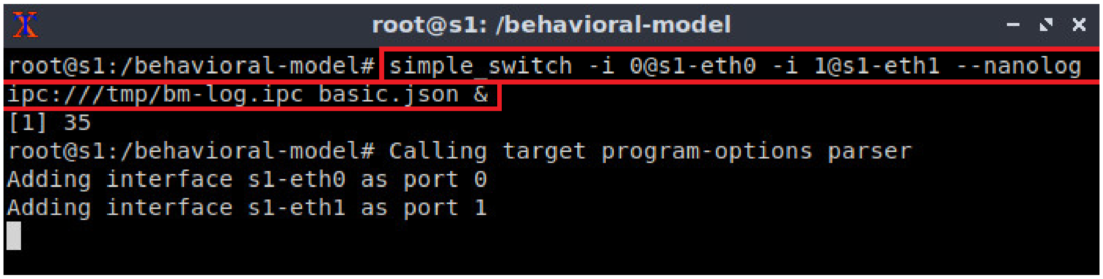


```
root@s1:/behavioral-model# pkill simple_switch
root@s1:/behavioral-model#
```

Figure 44. Ending switch s1 P4 process.

Step 3. Type the command below in the terminal of the switch s1 to start the daemon with the new P4 program.

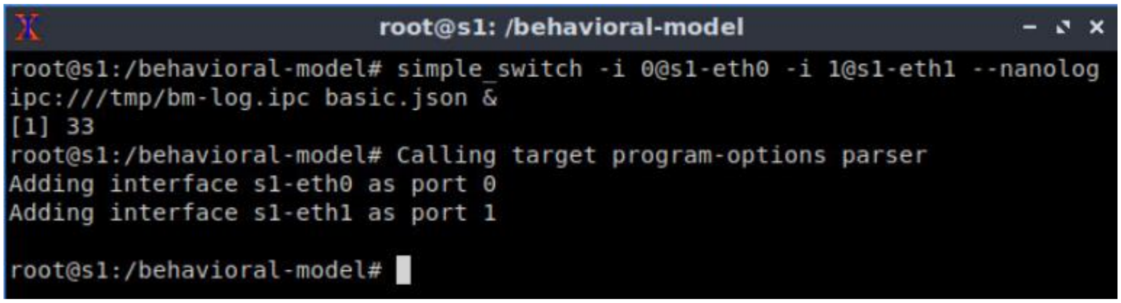
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog ipc:///tmp/bm-log.ipc
basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 35
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 45. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 4. In switch s1 terminal, press *Enter* to return the CLI.

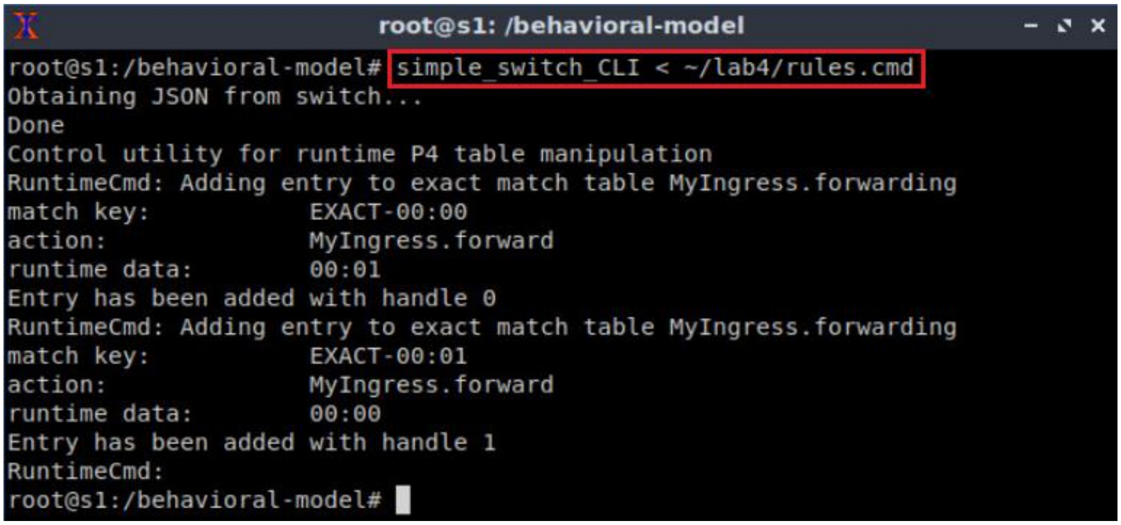


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 --nanolog
ipc:///tmp/bm-log.ipc basic.json &
[1] 33
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model#
```

Figure 46. Returning to switch s1 CLI.

Step 5. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab4/rules.cmd
```

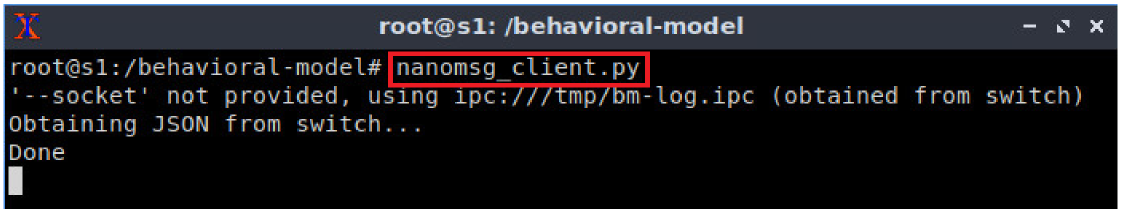


```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch_CLI < ~/lab4/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 47. Populating the forwarding table into switch s1.

Step 6. Type the following command to display the switch logs.

```
nanomsg_client.py
```

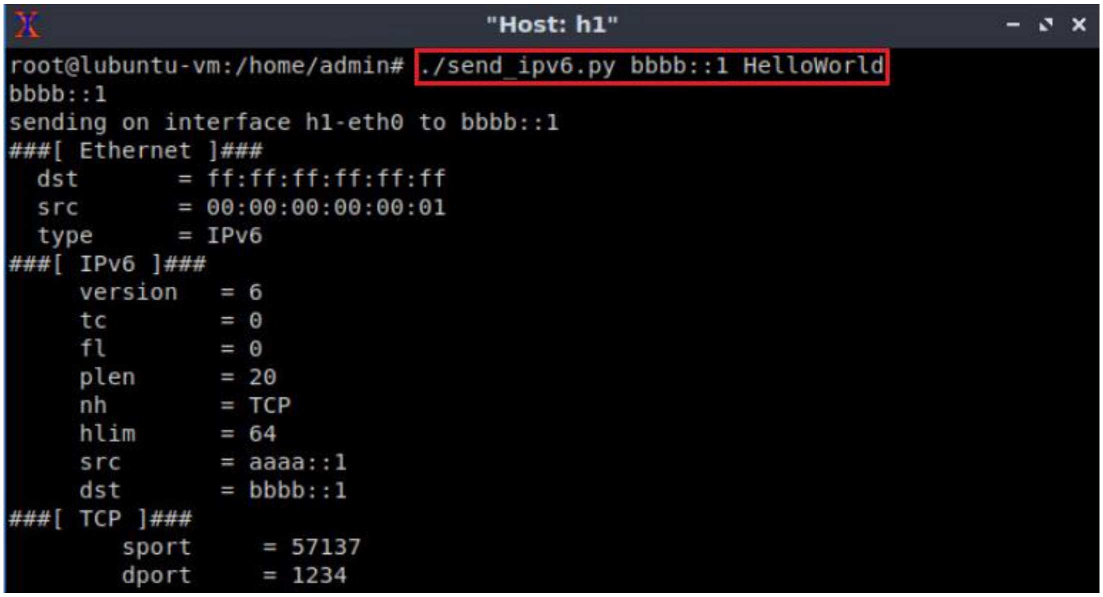


```
root@s1: /behavioral-model
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 48. Inspecting the logs in switch s1.

Step 7. On host h1's terminal, type the following command to send an IPv6 packet to host h2. Note that `bbbb::1` is IPv6 address of host h2.

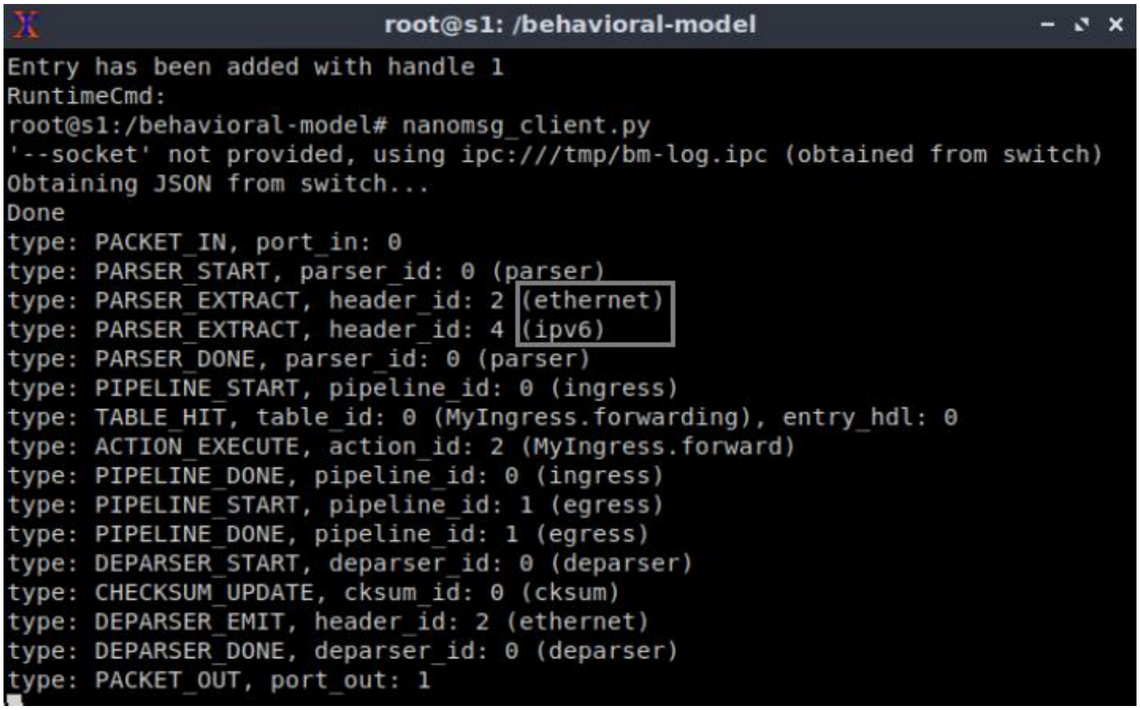
```
./send_ipv6.py bbbb::1 HelloWorld
```



```
"Host: h1"
root@lubuntu-vm:/home/admin# ./send_ipv6.py bbbb::1 HelloWorld
bbbb::1
sending on interface h1-eth0 to bbbb::1
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv6
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 0
     plen      = 20
     nh        = TCP
     hlim      = 64
     src       = aaaa::1
     dst       = bbbb::1
###[ TCP ]###
        sport     = 57137
        dport     = 1234
```

Figure 49. Sending an IPv6 test packet from host h1 to host h2.

Step 8. Go back to switch s1 and inspect the logs.



```
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 4 (ipv6)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: TABLE_HIT, table_id: 0 (MyIngress.forwarding), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 2 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 50. Inspecting the logs in switch s1.

The figure above shows that the Ethernet and IPv6 header are extracted.

This concludes lab 4. Stop the emulation and then exit out of MiniEdit.

References

1. C. Cascaval, D. Daly. "P4 Architectures." [Online]. Available: https://tinyurl.com/3zk8vs6a.
2. “p4c core.p4”. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/core.p4.
3. “p4c v1model.p4”. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.
4. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 5: Introduction to Match-action Tables

Document Version: 03-30-2023

Contents

Overview

This lab describes match-action tables and how to define them in a P4 program. It then explains the different types of matching that can be performed on keys. The lab further shows how to track the misses/hits of a table key while a packet is received on the switch.

Objectives

By the end of this lab, students should be able to:

1. Understand what match-action tables are used for.
2. Describe the basic syntax of a match-action table.
3. Implement a simple table in a P4.
4. Trace a table's misses/hits when a packet enters to the switch.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Defining a table with exact match lookup.
4. Section 4: Defining a table with LPM matching
5. Section 5: Loading the P4 program.
6. Section 6: Configuring switch s1.
7. Section 7: Testing and verifying the P4 program.

1 Introduction to control blocks

Control blocks are essential for processing a packet. For example, a control block for layer-3 forwarding may require a forwarding table that is indexed by the destination IP address. The control block may include actions to forward a packet when a hit occurs, and to drop the packet otherwise. To forward a packet, a switch must perform routing lookup on the destination IP address. Figure 1 shows the basic structure of a control block.

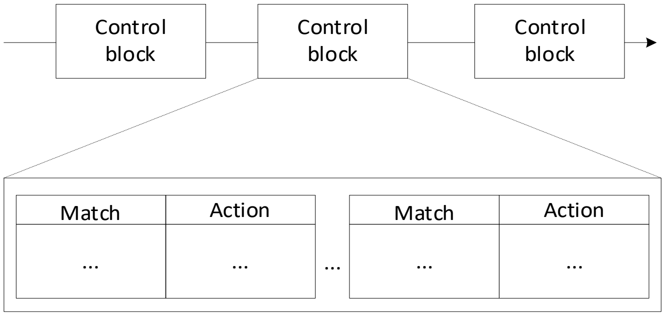


Figure 1. Control blocks.

1.1 Tables

Tables are essential components that define the processing behavior of a packet inside the switch. A table is specified in the P4 program and has one or more entries (rows) which are populated by the control plane. An entry contains a key, an action, and action data.

- Key: it is used for lookup operations. The switch builds a key for the incoming packet using one or more header fields (e.g., destination IP address) and then lookups for that value in the table.
- Action: once a match occurs, the action specified in the entry is performed by the arithmetic logic unit. Actions are simple operations such as modify a header field, forward the packet to an egress port, and drop the packet. The P4 program contains the possible actions.
- Action data: it can be considered as parameter/s used along with the action. For example, the action data may represent the port number the switch must use to forward the packet. Action data is populated by the control plane.

1.2 Match types

There are three types of matching: exact match, Longest Prefix match (LPM), and ternary match. They are defined in the standard library (*core.p4*[1]). Note that architectures may define and implement additional match types. For example, the V1Model[2] also has matching based on ranges and selectors. In this lab we will discuss exact match.

1.3 Exact match

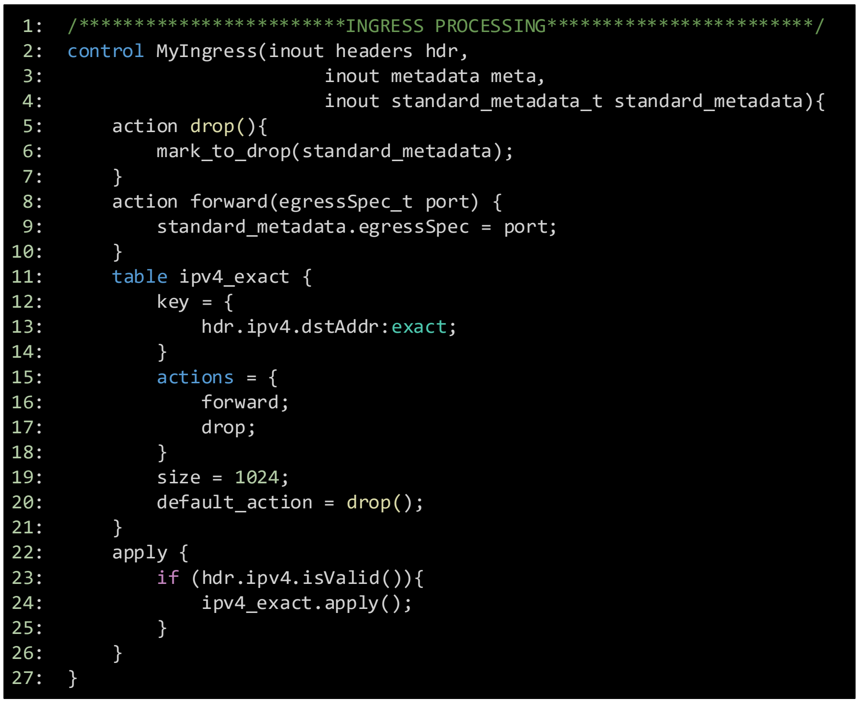
Assume that the exact match lookup is used to search for a specific value of an entry in a table. Assume that Table 2 matches on the destination IP address. If an incoming packet has 10.0.0.2 as the destination IP address, then it will match against the second entry and the P4 program will forward the packet using port 2 as the egress port.

Table 2. Exact match table.

Key	Action	Action data
10.0.0.1	forward	port 1
10.0.0.2	forward	port 2
default	drop	

Figure 2 shows the ingress control block portion of a P4 program. Two actions are defined, `drop` and `forward`. The `drop` action (lines 5 - 7) invokes the `mark_to_drop` primitive, causing the packet to be dropped at the end of the ingress processing. The `forward` action (lines 8 - 10) accepts as input (i.e., action data) the destination port. This parameter is inserted by the control plane and updated in the packet during the ingress processing. In line 9, the P4 program assigns the egress port defined by the control plane to the `standard_metadata` egress specification field (i.e., the field that the traffic manager looks at to determine which port the packet will be sent to). Lines 11-21 implement a table named `ipv4_exact`. The match is against the destination IP address using the exact lookup method. The actions associated with the table are forward and drop. The default action which is invoked when there is a miss is drop. The maximum number of entries a table can support is configured manually by the programmer (i.e., 1024 entries, see line 19). Note, however, that the number of entries is limited by the amount of memory in the switch.

The control block starts executing from the apply statement (see lines 22-26) which contains the control logic. In this program, the `ipv4_exact` table is enabled when the incoming packet has a valid IPv4 header.



```
 1:  /************************INGRESS PROCESSING************************/
 2:  control MyIngress(inout headers hdr,
 3:                    inout metadata meta,
 4:                    inout standard_metadata_t standard_metadata){
 5:      action drop(){
 6:          mark_to_drop(standard_metadata);
 7:      }
 8:      action forward(egressSpec_t port) {
 9:          standard_metadata.egressSpec = port;
10:      }
11:      table ipv4_exact {
12:          key = {
13:              hdr.ipv4.dstAddr:exact;
14:          }
15:          actions = {
16:              forward;
17:              drop;
18:          }
19:          size = 1024;
20:          default_action = drop();
21:      }
22:      apply {
23:          if (hdr.ipv4.isValid()){
24:              ipv4_exact.apply();
25:          }
26:      }
27:  }
```

Figure 2. Ingress control block portion of a P4 program. The code implements a match-action table with exact match lookup.

1.4 Longest prefix match (LPM)

Table 2 is an example of a match-action table that uses LPM. Assume that the key is formed with the destination IP address. If an incoming packet has the destination IP address 172.168.3.5, two entries match. The first entry matches because the first 29 bits in the entry are the same as the first 29 bits of the destination IP. The second entry also matches because the first 16 bits in the entry are the same as the first 16 bits of the destination IP. The LPM algorithm will select 172.168.3.0/29 because of the longest prefix preference.

Table 2. Match-action table using LPM as the lookup algorithm.

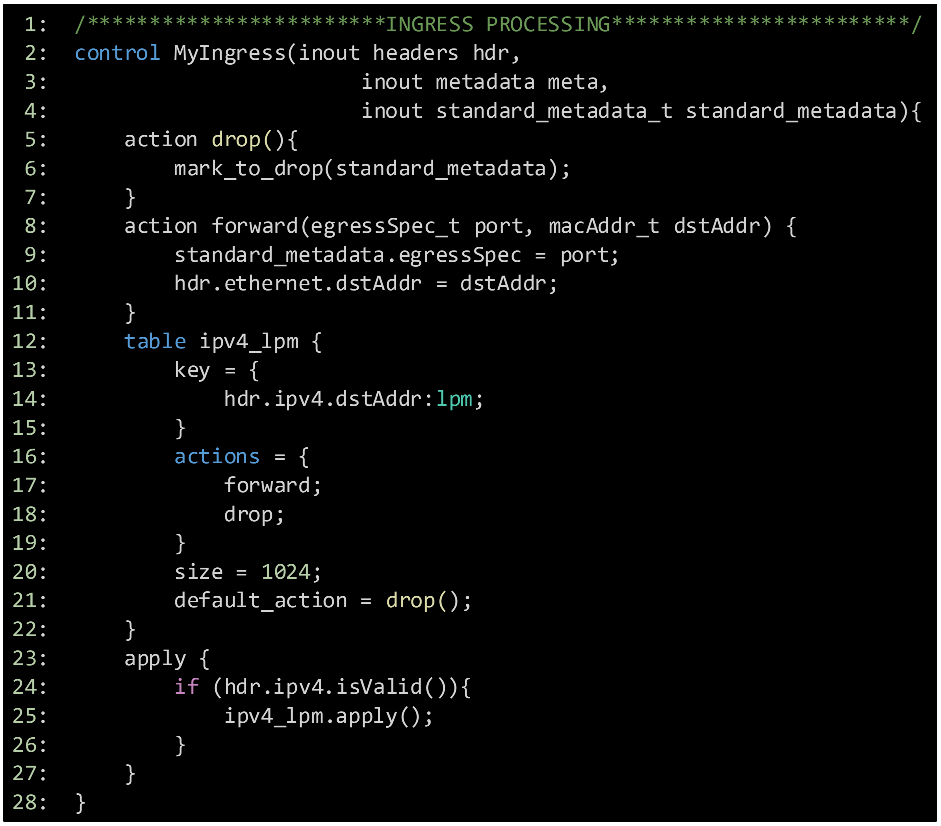
Key	Action	Action data
172.168.3.0/29	forward	port 1, macAddr=00:00:00:00:00:01
172.168.0.0/16	forward	port 2, macAddr=00:00:00:00:00:02
default	drop	

Figure 3 shows the ingress control block portion of a P4 program. Two actions are defined, `drop` and `forward`. The `drop` action (lines 5 - 7) invokes the `mark_to_drop` primitive, causing the packet to be dropped at the end of the ingress processing. The `forward` action (lines 8 - 11) accepts as input (action data) the port and the destination MAC address. These parameters are inserted by the control plane and updated in the packet during the ingress processing.

In line 9, the P4 program assigns the new egress port to the `standard_metadata` egress port field (i.e., the field that the traffic manager looks at to determine which port the packet must be sent to). Line 10 assigns the destination MAC address passed as parameter to the packet's new destination address.

Lines 12-22 implement a table named `ipv4_lpm`. The table is matching against the destination IP address using the LPM type. The actions associated with the table are `forward` and `drop`. The default action is invoked when there is a miss. The maximum number of entries is defined by the programmer (i.e., 1024 entries, see line 20).

The control block starts executing from the apply statement (see lines 23-27) which contains the control logic. In this program, the `ipv4_lpm` table is activated in case the incoming packet has a valid IPv4 header.



```
 1:  /***********************INGRESS PROCESSING***********************/
 2:  control MyIngress(inout headers hdr,
 3:                    inout metadata meta,
 4:                    inout standard_metadata_t standard_metadata){
 5:      action drop(){
 6:          mark_to_drop(standard_metadata);
 7:      }
 8:      action forward(egressSpec_t port, macAddr_t dstAddr) {
 9:          standard_metadata.egressSpec = port;
10:          hdr.ethernet.dstAddr = dstAddr;
11:      }
12:      table ipv4_lpm {
13:          key = {
14:              hdr.ipv4.dstAddr:lpm;
15:          }
16:          actions = {
17:              forward;
18:              drop;
19:          }
20:          size = 1024;
21:          default_action = drop();
22:      }
23:      apply {
24:          if (hdr.ipv4.isValid()){
25:              ipv4_lpm.apply();
26:          }
27:      }
28:  }
```

Figure 3. Ingress control block portion of a P4 program. The code implements a match-action table with LPM lookup.

2 Lab topology

Let’s get started by opening a simple Mininet topology using MiniEdit. The topology comprises three end hosts and one P4 programmable switch.

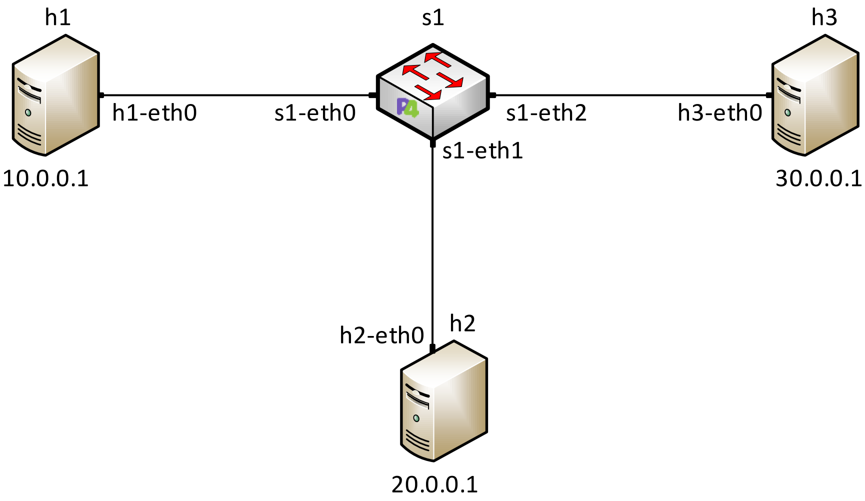


Figure 4. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

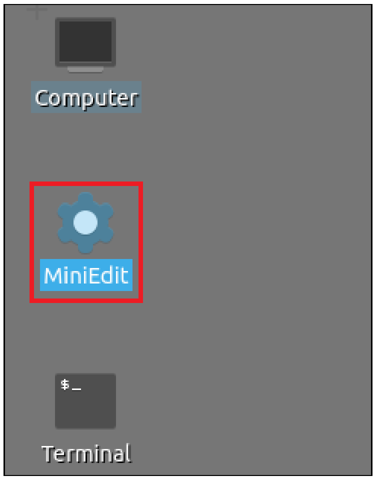


Figure 5. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab5* folder and search for the topology file called *lab5.mn* and click on Open. A new topology will be loaded to MiniEdit.

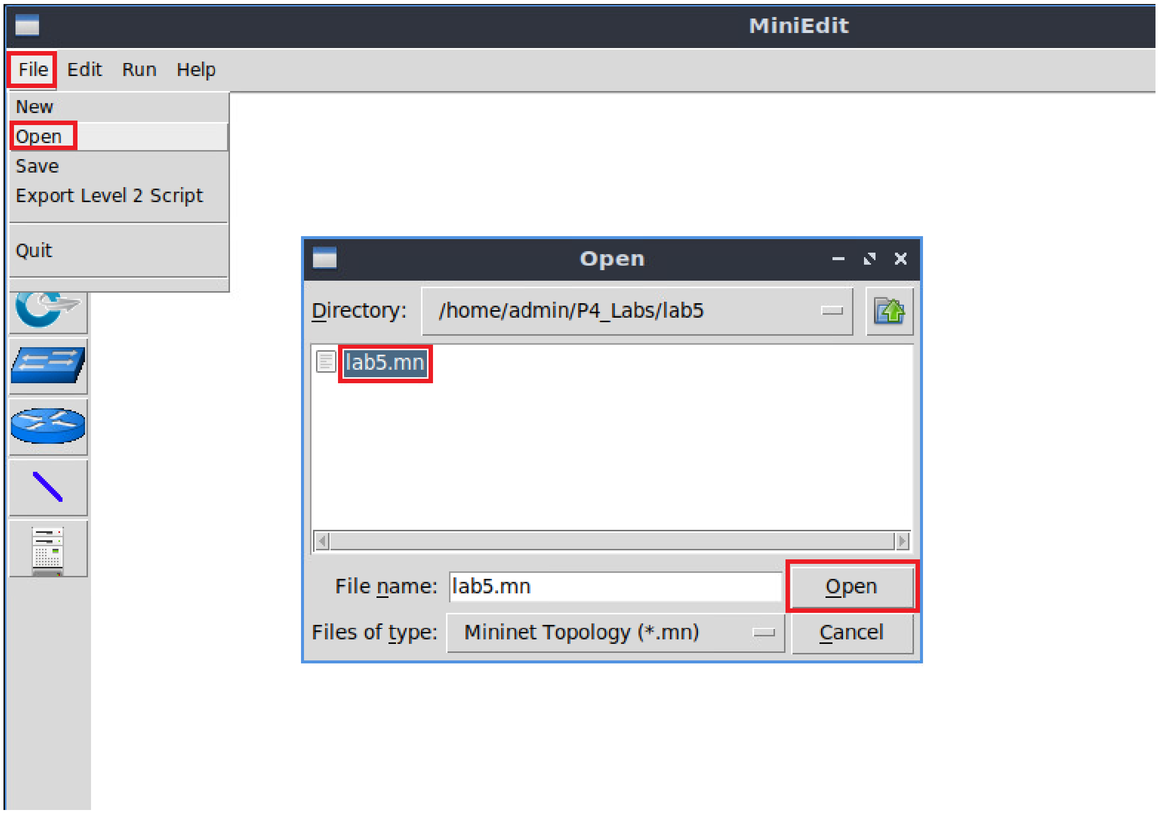


Figure 6. MiniEdit's *Open* dialog.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

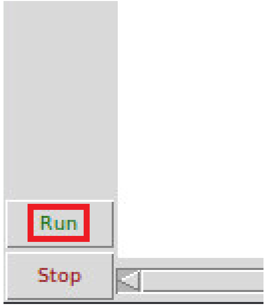


Figure 7. Running the emulation.

2.1 Starting end hosts

Step 1. Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

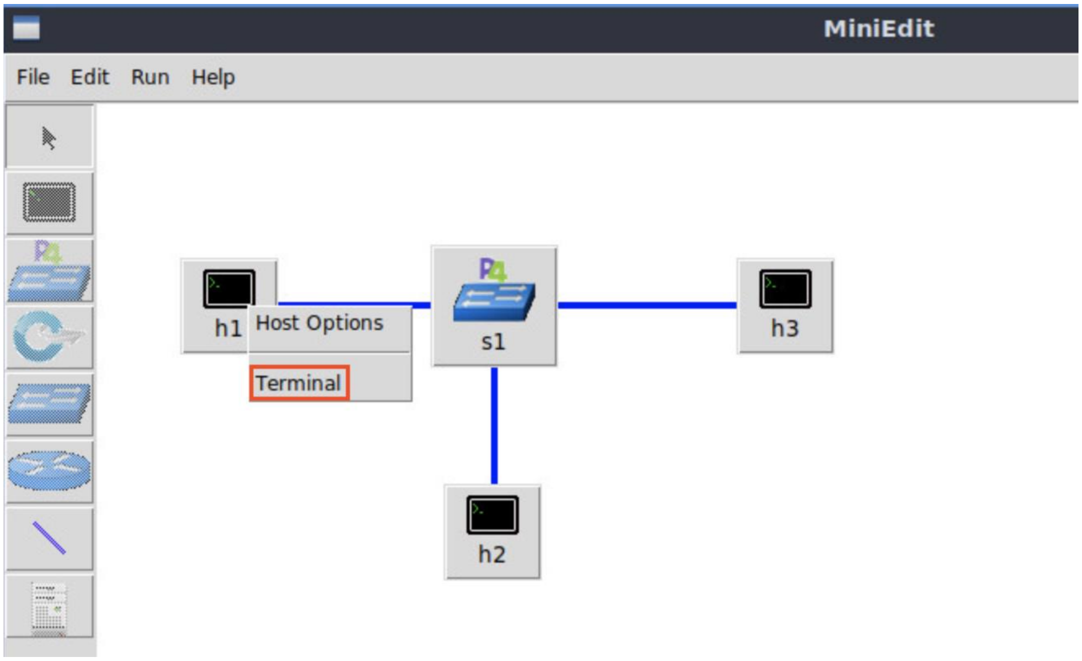
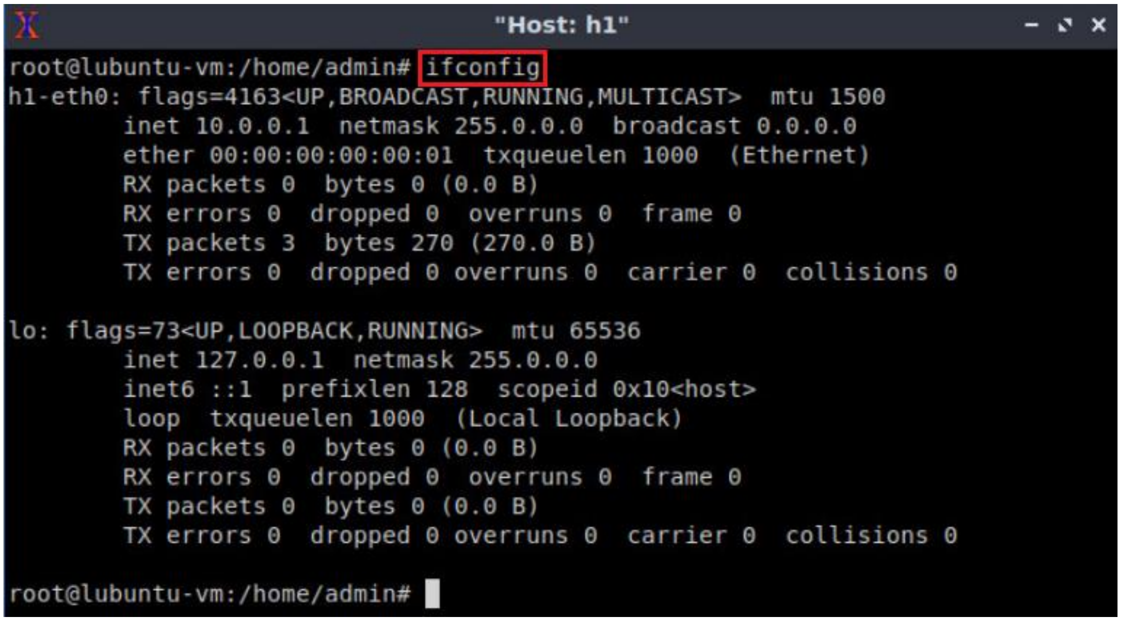


Figure 8. Opening a terminal on host h1.

Step 2. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 270 (270.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin# 
```

Figure 9. Verifying the configuration host h1 interfaces.

3 Defining a table with exact match lookup

This section demonstrates how to implement a simple table in P4 that uses exact matching on the destination IP address of the packet. When there is a match, the switch forwards the packet from a certain port. Otherwise, the switch drops the packet.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the icon located on the desktop.

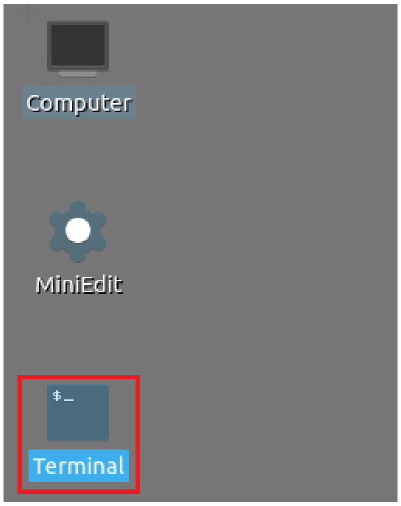
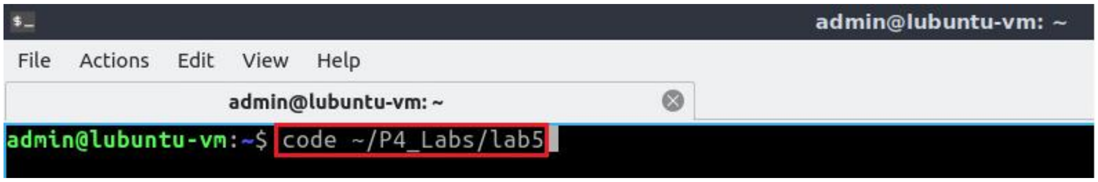


Figure 10. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI).

Step 2. In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab5
```

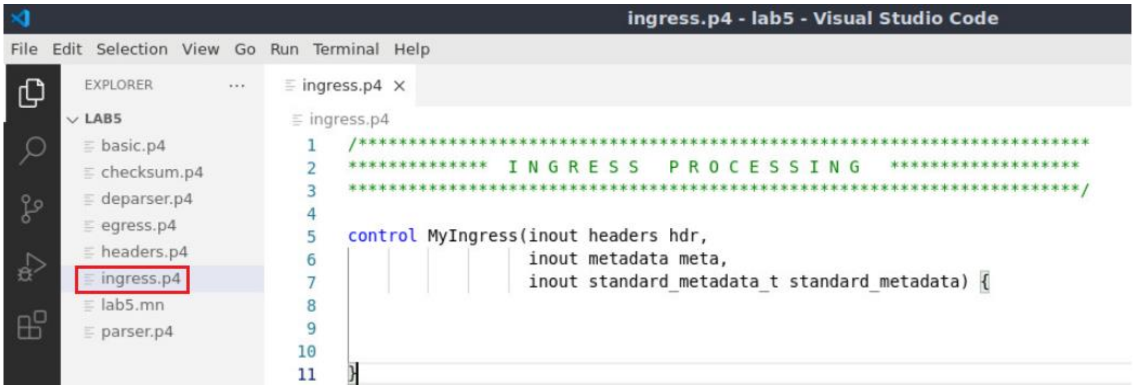


```
admin@lubuntu-vm:~$ code ~/P4_Labs/lab5
```

Figure 11. Launching the editor and opening the lab5 directory.

3.2 Programming the exact table in the ingress block

Step 1. Click on the *ingress.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.



```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {



}
```

Figure 12. Opening the ingress processing block.

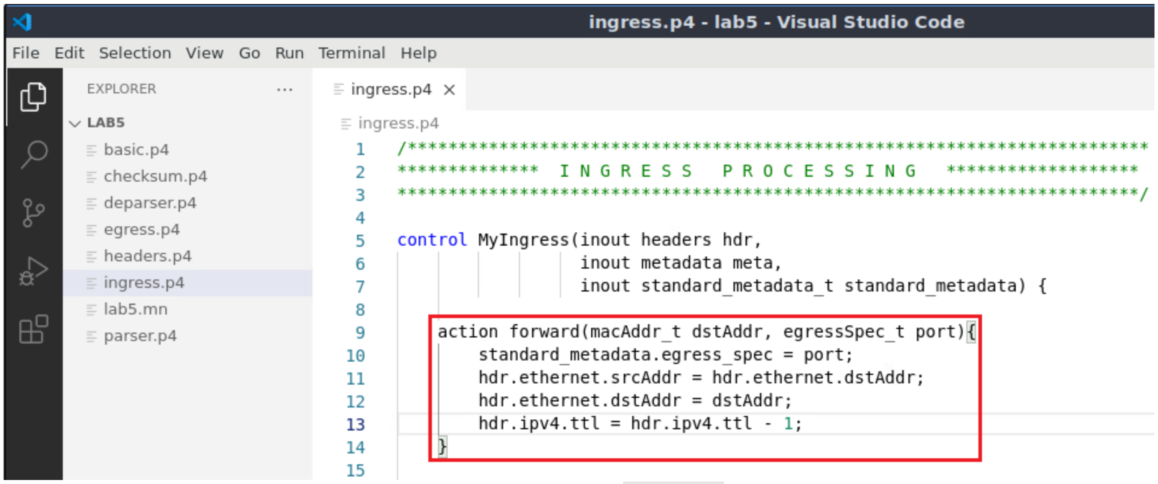
We can see that the *ingress.p4* declares a control block named *MyIngress*. Note that the body of the control block is empty. Our objective is to define a P4 table, its actions, and then invoke them inside the block.

Step 2. We will start by defining the possible actions that a table will call. In this simple forwarding program, we have two actions:

- `forward`: This action defines a set of basic operations on a packet header. Such operations are defined as follows: 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.
- `drop`: this action will be used to drop the packet.

Step 3. The following code fragment describes the behavior of the `forward` action. Insert the code below inside the *MyIngress* control block.

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



```
1  /***************************************************************************
2  *************  I N G R E S S   P R O C E S S I N G   *******************
3  ***************************************************************************/
4
5  control MyIngress(inout headers hdr,
6                    inout metadata meta,
7                    inout standard_metadata_t standard_metadata) {
8
9      action forward(macAddr_t dstAddr, egressSpec_t port){
10         standard_metadata.egress_spec = port;
11         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
12         hdr.ethernet.dstAddr = dstAddr;
13         hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
14     }
15
```

Figure 13. Defining the `forward` action.

The action `forward` accepts as parameters the next hop's MAC address (i.e., `macAddr_t dstAddr`) and the port number (i.e., `egressSpec_t port`) to be used by the switch to forward the packet. Note that `egressSpec_t` is just a typedef that corresponds to `bit<9>` and `macAddr_t` is a typedef that corresponds to `bit<48>`. These types are defined in the *headers.p4* file.

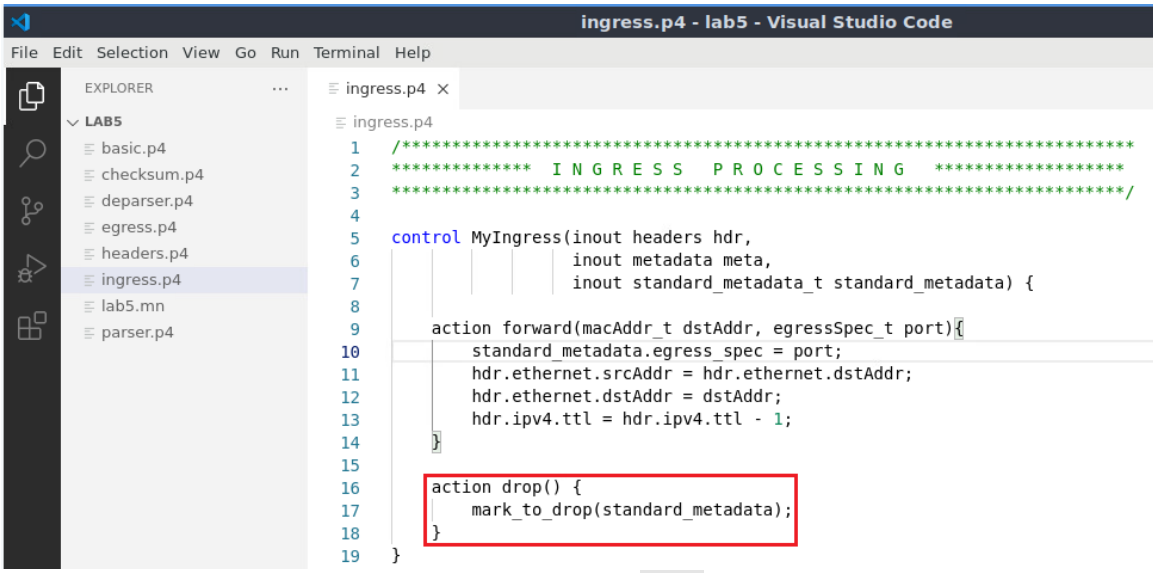
The `standard_metadata` is an instance of the `standard_metadata_t` struct provided by the V1Model[1]. This struct contains intrinsic metadata used in packet processing and in more advanced features. For example, to determine the port on which a packet arrives, we can use the `ingress_port` field in the `standard_metadata` (i.e., `standard_metadata.ingress_port`). Similarly, the egress port `egress_spec` field of the `standard_metadata` defines the egress port. Line 12 shows how to assign the egress port to forward an incoming packet to its destination.

To modify header fields inside the packet, we refer to the field name based on where it exists inside the headers. Recall that the names of the headers and the fields are defined by the programmer. The file *headers.p4* defines the program's headers. Line 13 shows how we are assigning the destination MAC address of the packet (i.e., `hdr.ethernet.dstAddr`) to be the new source MAC of the packet (i.e., `hdr.ethernet.srcAddr`). Line 14 shows how we are assigning the destination MAC address which is provided as a parameter (assigned later in the control plane) to be the new destination MAC of the packet.

It is possible in P4 to perform basic arithmetic operations on header fields and other variables. In line 15, we are decrementing the TTL value of the header field.

Step 4. Now we will define the drop action. Insert the code below inside the *MyIngress* control block.

```
action drop() {
    mark_to_drop(standard_metadata);
}
```



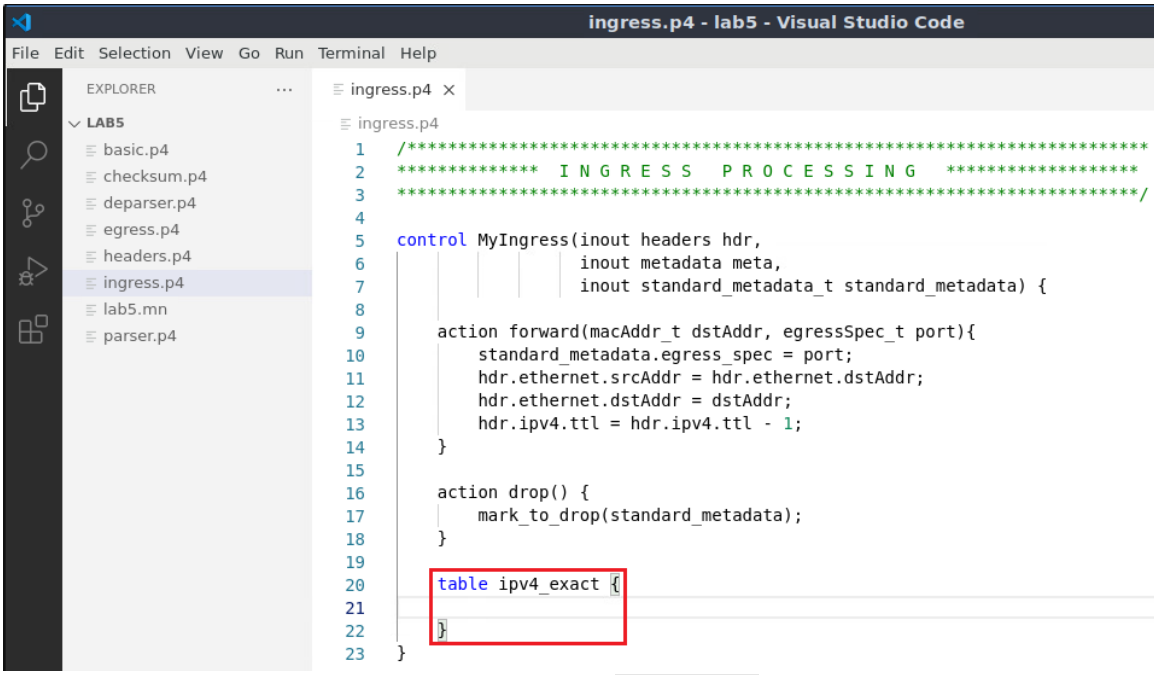
```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

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

    action drop() {
        mark_to_drop(standard_metadata);
    }
}
```

Figure 14. Defining the `drop` action.

The `drop()` action invokes a primitive action `mark_to_drop()` that modifies the `standard_metadata.egress_spec` to an implementation-specific special value that causes the packet to be dropped.

Step 5. Now we will define the table named `ipv4_exact`. Write the following piece of code inside the body of the *MyIngress* control block.

```
table ipv4_exact {

}
```



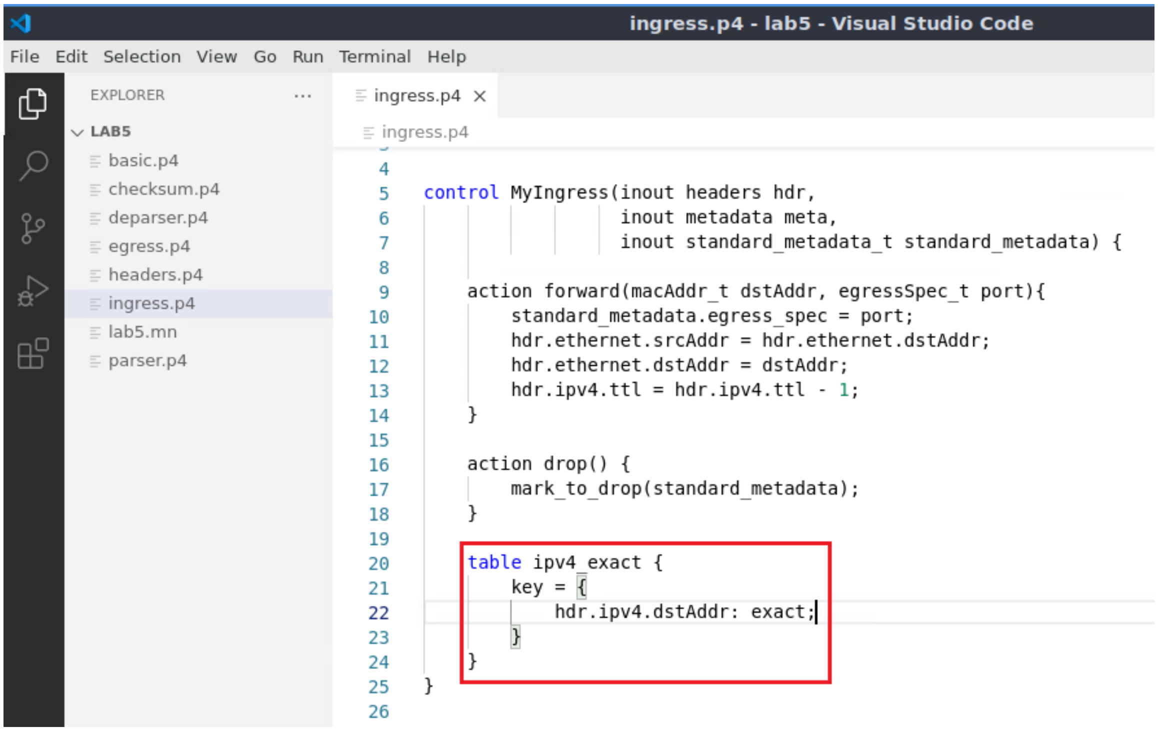
```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table ipv4_exact {

    }
}
```

Figure 15. Declaring the `ipv4_exact` table.

Tables require keys and actions. In the next step we will define a key.

Step 6. Add the following code inside the forwarding table.

```
key = {
        hdr.ipv4.dstAddr: exact;
    }
```



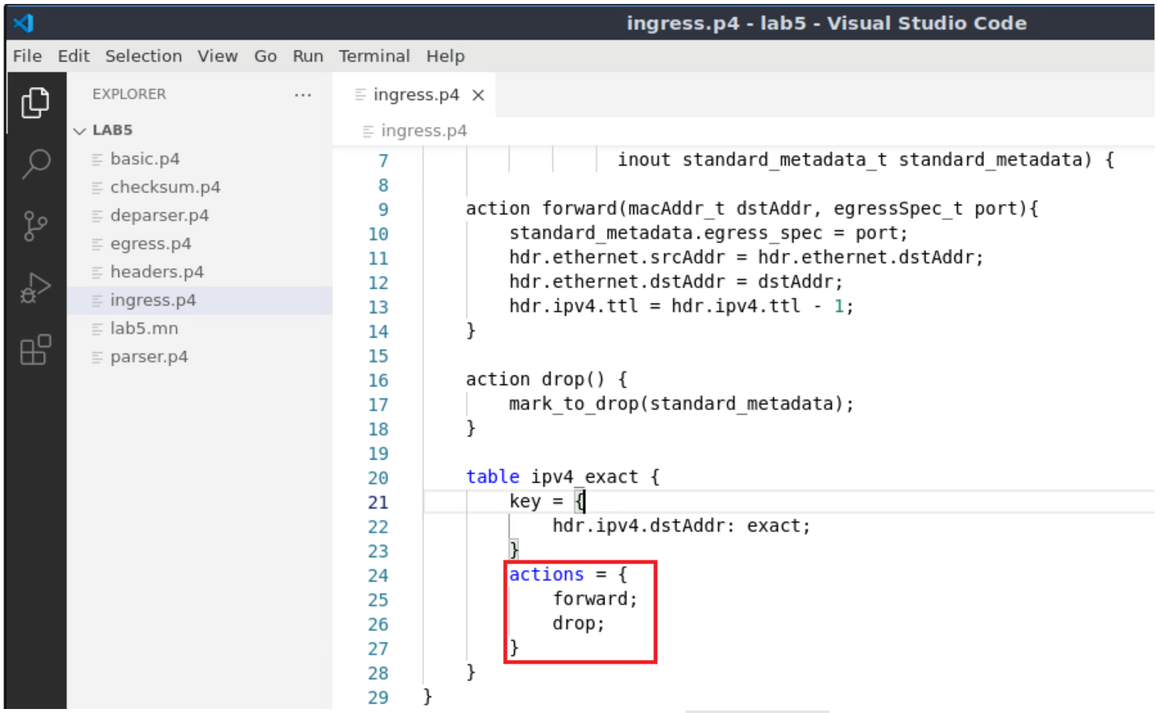
```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table ipv4_exact {
        key = {
            hdr.ipv4.dstAddr: exact;
        }
    }
}
```

Figure 16. Specifying the key and the match type.

The inserted code specifies that the destination IPv4 address of a packet (`hdr.ipv4.dstAddr`) will be used as a key in the table. Also, the match type is `exact`, denoting that the value of the destination IP address will be matched as is against a value specified later in the control plane.

Step 7. Add the following code inside the forwarding table to list the possible actions that will be used in this table.

```
actions = {
    forward;
    drop;
}
```



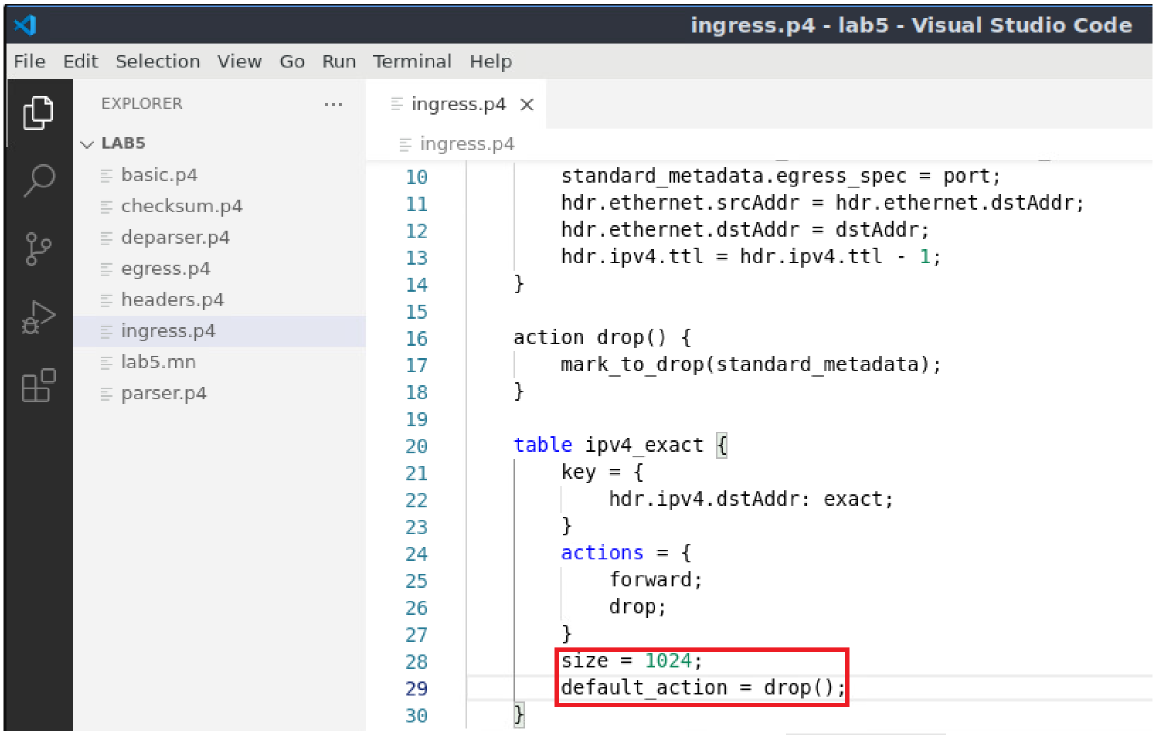
```
                    inout standard_metadata_t standard_metadata) {

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table ipv4_exact {
        key = {
            hdr.ipv4.dstAddr: exact;
        }
        actions = {
            forward;
            drop;
        }
    }
}
```

Figure 17. Adding the actions to the `ipv4_exact` table.

The code above defines the possible actions.

Step 8. Add the following code inside the forwarding table. The `size` keyword specifies the maximum number of entries that can be inserted into this table from the control plane. The `default_action` keyword specifies which default action to be invoked whenever there is a miss.

```
size = 1024;
default_action = drop();
```



```
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table ipv4_exact {
        key = {
            hdr.ipv4.dstAddr: exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }
```

Figure 18. Specifying the size and default action of the `ipv4_exact` table.

The code above denotes that a maximum of 1024 rules can be inserted into the table, and the default action to take whenever we have a miss is the `drop()` action.

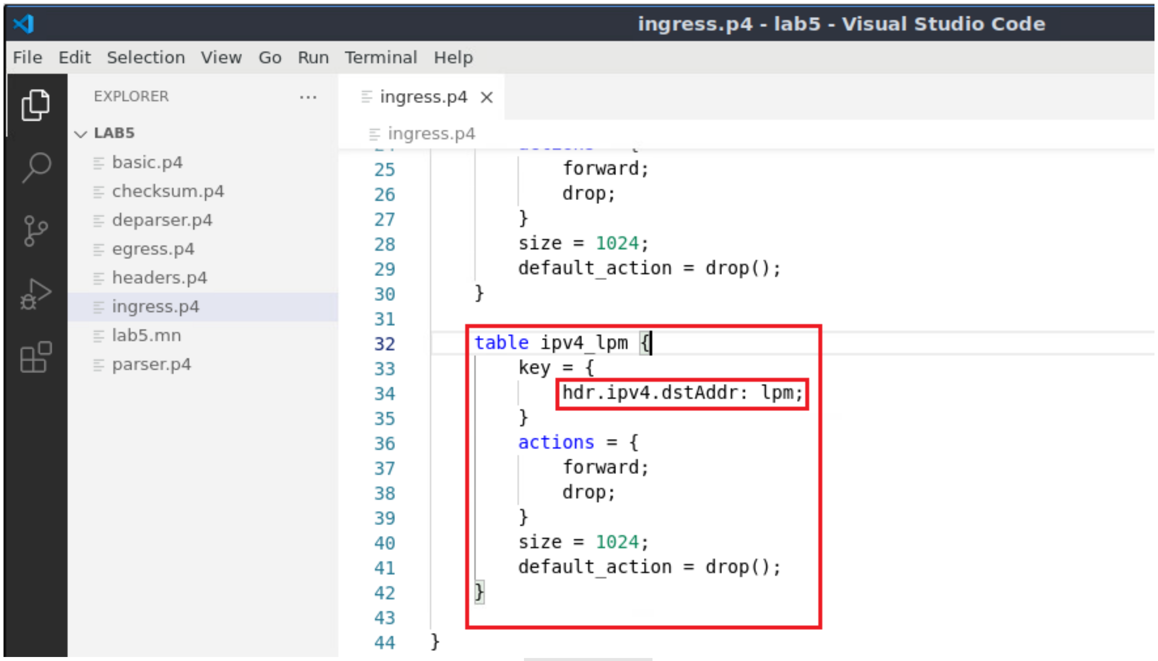
4 Defining a table with LPM matching

This section demonstrates how to implement a simple table in P4 that uses LPM matching on the packet's destination IP address. When there is a match, the switch forwards the packet from a certain port. Otherwise, the switch drops the packet.

4.1 Programming the ingress block

Step 1. Now we will define a table that performs a LPM on the destination IP address of the packet. The table will be invoking the forward and the drop actions, and hence, those actions will be listed inside the table definition.

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        forward;
        drop;
    }
    size = 1024;
    default_action = drop();
}
```



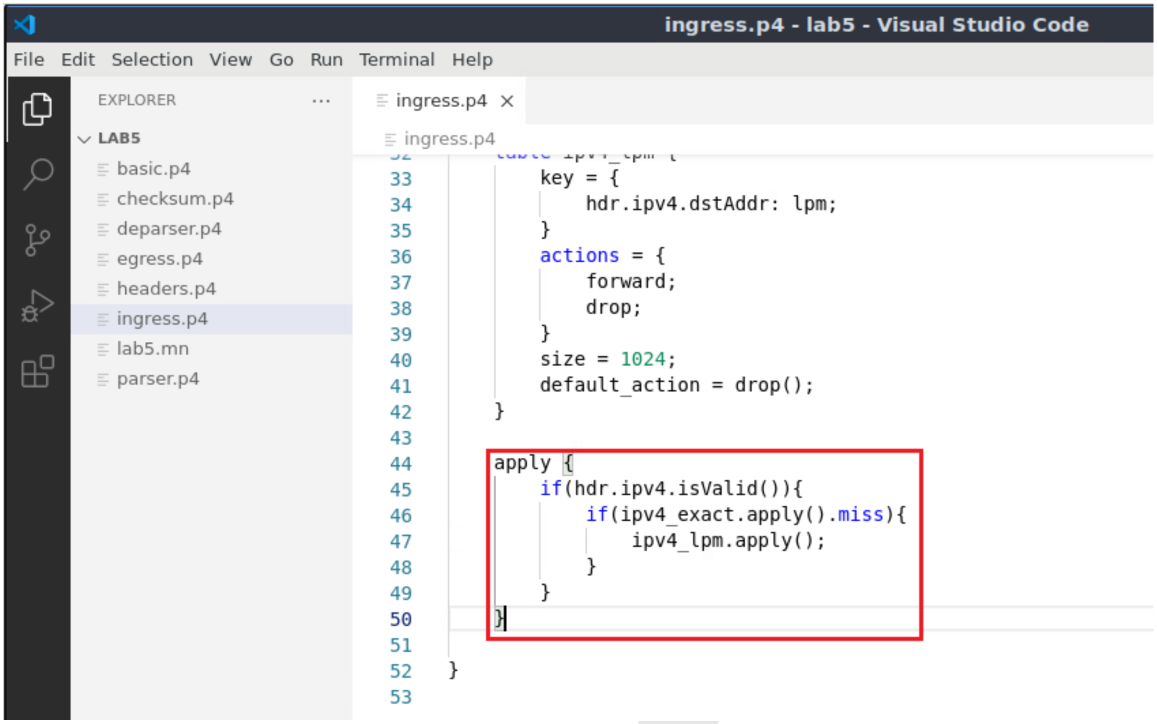
```
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

}
```

Figure 19. Defining the table `ipv4_lpm` implementing LPM lookup.

The code above shows that the match type is `lpm`. The possible actions are `forward` and `drop`. A maximum of 1024 rules can be inserted into the table, and the default action to take whenever we have a miss is the `drop()` action.

Step 2. Add the following code at the end of the *MyIngress* block. The `apply` block defines the sequential flow of packet processing. It is required in every control block, otherwise the program will not compile. It describes the sequence of tables to be invoked, in addition to other packet processing instructions.

```
apply {
    if(hdr.ipv4.isValid()) {
        if(ipv4_exact.apply().miss) {
            ipv4_lpm.apply();
        }
    }
}
```



```
        table ipv4_lpm {
            key = {
                hdr.ipv4.dstAddr: lpm;
            }
            actions = {
                forward;
                drop;
            }
            size = 1024;
            default_action = drop();
        }

        apply {
            if(hdr.ipv4.isValid()){
                if(ipv4_exact.apply().miss){
                    ipv4_lpm.apply();
                }
            }
        }

    }
```

Figure 20. Defining the `apply` block.

The logic of the code above is as follows: if the packet has an IPv4 header, apply the `ipv4_exact` table which performs an exact match lookup on the destination IP address. If there is no *hit* (i.e., the table does not contain a rule that corresponds to this IPv4 address, denoted by the *miss* keyword), apply the `ipv4_lpm` table, which matches the destination IP address of the packet against a network address.

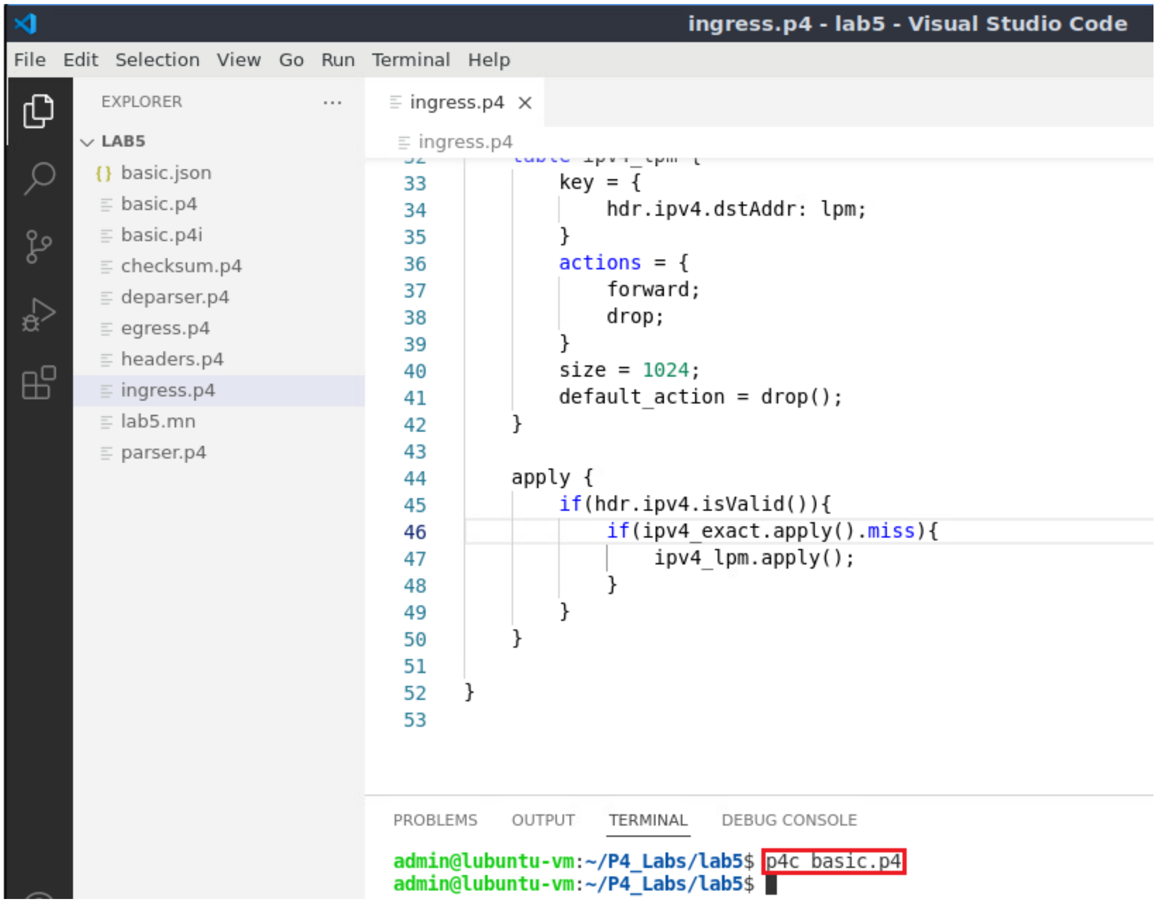
Step 3. Save the changes to the file by pressing `Ctrl + s`.

5 Loading the P4 program

5.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside the VS Code to compile the program.

```
p4c basic.p4
```



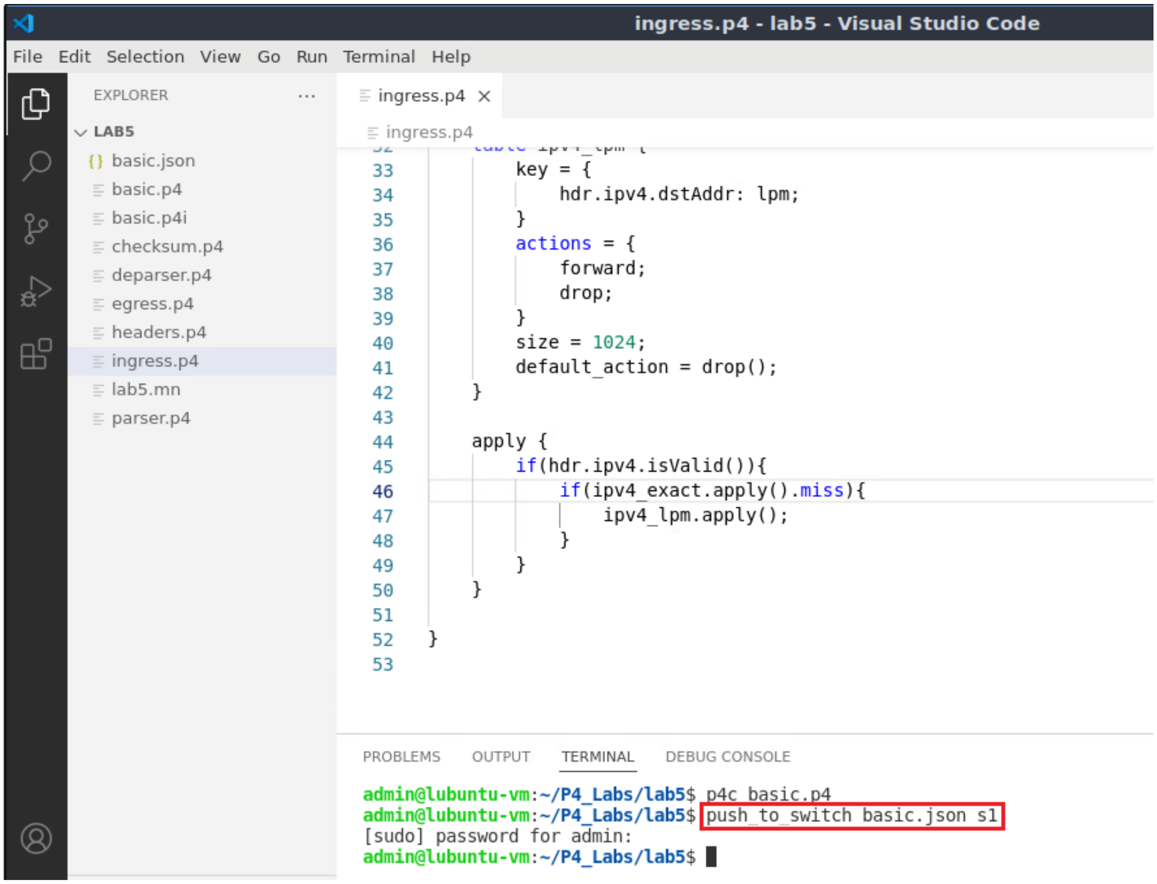
```
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if(hdr.ipv4.isValid()){
            if(ipv4_exact.apply().miss){
                ipv4_lpm.apply();
            }
        }
    }

}
```

```
admin@lubuntu-vm:~/P4_Labs/lab5$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab5$ 
```

Figure 21. Compiling a P4 program.

Step 2. Type the command below in the terminal panel to push the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the p4c compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



```
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if(hdr.ipv4.isValid()){
            if(ipv4_exact.apply().miss){
                ipv4_lpm.apply();
            }
        }
    }

}

```

```
admin@lubuntu-vm:~/P4_Labs/lab5$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab5$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab5$
```

Figure 22. Pushing the *basic.json* file to switch s1.

5.2 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.

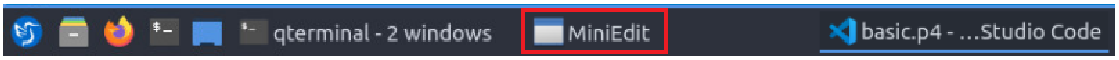


Figure 23. Maximizing the MiniEdit window.

Step 2. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

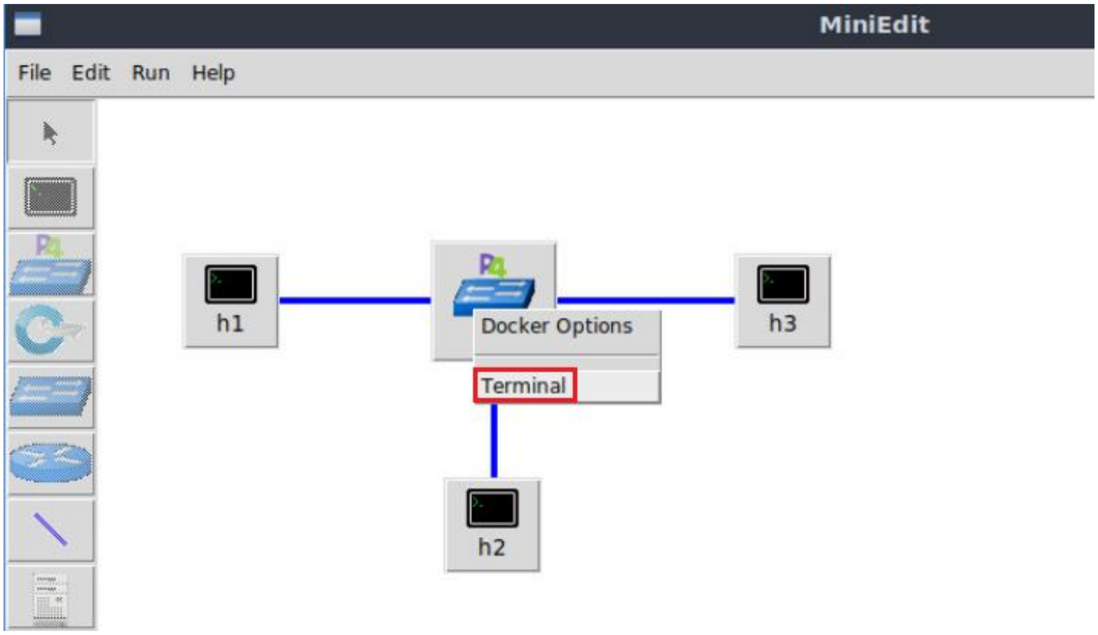
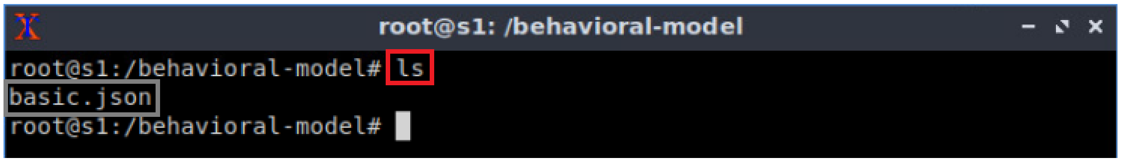


Figure 24. Opening switch s1 terminal.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

Step 3. Issue the following command on switch s1 terminal to inspect the content of the current folder.

```
ls
```



```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

Figure 25. Displaying the content of the current directory in the switch s1.

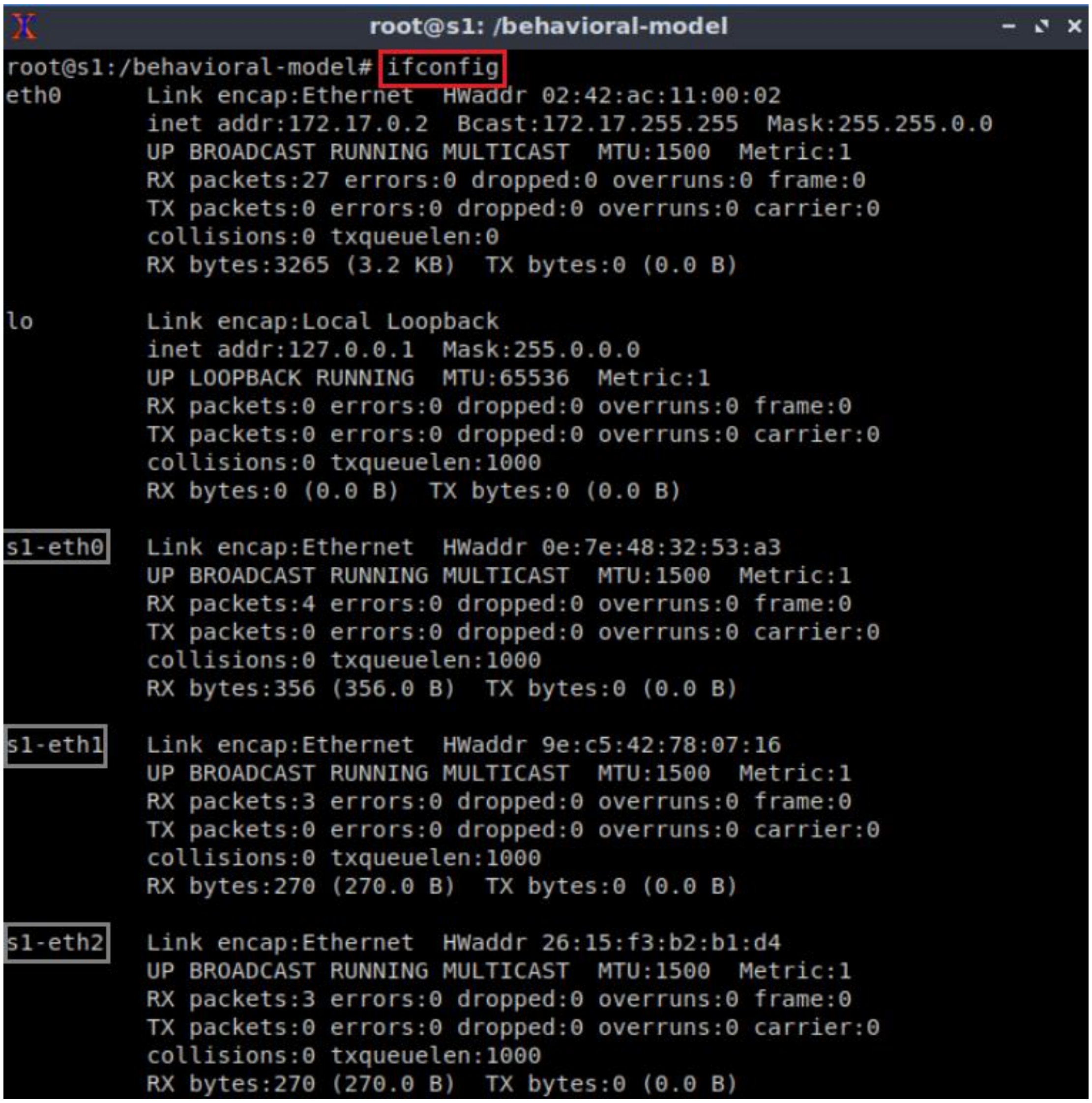
We can see that the switch contains the *basic.json* file that was pushed previously after compiling the P4 program.

6 Configuring switch s1

6.1 Mapping P4 program's ports

Step 1. Issue the following command on switch s1.

```
ifconfig
```



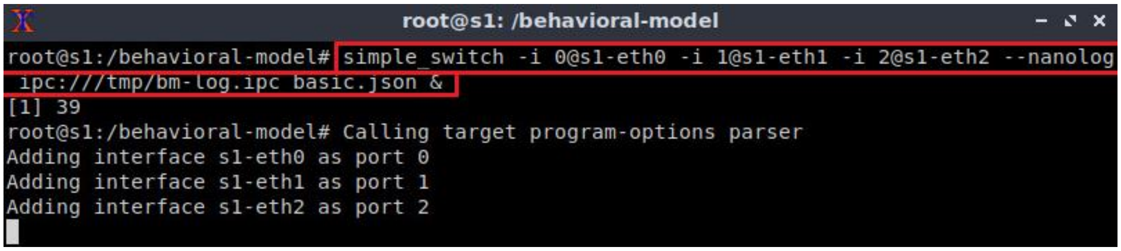
```
root@s1:/behavioral-model# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:27 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3265 (3.2 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

s1-eth0   Link encap:Ethernet  HWaddr 0e:7e:48:32:53:a3
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:356 (356.0 B)  TX bytes:0 (0.0 B)

s1-eth1   Link encap:Ethernet  HWaddr 9e:c5:42:78:07:16
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)

s1-eth2   Link encap:Ethernet  HWaddr 26:15:f3:b2:b1:d4
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:270 (270.0 B)  TX bytes:0 (0.0 B)
```

Figure 26. Displaying switch s1 interfaces.

The output displays switch s1 interfaces (i.e., *s1-eth0*, *s1-eth1* and *s1-eth2)*. The interface *s1-eth0* on the switch s1 connects to the host h1. The interface *s1-eth1* on the switch s1 connects to the host h2 and *s2-eth2* is connected to host h3.

Step 2. Start the switch daemon and map the logical interfaces (i.e., ports) to the switch's interfaces by issuing the following command. The `--nanolog` parameter is used to instruct the switch daemon to provide the switch's logs.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog ipc:///tmp/bm-
log.ipc  basic.json &
```

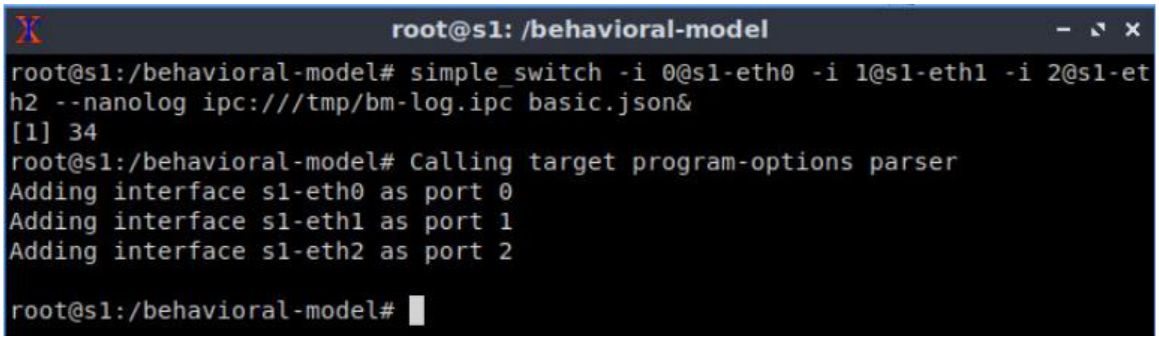


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 --nanolog
 ipc:///tmp/bm-log.ipc basic.json &
[1] 39
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 27. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

6.2 Loading the rules to the switch

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

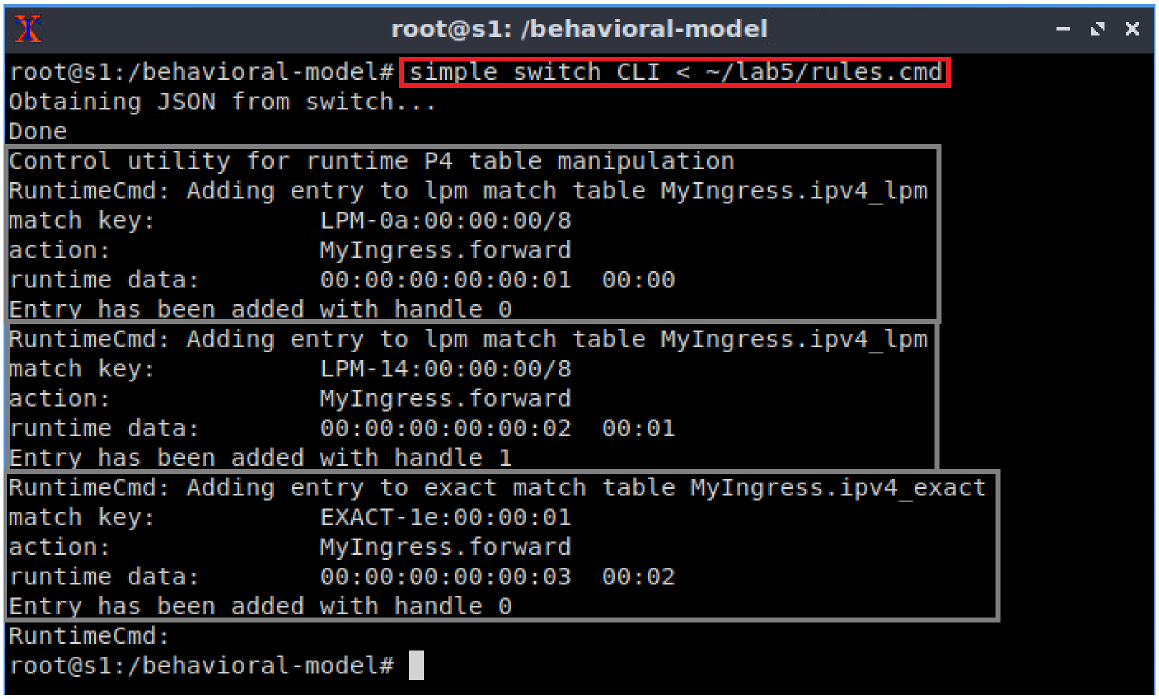


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 --nanolog ipc:///tmp/bm-log.ipc basic.json&
[1] 34
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model# 
```

Figure 28. Returning to switch s1 CLI.

Step 2. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab6/rules.cmd
```



```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab5/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to lpm match table MyIngress.ipv4_lpm
match key:           LPM-0a:00:00:00/8
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to lpm match table MyIngress.ipv4_lpm
match key:           LPM-14:00:00:00/8
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-1e:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 0
RuntimeCmd:
root@s1:/behavioral-model# 
```

Figure 29. Populating the forwarding table into switch s1.

The script above pushes the rules to the switch daemon. We can see that we added three entries to the `ipv4_exact` and `ipv4_lpm` tables.

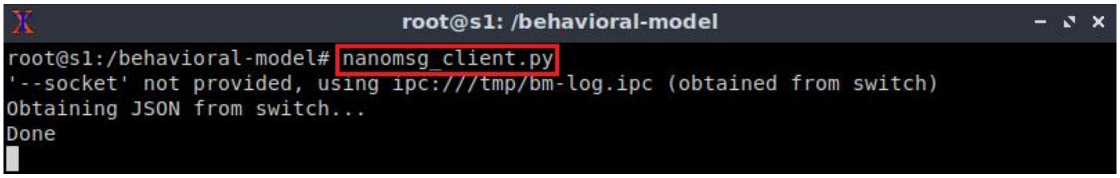
- The key of the first entry is 10.0.0.0/8 (which translates to 0a:00:00:00 in hexadecimal as shown in the figure above, next to match key) and its action is forward. This entry is added to the `ipv4_lpm` table. The action parameters or runtime data are 00:00:00:00:00:01 for the destination MAC (i.e., host h1's MAC address) and 0 for the output port (i.e., the port facing host h1).

- The key of the second entry is 20.0.0.0/8 (which translates to 14:00:00:00 in hexadecimal as shown in the figure above, next to match key) and its action is forward. This entry is added to the `ipv4_lpm` table. The action parameter or runtime data are 00:00:00:00:00:02 for the destination MAC (i.e., host h2's MAC address) and 1 for the output port (i.e., the port facing host h2).
- The key of the third entry is 30.0.0.1 (which translates to 1e:00:00:01 in hexadecimal as shown in the figure above, next to match key) and its action is forward. This entry is added to the `ipv4_exact` table. The action values are 00:00:00:00:00:03 for the destination MAC (i.e., host h3's MAC address) and 2 for the output port (i.e., the port facing host h3).

7 Testing and verifying the P4 program

Step 1. Type the following command to display the switch logs.

```
nanomsg_client.py
```

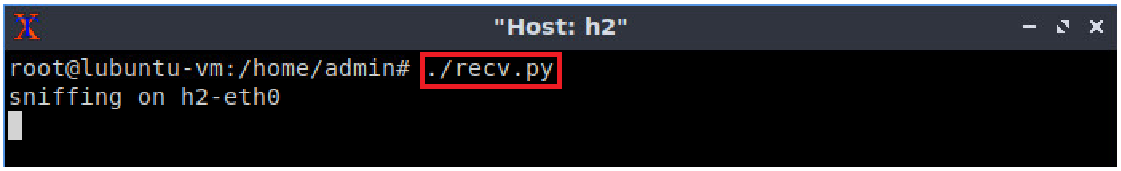


```
root@s1:/behavioral-model# nanomsg_client.py
'--socket' not provided, using ipc:///tmp/bm-log.ipc (obtained from switch)
Obtaining JSON from switch...
Done
```

Figure 30. Displaying switch s1 logs.

Step 2. On host h2's terminal, type the command the command below so that the host starts listening for packets.

```
./recv.py
```

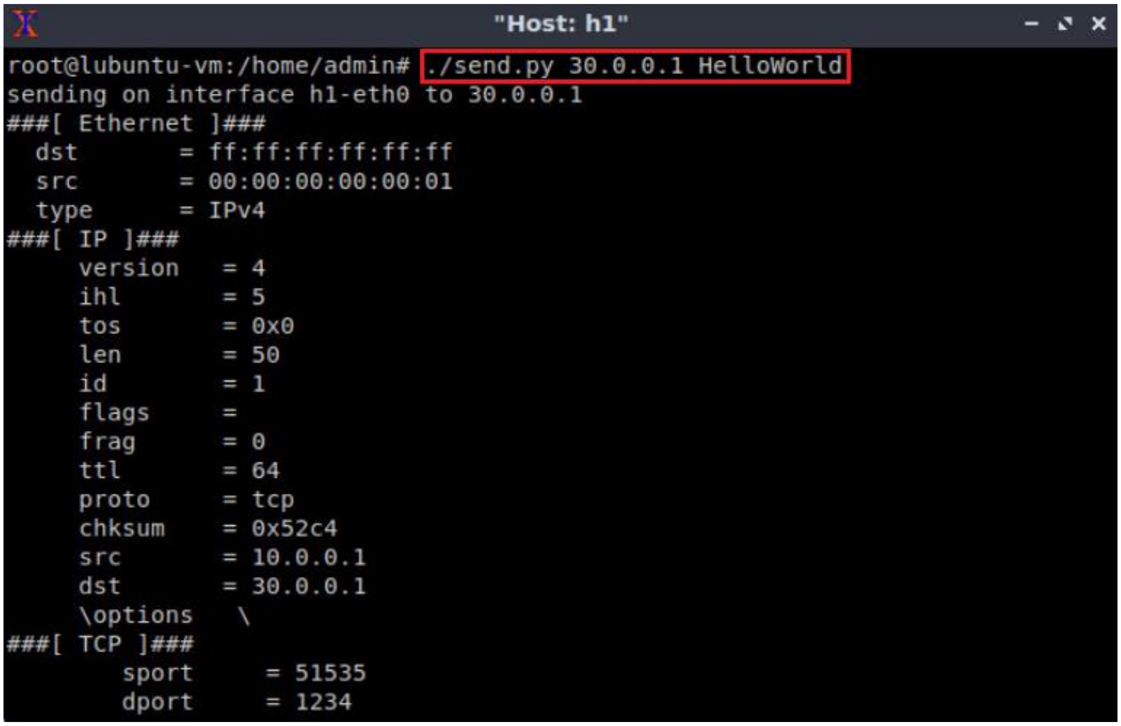


```
root@lubuntu-vm:/home/admin# ./recv.py
sniffing on h2-eth0
```

Figure 31. Listening for incoming packets in host h2.

Step 3. On host h1's terminal, type the following command to send a message to host h3.

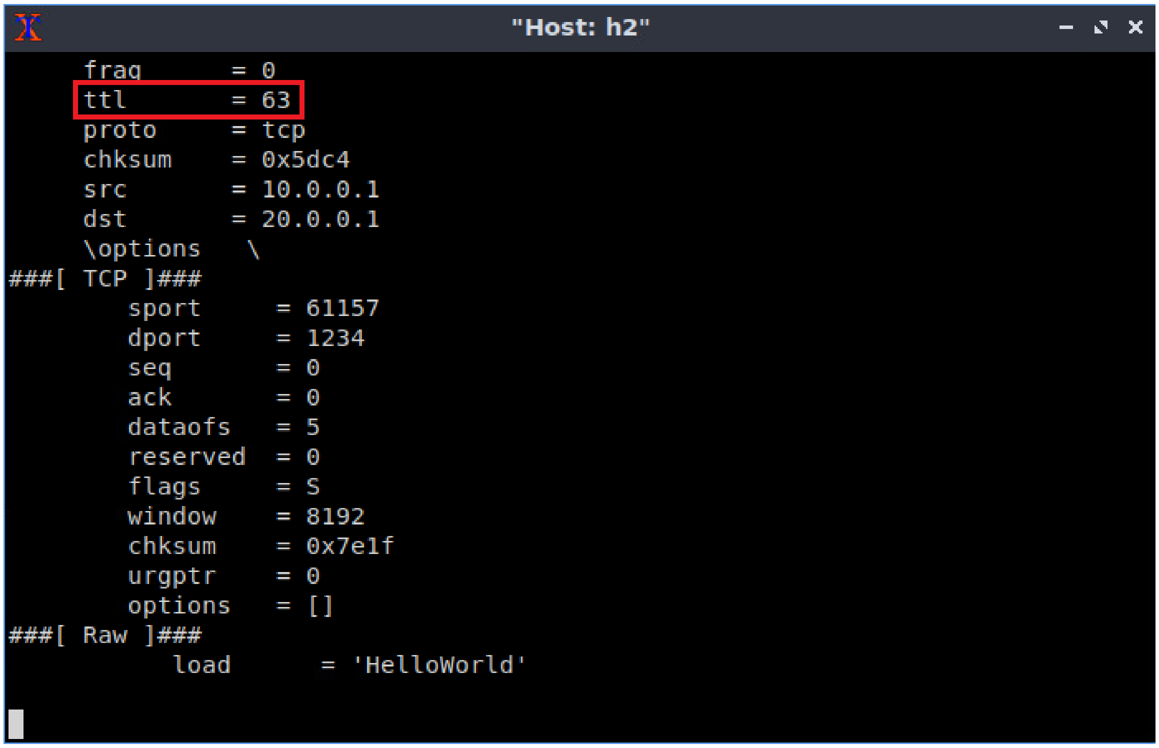
```
./send.py 30.0.0.1 HelloWorld
```



```
root@lubuntu-vm:/home/admin# ./send.py 30.0.0.1 HelloWorld
sending on interface h1-eth0 to 30.0.0.1
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 50
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x52c4
     src       = 10.0.0.1
     dst       = 30.0.0.1
     \options   \
###[ TCP ]###
        sport     = 51535
        dport     = 1234
```

Figure 32. Sending a test packet from host h1 to host h3.

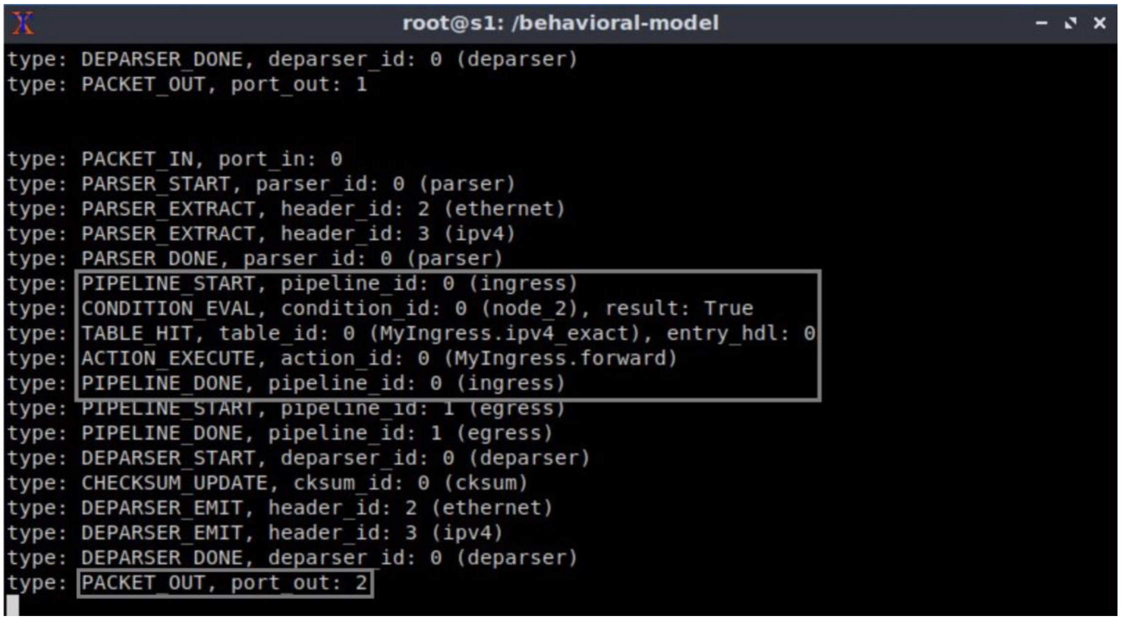
Step 4. Verify that the packet was received on host h2. Notice that the TTL was decremented.



```
     frag      = 0
     ttl       = 63
     proto     = tcp
     chksum    = 0x5dc4
     src       = 10.0.0.1
     dst       = 20.0.0.1
     \options   \
###[ TCP ]###
        sport     = 61157
        dport     = 1234
        seq       = 0
        ack       = 0
        dataofs   = 5
        reserved  = 0
        flags     = S
        window    = 8192
        chksum    = 0x7e1f
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = 'HelloWorld'
```

Figure 33. Sending a test packet from host h1 to host h3.

Step 5. Inspect the logs on switch s1 terminal.



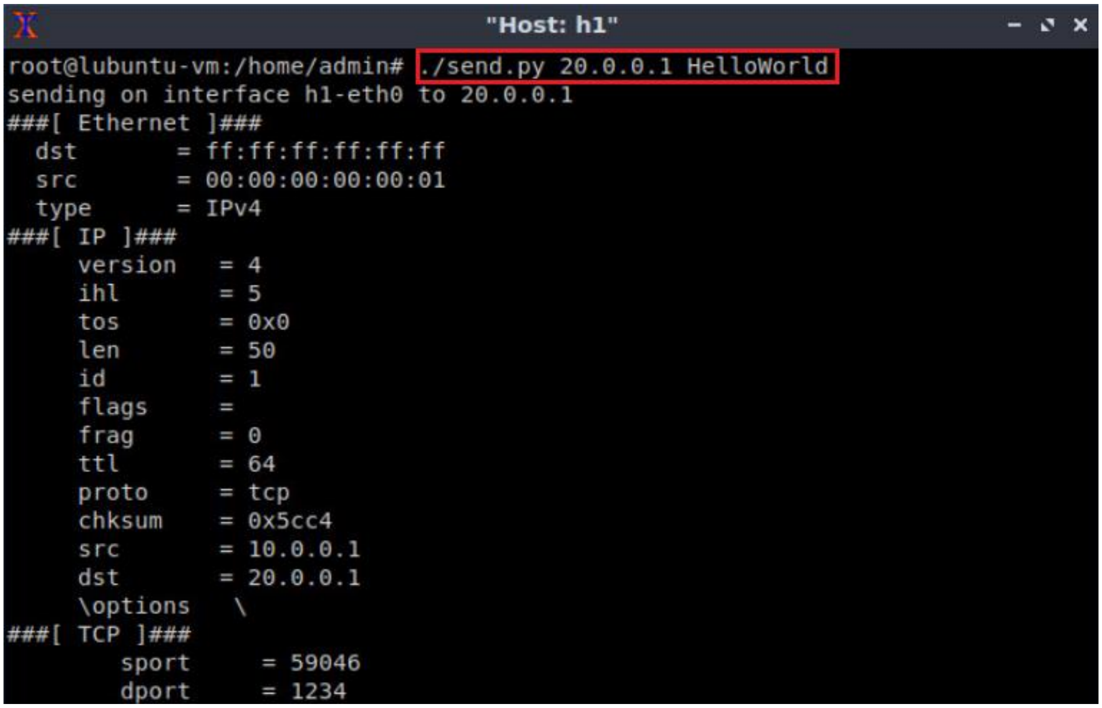
```
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1


type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, condition_id: 0 (node_2), result: True
type: TABLE_HIT, table_id: 0 (MyIngress.ipv4_exact), entry_hdl: 0
type: ACTION_EXECUTE, action_id: 0 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 2
```

Figure 34. Inspecting the logs in switch s1.

The figure above shows that there is a hit in the `ipv4_exact` table. Then, the packet is forwarded through port 2, which is connected to host h3.

Step 6. On host h1's terminal, type the following command to send a message to host h2. The output will show the Ethernet, IP and TCP header fields and their values. The payload is `HelloWorld`.

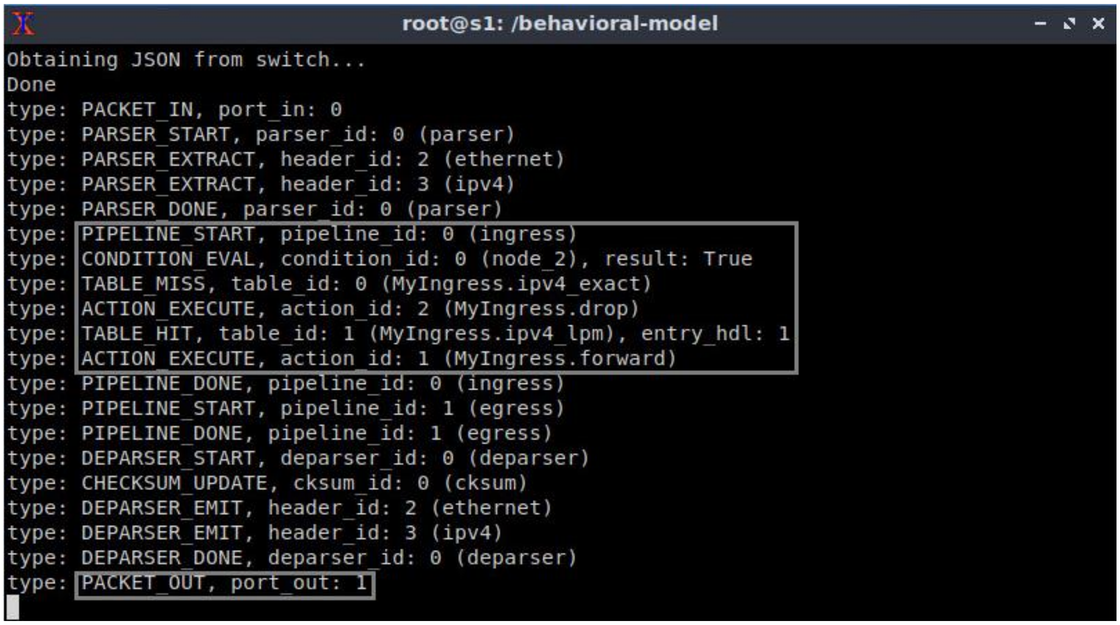
```
./send.py 20.0.0.1 HelloWorld
```



```
root@lubuntu-vm:/home/admin# ./send.py 20.0.0.1 HelloWorld
sending on interface h1-eth0 to 20.0.0.1
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 50
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x5cc4
     src       = 10.0.0.1
     dst       = 20.0.0.1
     \options   \
###[ TCP ]###
        sport     = 59046
        dport     = 1234
```

Figure 35. Sending a test packet from host h1 to host h2.

Step 7. Inspect the logs on switch s1 terminal.



```
Obtaining JSON from switch...
Done
type: PACKET_IN, port_in: 0
type: PARSER_START, parser_id: 0 (parser)
type: PARSER_EXTRACT, header_id: 2 (ethernet)
type: PARSER_EXTRACT, header_id: 3 (ipv4)
type: PARSER_DONE, parser_id: 0 (parser)
type: PIPELINE_START, pipeline_id: 0 (ingress)
type: CONDITION_EVAL, condition_id: 0 (node_2), result: True
type: TABLE_MISS, table_id: 0 (MyIngress.ipv4_exact)
type: ACTION_EXECUTE, action_id: 2 (MyIngress.drop)
type: TABLE_HIT, table_id: 1 (MyIngress.ipv4_lpm), entry_hdl: 1
type: ACTION_EXECUTE, action_id: 1 (MyIngress.forward)
type: PIPELINE_DONE, pipeline_id: 0 (ingress)
type: PIPELINE_START, pipeline_id: 1 (egress)
type: PIPELINE_DONE, pipeline_id: 1 (egress)
type: DEPARSER_START, deparser_id: 0 (deparser)
type: CHECKSUM_UPDATE, cksum_id: 0 (cksum)
type: DEPARSER_EMIT, header_id: 2 (ethernet)
type: DEPARSER_EMIT, header_id: 3 (ipv4)
type: DEPARSER_DONE, deparser_id: 0 (deparser)
type: PACKET_OUT, port_out: 1
```

Figure 36. Inspecting the logs in switch s1.

Results show that there is a miss in the `ipv4_exact` table, but there is a hit on the `ipv4_lpm` table. Then, the packet is forwarded through port 1, which is connected to host h2. This behavior corresponds to the logic described by the `apply` block in the ingress processing.

This concludes lab 5. Stop the emulation and then exit out of MiniEdit.

References

1. "p4c core.p4". [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/core.p4.
2. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.
3. Mininet walkthrough. [Online]. Available: http://Mininet.org.
4. M. Peuster, J. Kampmeyer, H. Karl. "*Containernet 2.0: A rapid prototyping platform for hybrid service function chains.*" 4th IEEE Conference on Network Softwarization and Workshops (NetSoft). 2018.
5. R. Cziva. "*ESnet tutorial - P4 deep dive, slide 28*." [Online]. Available: https://tinyurl.com/rruscv3.
6. P4lang/behavioral-model github repository. *"The BMv2 simple switch target*." [Online]. Available: https://tinyurl.com/vrasamm.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 6: Implementing a Stateful Packet Filter for the ICMP Protocol

Document Version: 04-17-2023

Contents

Overview

This lab is an introduction to stateful packet filter in P4, a technique by which a network administrator can implement network-based access control. In particular, the lab uses P4 registers to store the state of a connection. The lab further implements a stateful packet filter for Internet Control Message Protocol (ICMP) via a policy defined by the network administrator.

Objectives

By the end of this lab, students should be able to:

1. Understand stateful packet filters.
2. Understand what registers are used for.
3. Implement stateful packet filters in P4 using registers.
4. Test the defined policy for the stateful packet filter.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client machine.

Device	**Account**	**Password**
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Creating a P4 program that performs stateful packet filtering.
4. Section 4: Loading the P4 program.
5. Section 5: Testing and verifying the P4 program.

1 Introduction

Packet filters control and manage the data flow across a network by filtering and analyzing outgoing and incoming packets[1]. They are commonly implemented in firewalls or routers to protect networks from unauthorized access and malicious activities. Packet filters can be broadly classified into two categories: stateless and stateful.

Stateless packet filters operate on a per-packet basis, examining each packet individually without considering any previous packets[2]. Stateless filters use predefined rules based on packet header information, such as source and destination IP addresses, port numbers, and protocols. Based on these rules, the filter decides whether to allow or deny the packet. Stateless filters are relatively simple and fast, as they do not maintain any information about ongoing connections or packet history. However, their simplicity can also be a disadvantage, as they are unable to recognize the context of a network connection and may be less effective in detecting complex attacks or handling certain protocols.

Stateful packet filters, on the other hand, maintain a state table that tracks the status of ongoing network connections[2]. By keeping track of connection states, stateful filters can make more informed decisions about whether to allow or deny a packet. When a new packet arrives, the stateful filter examines both the packet header and the current state of the connection in its state table. If the packet is part of an existing, legitimate connection, it is allowed through; otherwise, it may be denied based on the filter's rules. Stateful packet filters provide a higher level of security compared to stateless filters, as they can better handle connection-oriented protocols and detect malicious activities that span multiple packets. However, they can be more resource-intensive and slower due to the additional overhead of maintaining and updating the state table.

1.1 P4 registers

P4 targets implement registers to save arbitrary data. Multiple packets can access the data stored in the registers. Registers in P4 are organized into named arrays of cells. Registers can be read and written by both the control and the data plane. In P4, registers are global memory resources meaning that any match-action tables can reference them.

The syntax below shows how to declare a register array in P4. The register array R1 contains `M` values of `N` bits.

```
register<bit<N>>(M) R1;
```

Figure 1 depicts a graphical representation of the register `R1`. The functions `write` and `read` are used to store and retrieve values from a specific position, where an index specifies the position[3]. For example, the programmer invokes the following function to store the value `val` in position 0 in the register array R1.

```
R1.write(0,val)
```

Similarly, the user invokes the function shown below to read a value stored in position 3. Note that the retrieved value is stored in the variable `res`.

```
R1.read(res,3)
```

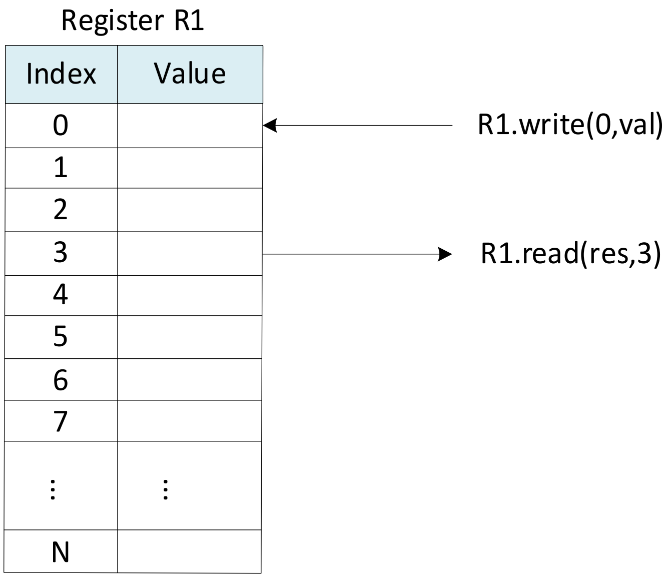


Figure 1. Register array R1. The register array contains N entries of M bits. The index indicates the position of the value. Using the functions `read` and `write`, programmers can retrieve and modify values in the register array.

1.2 Hashes in P4

P4 targets implement hash functions to map arbitrary data to a hash value. For example, the V1Model implements hash functions as externs[4]. The following code shows how to call a hash function in P4.

```
hash(hash_val, algo, min_val, {val_1, val_2, ..., val_N}, (n_bits, max_val))
```

The parameters of the hash function are as follows:

- `hash_val`: variable used to store the hash value.
- `algo`: indicates the hashing algorithm. For example, the V1Model supports crc16, crc32, universal hashing (i.e., random), xor32, and others.
- `min_val`: establishes the minimum hash value.
- `{val_1,val_2,…,val_N}`: values to be hashed.
- `n_bit`: number of bits of the output (i.e., width).
- `max_val`: maximum hash value.

1.3 Lab scenario

This lab demonstrates how to implement a stateful packet filter for the ICMP protocol using registers. Hashes are used to identify a flow, and registers are used to store the flow's state. The stateful packet filter will only allow hosts in the internal network to originate ping tests towards hosts in the external network. ICMP flows that are not originating from the internal network are dropped.

The P4 program presented in this lab performs the following:

1- If the packet is an ICMP REQUEST, the state of the flow (i.e., the ICMP identifier) is stored in a register.
2- If the packet is an ICMP reply, the ICMP identifier of the flow is extracted from the register. If the extracted ICMP identifier matches the one in the ICMP REPLY headers, the packet is accepted.

2 Lab topology

Let us get started by opening a simple Mininet topology using MiniEdit. The topology comprises three end hosts and one P4 programmable switch. Host h1 is in the internal network, host h2 is in the DMZ network, and host h3 is in the external network.

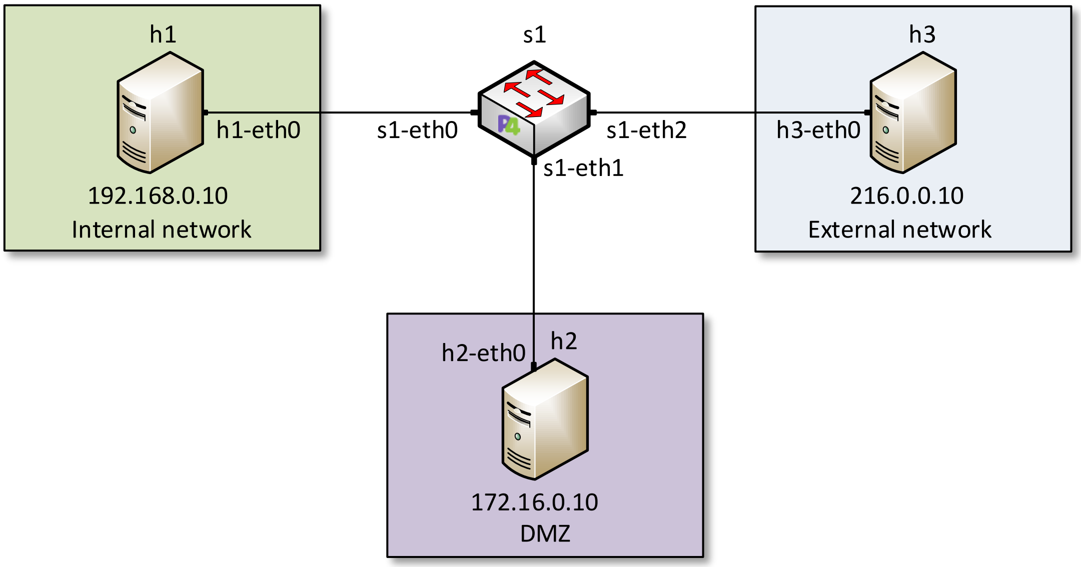


Figure 2. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

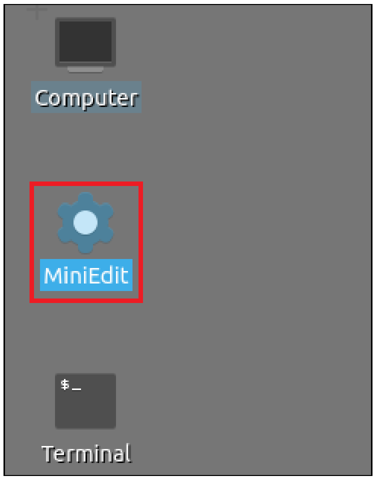
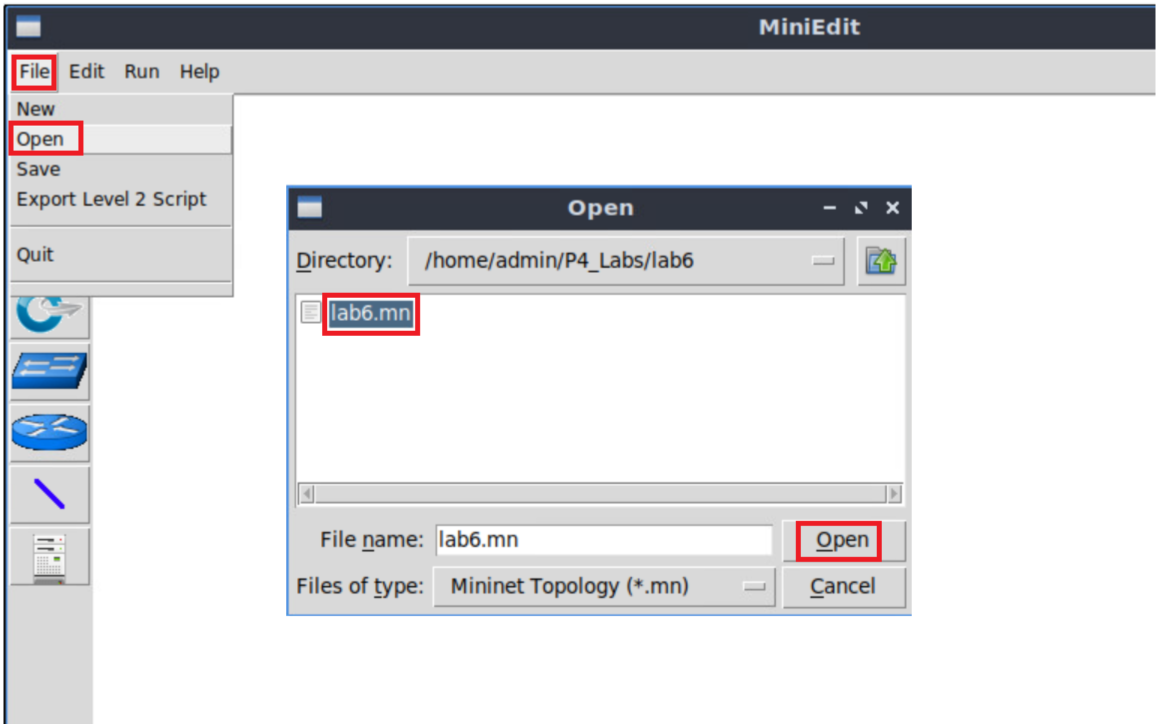


Figure 3. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab6* folder and search for the topology file called *lab6.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4_Labs/lab6

lab6.mn

File name: lab6.mn

Files of type: Mininet Topology (*.mn)

Figure 4. MiniEdit's *Open* dialog.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

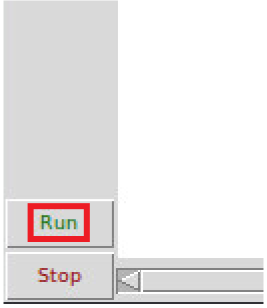


Figure 5. Running the emulation.

2.1 Verifying the configuration of the end hosts

Step 1. Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

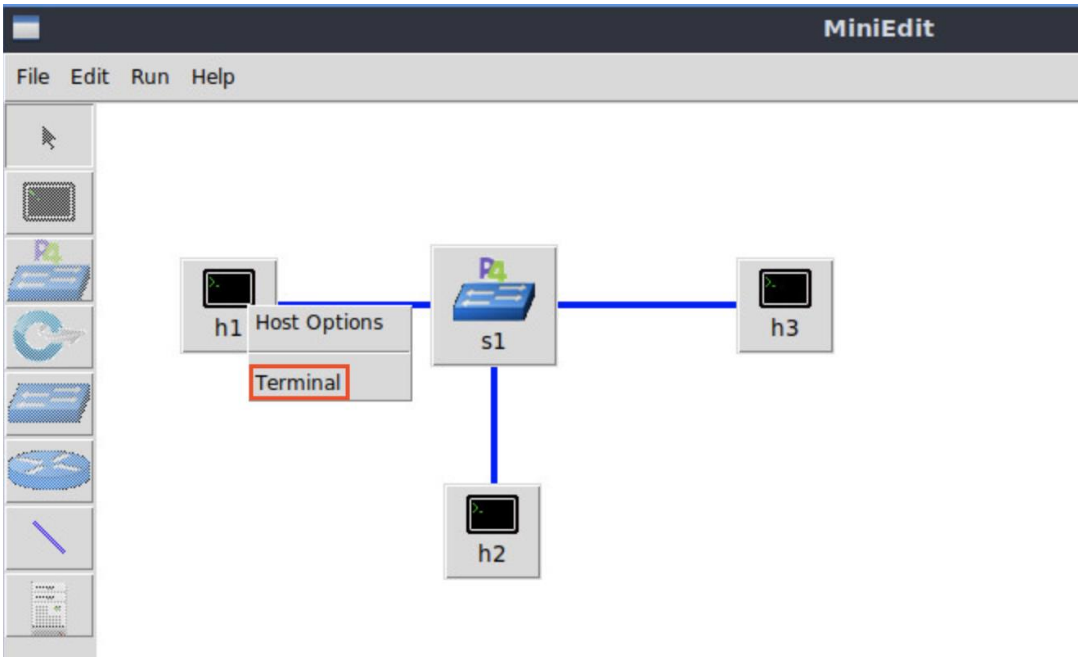
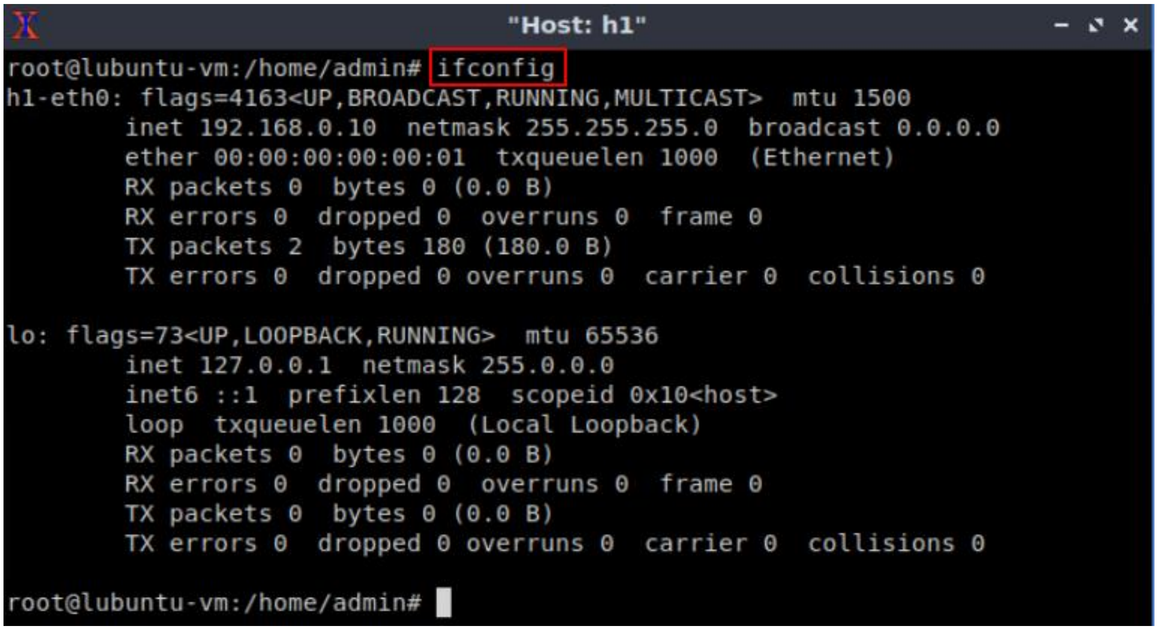


Figure 6. Opening a terminal on host h1.

Step 2. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 180 (180.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin# 
```

Figure 7. Verifying the configuration host h1 interfaces.

Step 3. Hold the right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

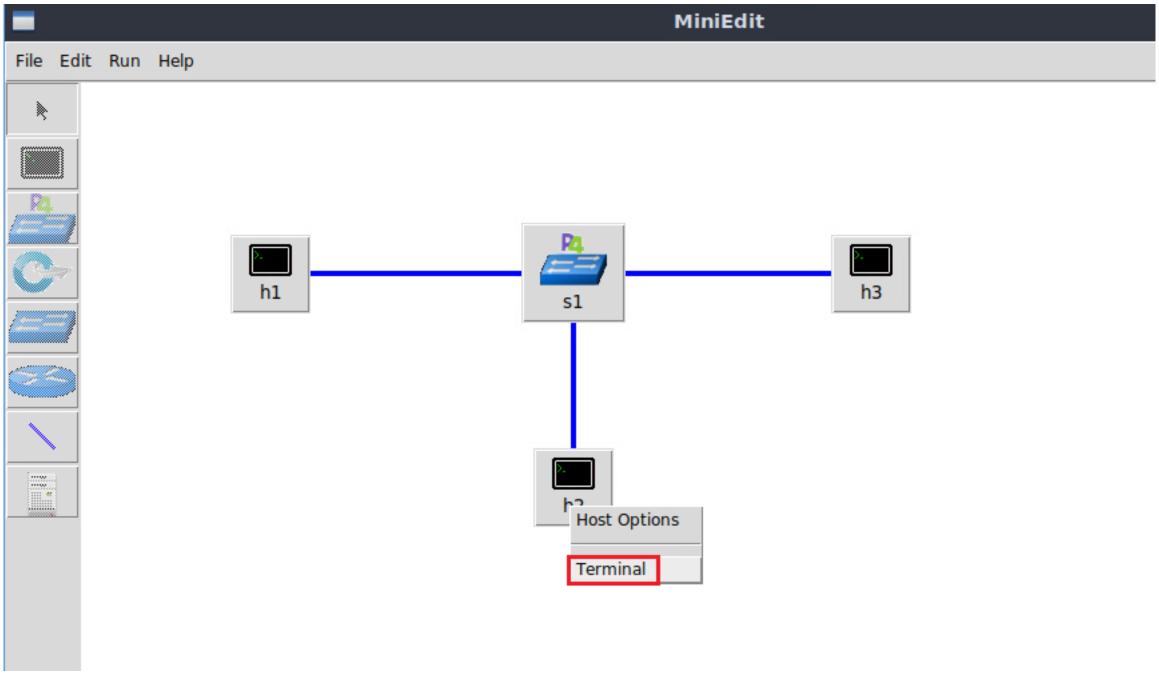
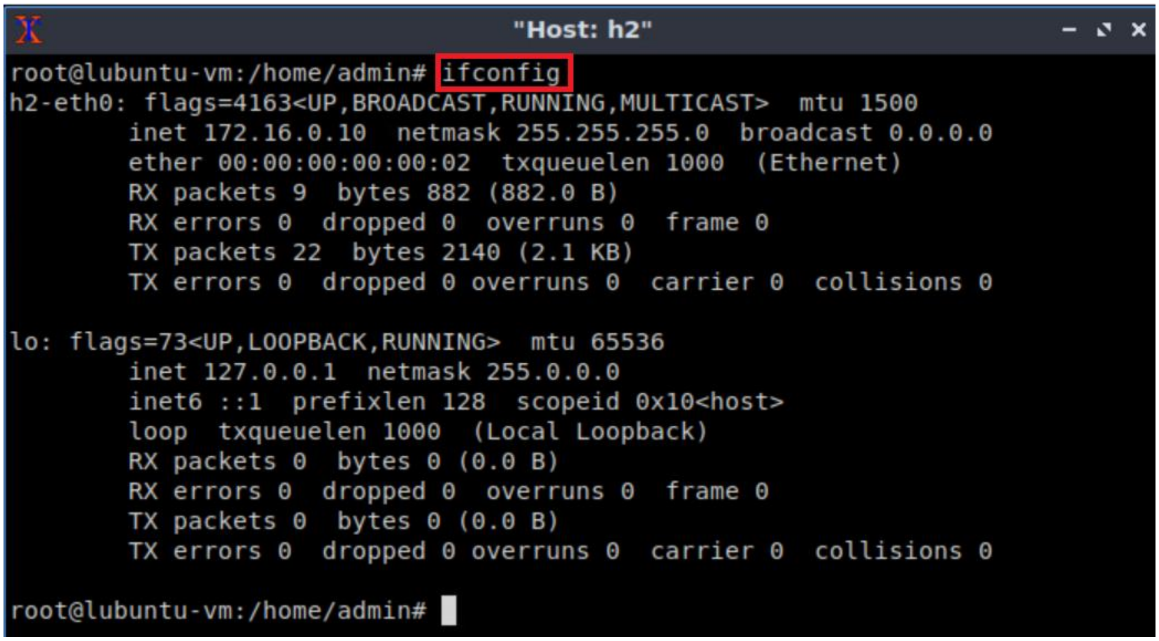


Figure 8. Opening a terminal on host h2.

Step 4. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.16.0.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:02  txqueuelen 1000  (Ethernet)
        RX packets 9  bytes 882 (882.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2140 (2.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin# 
```

Figure 9. Verifying the configuration host h2 interfaces.

Step 5. Hold the right-click on host h3 and select *Terminal*. This opens the terminal of host h3 and allows the execution of commands on that host.

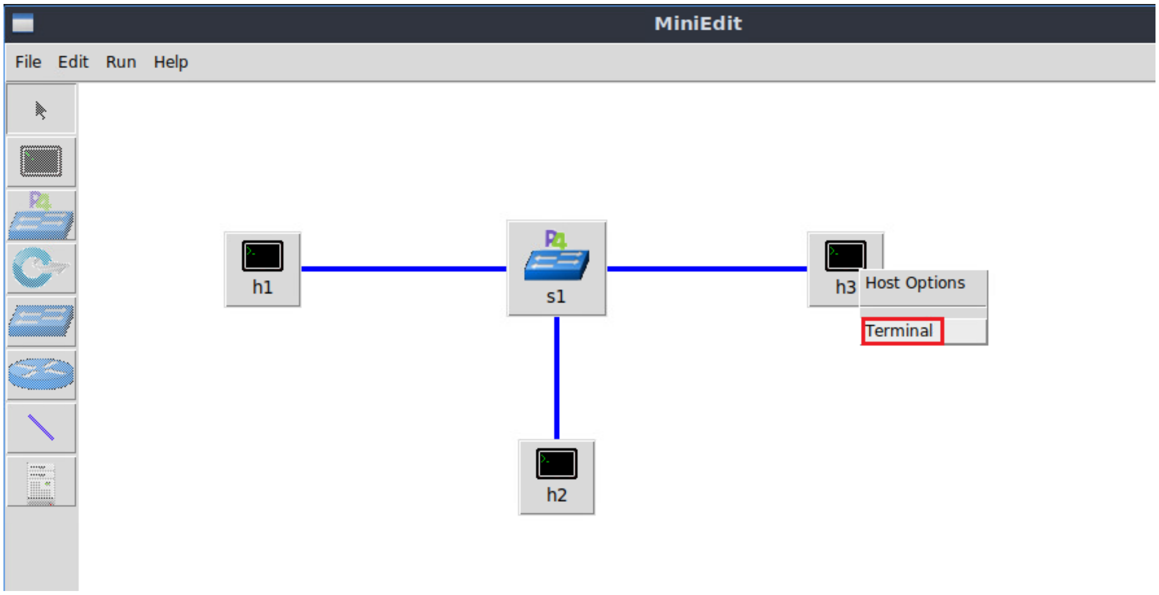
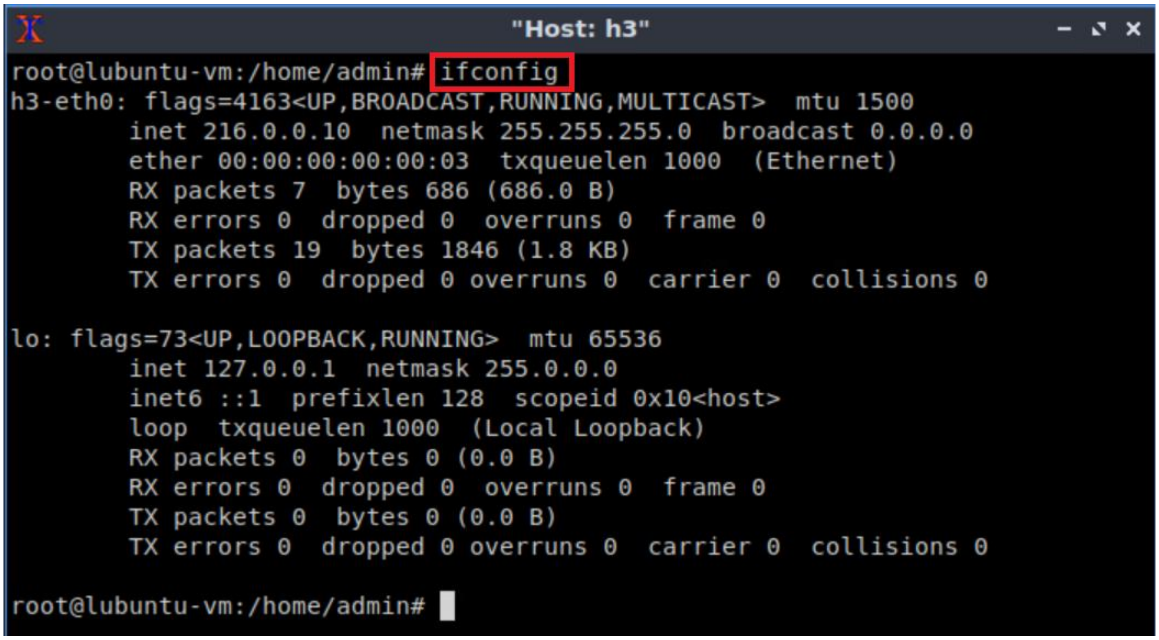


Figure 10. Opening a terminal on host h3.

Step 6. Verify the interfaces’ configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h3-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 216.0.0.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:03  txqueuelen 1000  (Ethernet)
        RX packets 7  bytes 686 (686.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19  bytes 1846 (1.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin#
```

Figure 11. Verifying the configuration host h3 interfaces.

3 Creating a P4 program that performs stateful packet filtering

This section demonstrates how to implement a stateful packet filter in P4 using registers. The stateful packet filter will be applied to ICMP. First, you will load the programming environment. Then, you will define the headers to parse ICMP. Afterwards, you will create the P4 tables that implement the filtering policies. You will also implement the registers to store the state of the flow. The flow ID is produced by hashing the source and the destination IPv4 addresses. This flow ID is used as an index for the register array.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the icon located on the desktop. Alternatively, click on the terminal icon in taskbar located in the lower left-hand side.

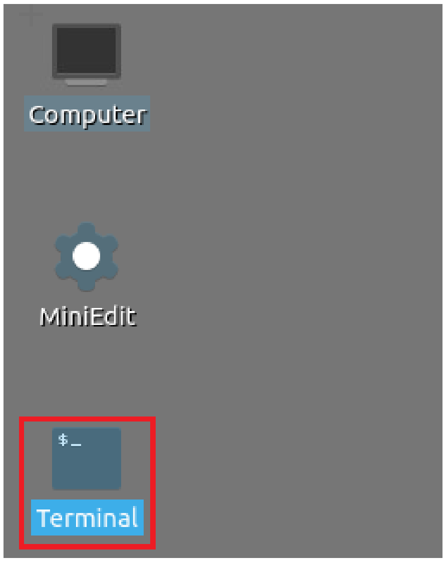
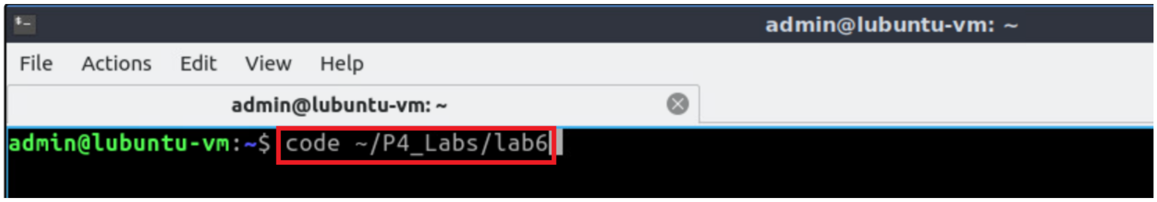


Figure 12. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab6
```

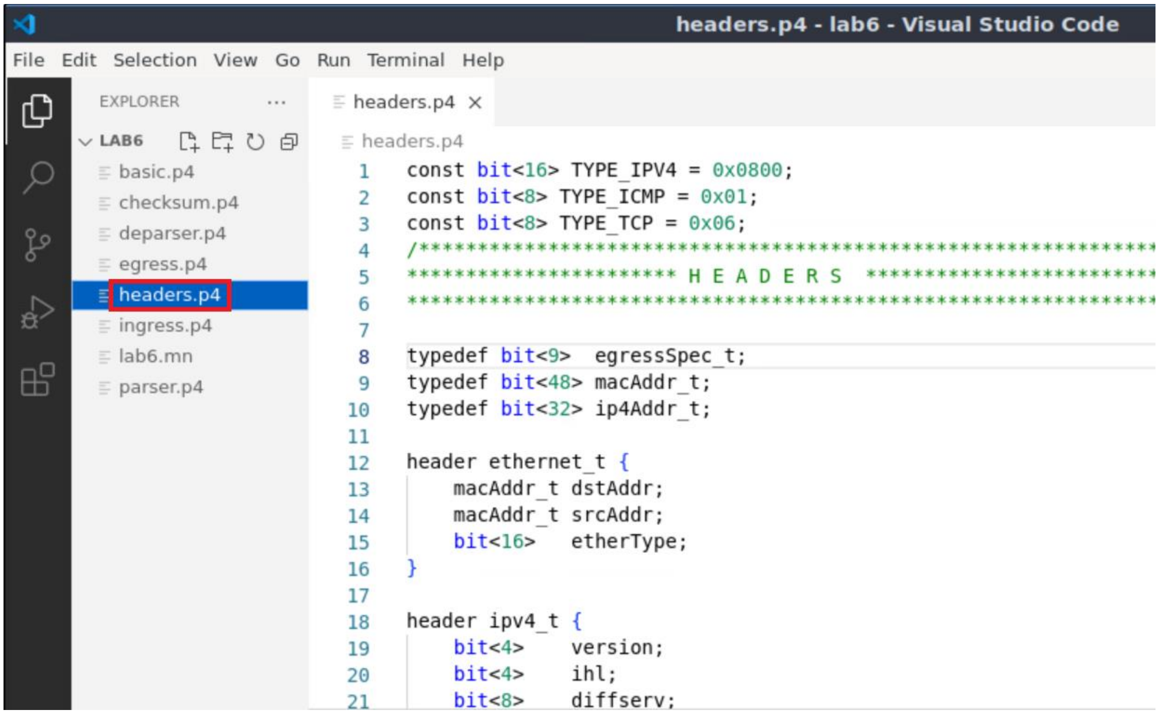


```
admin@lubuntu-vm:~$ code ~/P4_Labs/lab6
```

Figure 13. Launching the editor and opening the lab6 directory.

3.2 Defining the ICMP headers

Step 1. Click on the *headers.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.

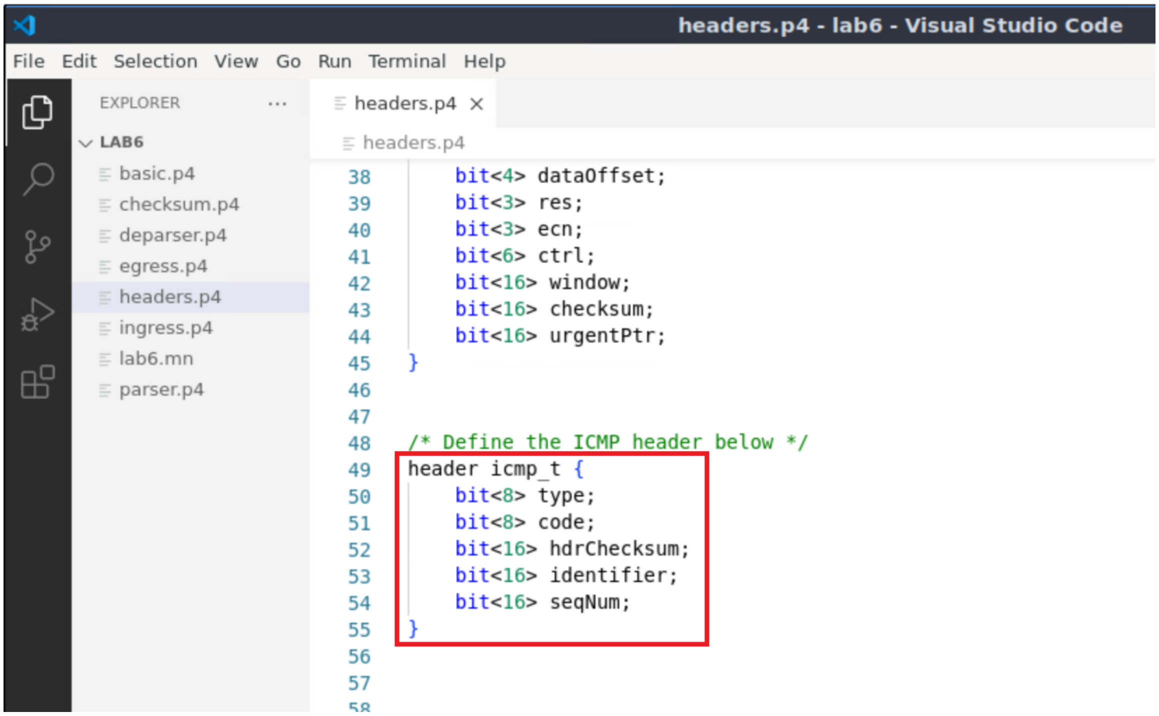


```
const bit<16> TYPE_IPV4 = 0x0800;
const bit<8> TYPE_ICMP = 0x01;
const bit<8> TYPE_TCP = 0x06;
/*************************************************************
********************** H E A D E R S  ***********************
*************************************************************

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 ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
```

Figure 14. Inspecting the *headers.p4* file.

Step 2. Define the ICMP header by adding the code shown below.

```
header icmp_t {
        bit<8> type;
        bit<8> code;
        bit<16> hdrChecksum;
        bit<16> identifier;
        bit<16> seqNum;
}
```



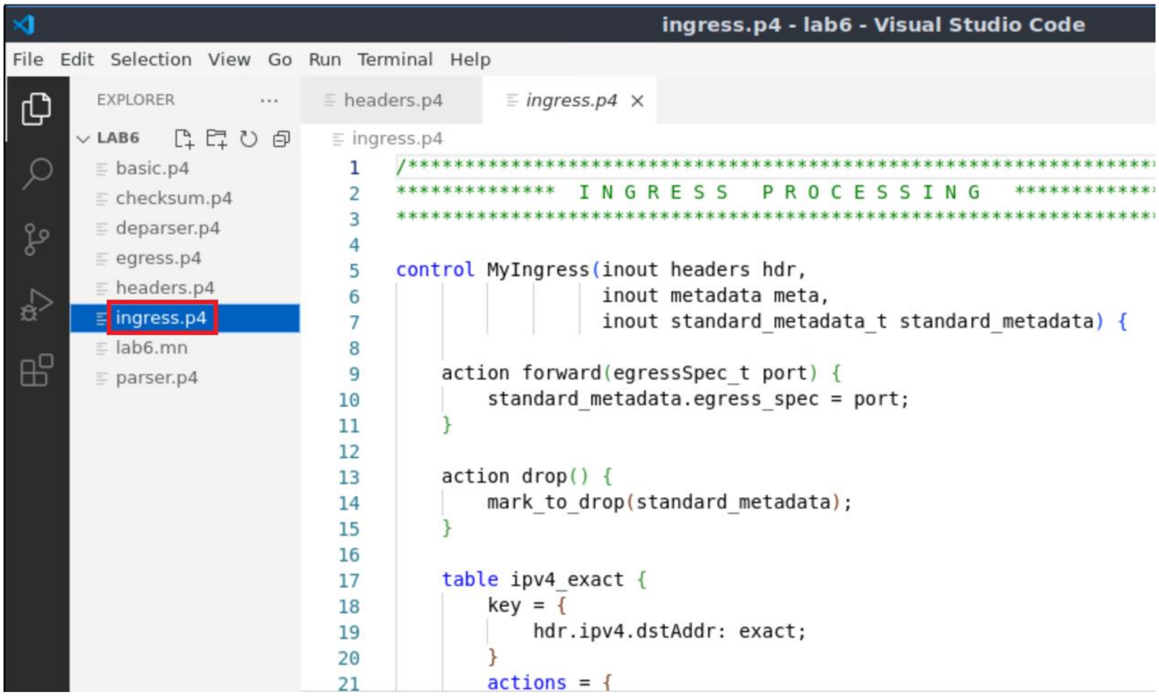
```
    bit<4> dataOffset;
    bit<3> res;
    bit<3> ecn;
    bit<6> ctrl;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}


/* Define the ICMP header below */
header icmp_t {
    bit<8> type;
    bit<8> code;
    bit<16> hdrChecksum;
    bit<16> identifier;
    bit<16> seqNum;
}
```

Figure 15. Defining the ICMP header type.

Step 3. Save the changes to the file by pressing `Ctrl + s`.

3.3 Implementing an ICMP stateful packet filter

Step 1. Click on the *ingress.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.



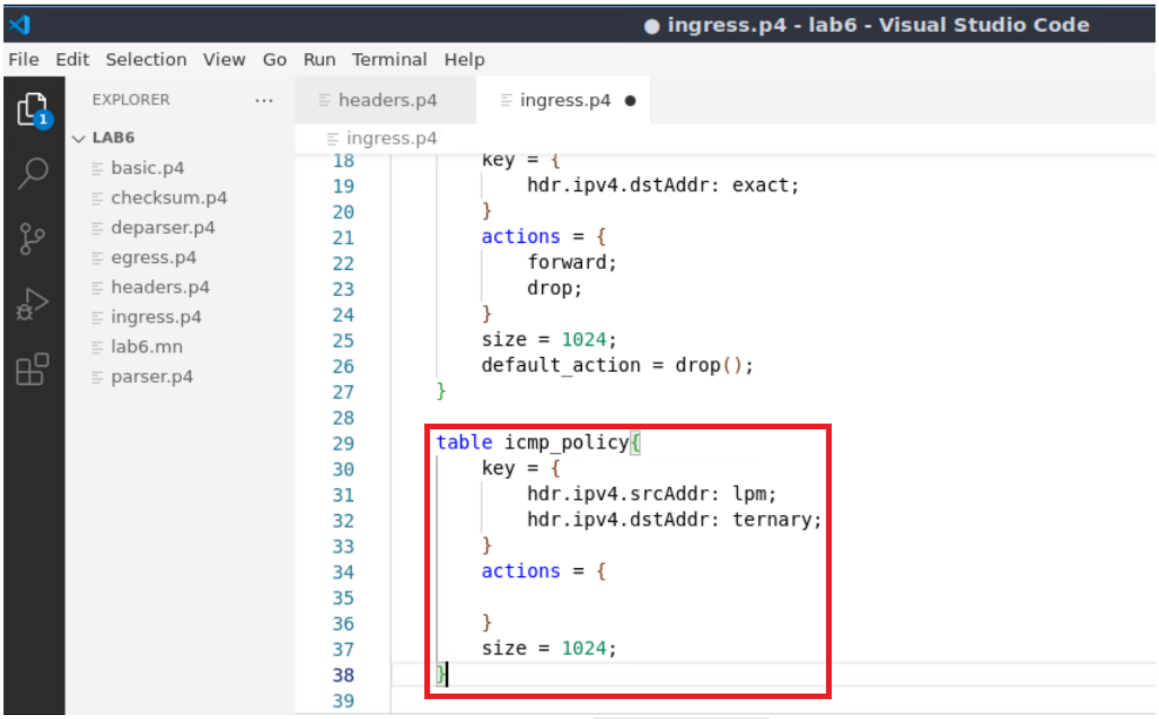
```
/*********************************************************************
**************  I N G R E S S   P R O C E S S I N G   *************
*********************************************************************

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action forward(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table ipv4_exact {
        key = {
            hdr.ipv4.dstAddr: exact;
        }
        actions = {
```

Figure 16. Inspecting the *ingress.p4* file.

We can see that the *ingress.p4* declares a control block named `MyIngress`. Inside this block, a table `ipv4_exact` is defined which matches on the destination IP address and can invoke the `forward` action to forward the packet out of a port, or the `drop` action to drop the packet.

Step 2. Now, we need to define a table that stores the rules for enforcing the ICMP policy. The policy allows the administrator to specify the hosts that can send ICMP packets to destination hosts. The following code implements a table that performs a Longest Prefix Match (LPM) and ternary match on the source and destination IP addresses of the packet. Insert the code below inside the *MyIngress* control block.

```
table icmp_policy {
    key = {
         hdr.ipv4.srcAddr: lpm;
         hdr.ipv4.dstAddr: ternary;
    }
    actions = {

    }
    size = 1024;
}
```



```
        key = {
            hdr.ipv4.dstAddr: exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    table icmp_policy{
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }
```

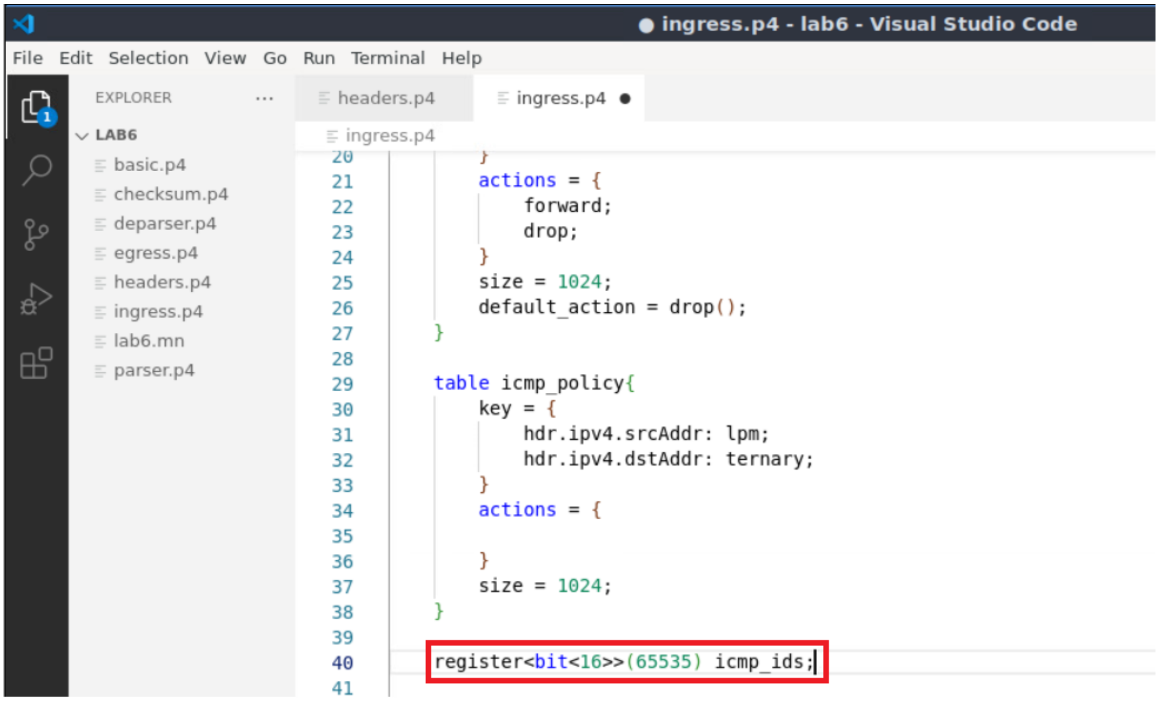
Figure 17. Defining the `icmp_policy` table.

Consider the figure above. The ICMP policy can specify which source hosts are able to ping (issue an ICMP request to) which destination hosts.

In order to make the ICMP policy flexible, the `icmp_policy` table implements `lpm` and `ternary` matches on the source and destination IP addresses so that network administrator can enforce the policy on subnets within a single rule. Note that BMv2 compiler (p4c) does not allow a table to have more than one LPM key field, thus, the ternary matching is used[5].

Step 3. The ICMP policy needs to allow ICMP replies corresponding to ICMP requests that match the enforced policy. Since ICMP requests and their corresponding replies have the same 16-bit identifier in their headers, this identifier will be stored in the switch. The following code defines the register `icmp_ids` that will store the ICMP identifiers. The register can store up to 65536 (0-65535) ICMP identifiers. Insert the code below inside the *MyIngress* control block.

```
register<bit<16>>(65535) icmp_ids;
```

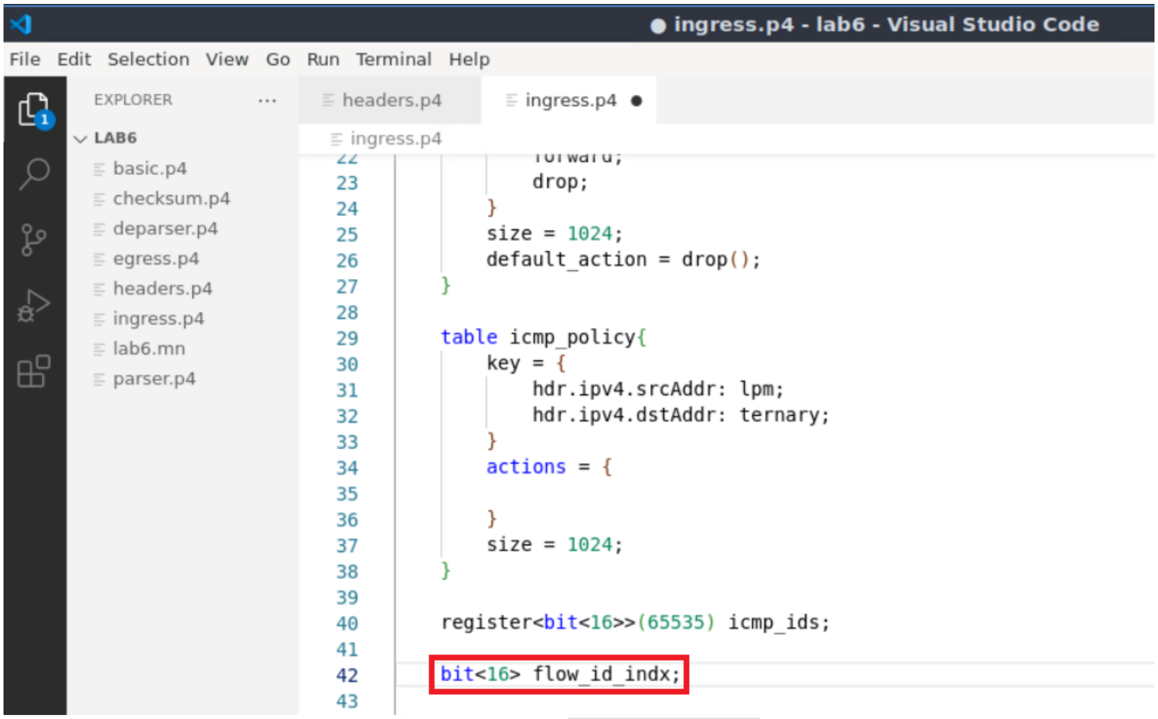


```
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    table icmp_policy{
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;
```

Figure 18. Defining the register stateful element to store ICMP identifiers.

Step 4. Insert the code below to define a 16-bit variable to store the index (`flow_id_indx`) where the ICMP flow will be saved in the register.

```
bit<16> flow_id_indx;
```

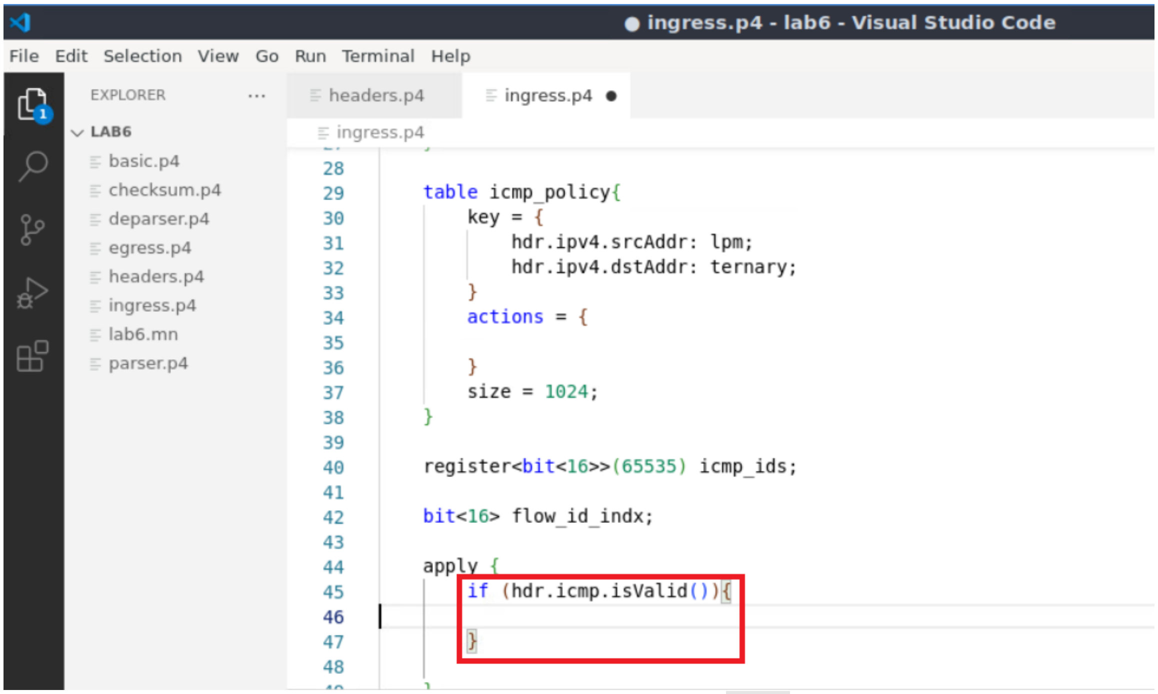


```
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    table icmp_policy{
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;
```

Figure 19. Defining the `flow_id_indx` variable.

Step 5. Add the following code inside the *apply* block of the *MyIngress* Control to check if the packet is an ICMP packet. The *apply* block defines the sequential flow of packet processing. It is required in every control block, otherwise the program will not compile. It describes the sequence of tables to be invoked, in addition to other packet processing instructions.

```
if (hdr.icmp.isValid()) {

}
```

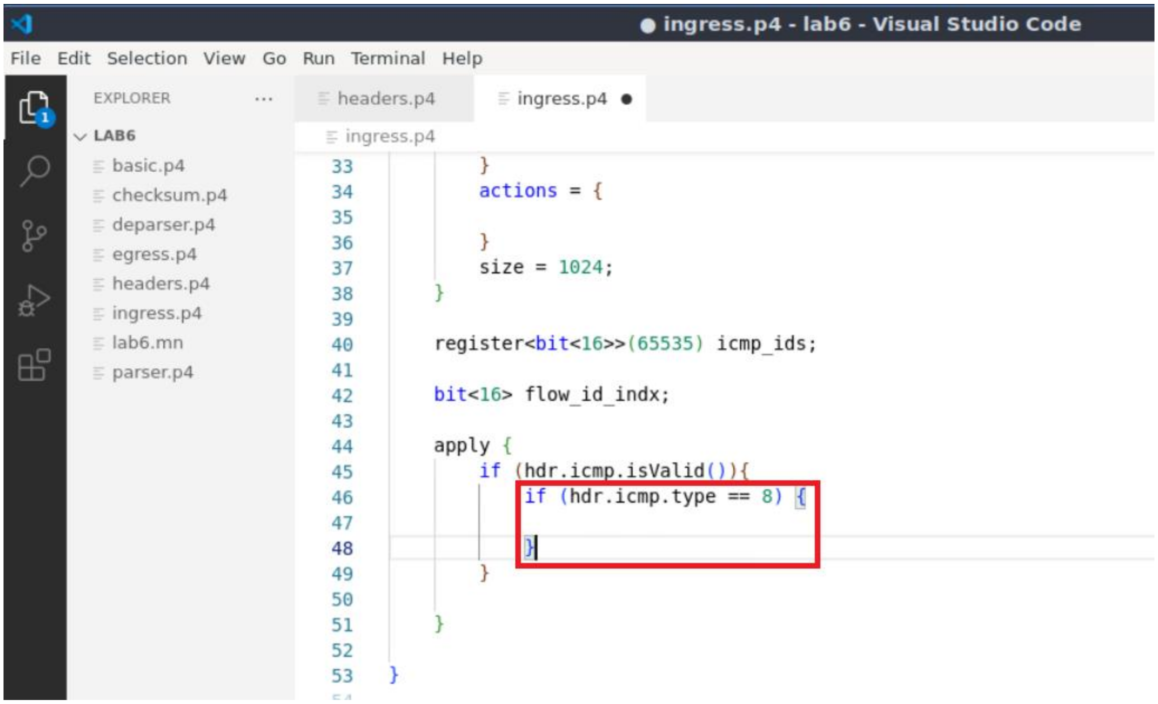


```
table icmp_policy{
    key = {
        hdr.ipv4.srcAddr: lpm;
        hdr.ipv4.dstAddr: ternary;
    }
    actions = {

    }
    size = 1024;
}

register<bit<16>>(65535) icmp_ids;

bit<16> flow_id_indx;

apply {
    if (hdr.icmp.isValid()){

    }
```

Figure 20. Checking the validity of the `icmp` header.

Step 6. Now, you need to check if the ICMP packet is of type REQUEST so that it can be checked in the `icmp_policy` table. Insert the code below inside the `if` statement which checks the validity of the `icmp` header.

```
if (hdr.icmp.type == 8) {

}
```



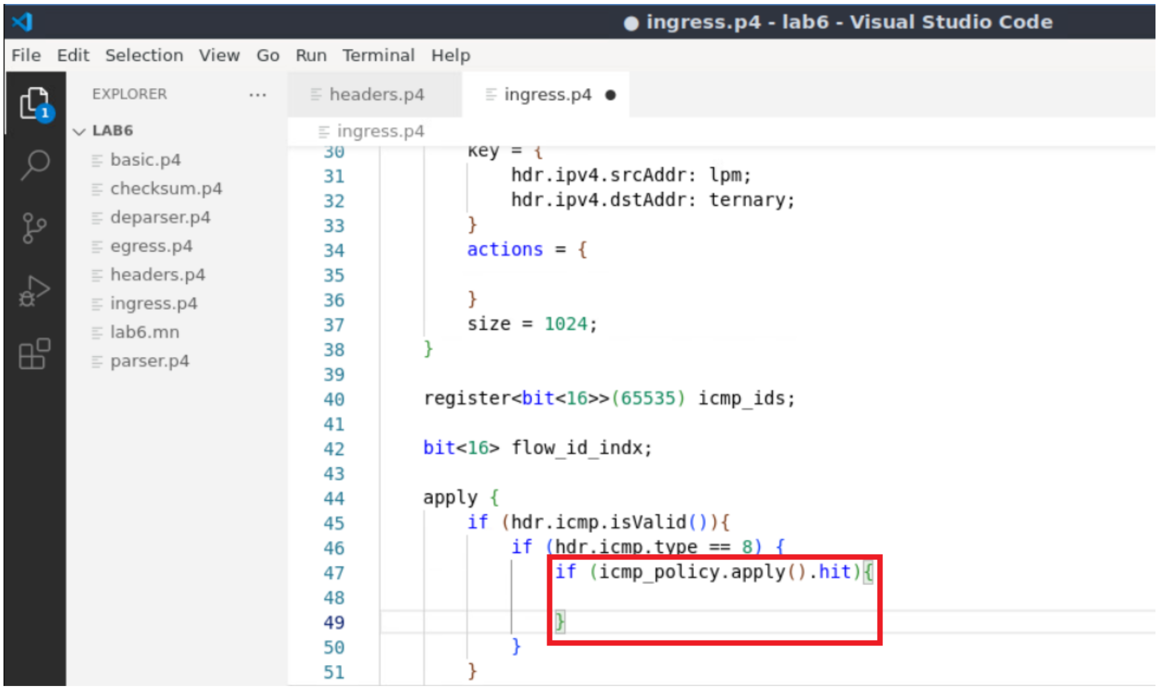
```
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {

            }
        }

    }
}
```

Figure 21. Checking if the ICMP packet is a request.

The header field `hdr.icmp.type` defines the type of the ICMP packet. When it is equal to `8`, the ICMP packet is of type REQUEST.

Step 7. Once an ICMP REQUEST packet arrives at the switch, the latter needs to check if it matches an entry in the defined `icmp_policy` table. Insert the code below to check if the ICMP REQUEST packet matches any rule in the `icmp_policy` table.

```
if (icmp_policy.apply().hit) {

}
```



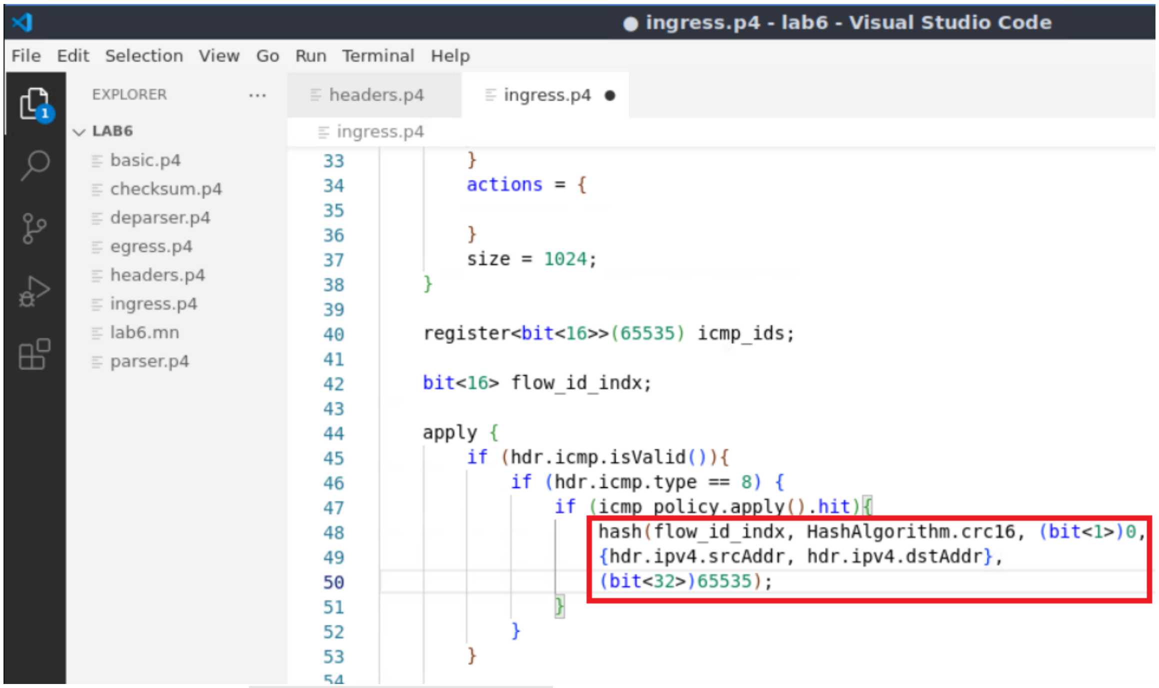
```
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){

                }
            }
        }
```

Figure 22. Checking if the ICMP request packet matches any of the enforced rules within the `icmp_policy` table.

The `icmp_policy` table is applied using the `icmp_policy.apply()` and the if statement which checks if the packet matches any of the installed rules within the table using the `icmp_policy.apply().hit`.

Step 8. Insert the code below to compute the flow identifier index that will be used to store the ICMP identifier in the register.

```
hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
     {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
     (bit<32>)65535);
```



```
    }
    actions = {

    }
    size = 1024;
}

register<bit<16>>(65535) icmp_ids;

bit<16> flow_id_indx;

apply {
    if (hdr.icmp.isValid()){
        if (hdr.icmp.type == 8) {
            if (icmp_policy.apply().hit){
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                (bit<32>)65535);
            }
        }
    }
```

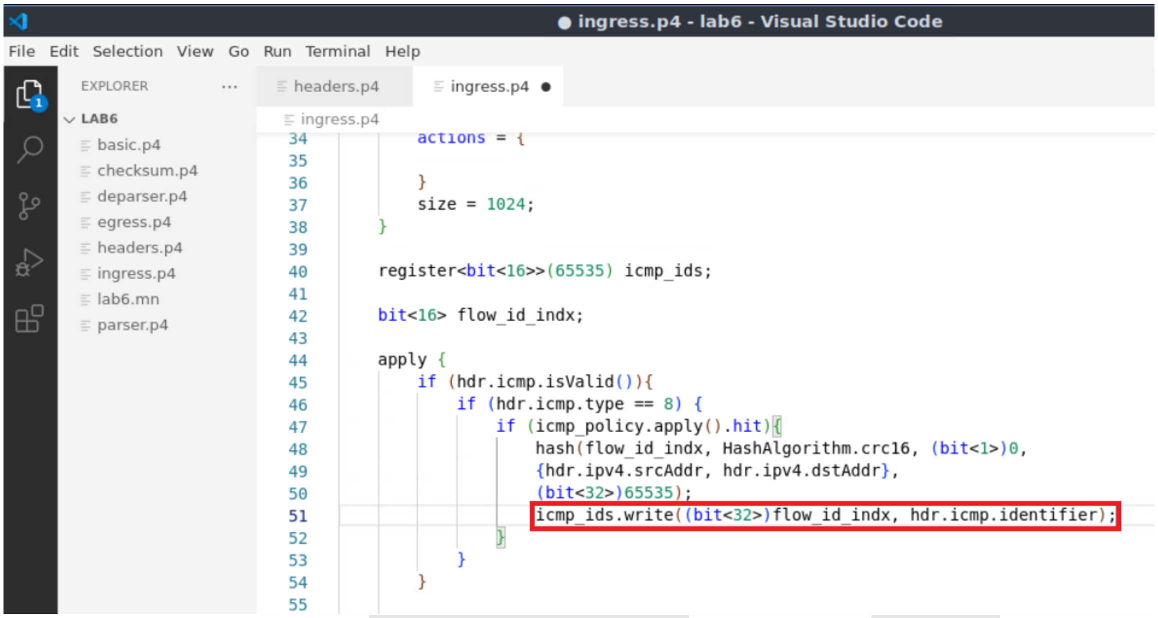
Figure 23. Using the `HashAlgorithm.crc16` to compute the index of the register where the ICMP identifier will be stored.

The code in the figure above hashes flows based on their source and destination IP addresses. The hash function produces a 16-bits output using the following parameters:

- `flow_id_indx`: The variable used to store the output.
- `HashAlgorithm.crc16`: the hash algorithm.
- `(bit<1>)0`: the minimum (or base) value produced by the hash algorithm.
- `hdr.ipv4.srcAddr` and `hdr.ipv4.dstAddr`: the data to be hashed.
- `(bit<32>)65535`: the maximum value produced by the hash algorithm.

Step 9. Insert the code below to store the ICMP identifier of the packet in the register `icmp_ids`.

```
icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
```



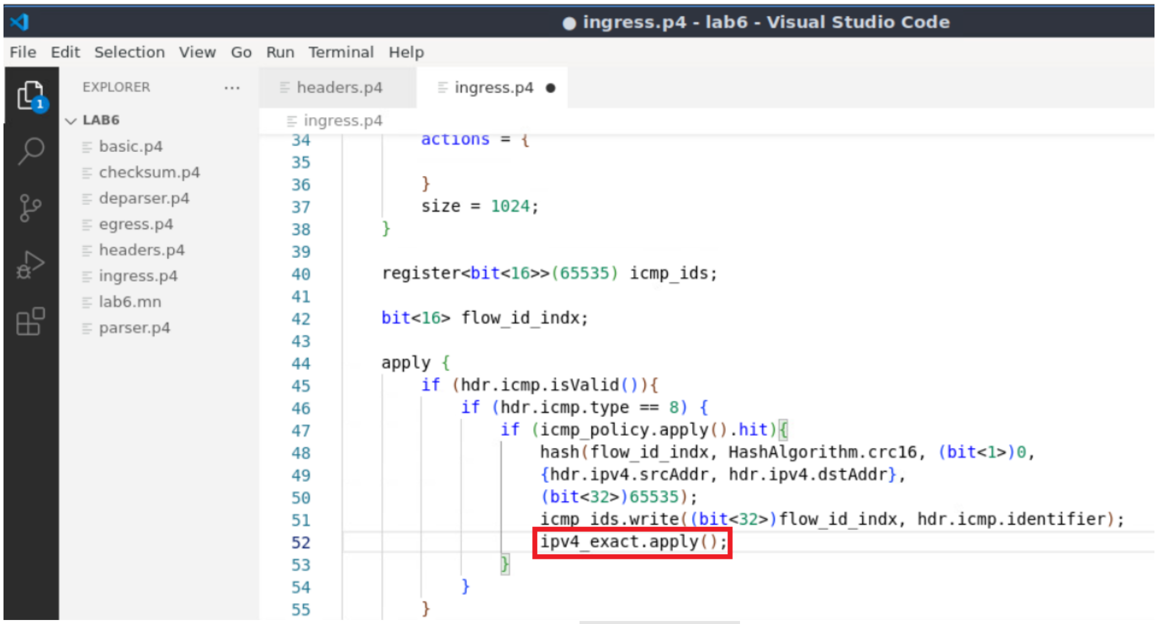
```
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                }
            }
        }
```

Figure 24. Storing `hdr.icmp.idenitifier` in the register `icmp_ids`.

The function `icmp_ids.write` stores the value of the `hdr.icmp.identifier` at the index `flow_id_indx` of the `icmp_ids` register. The index of the register must be a 32-bit value, thus, `flow_id_indx` is cast to a 32-bit value.

Step 10. Insert the code below to apply the `ipv4_exact` table and forward the packet to the destination host.

```
ipv4_exact.apply();
```

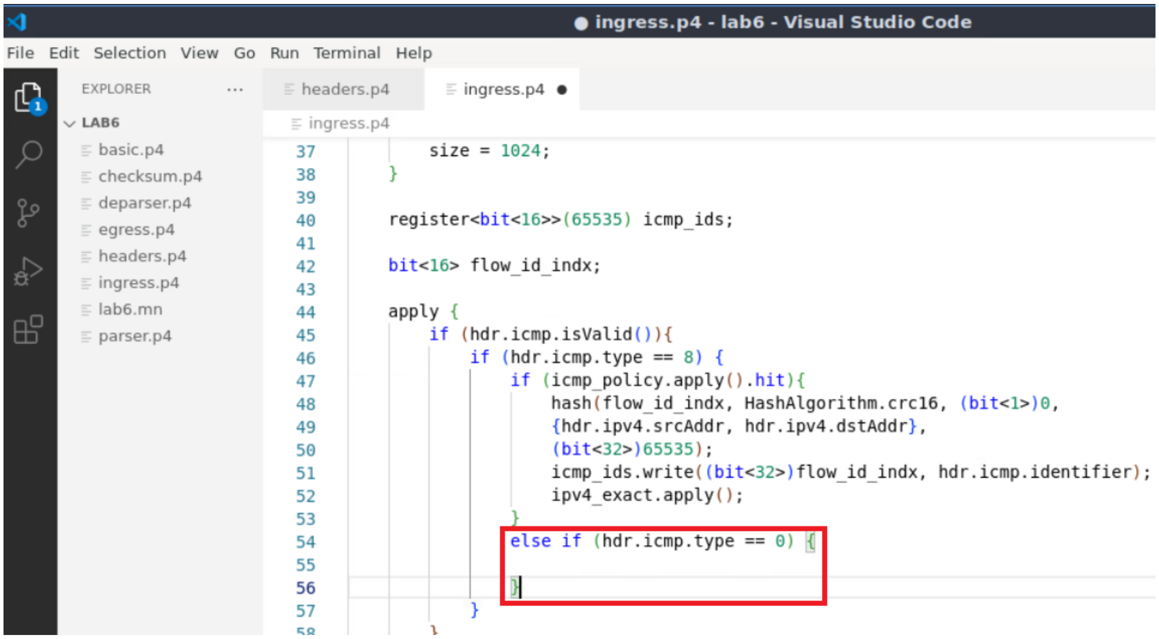


```
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
            }
        }
```

Figure 25. Applying `ipv4_exact` table.

Step 11. Insert the code below to check if the ICMP packet is a REPLY.

```
else if (hdr.icmp.type == 0) {
}
```



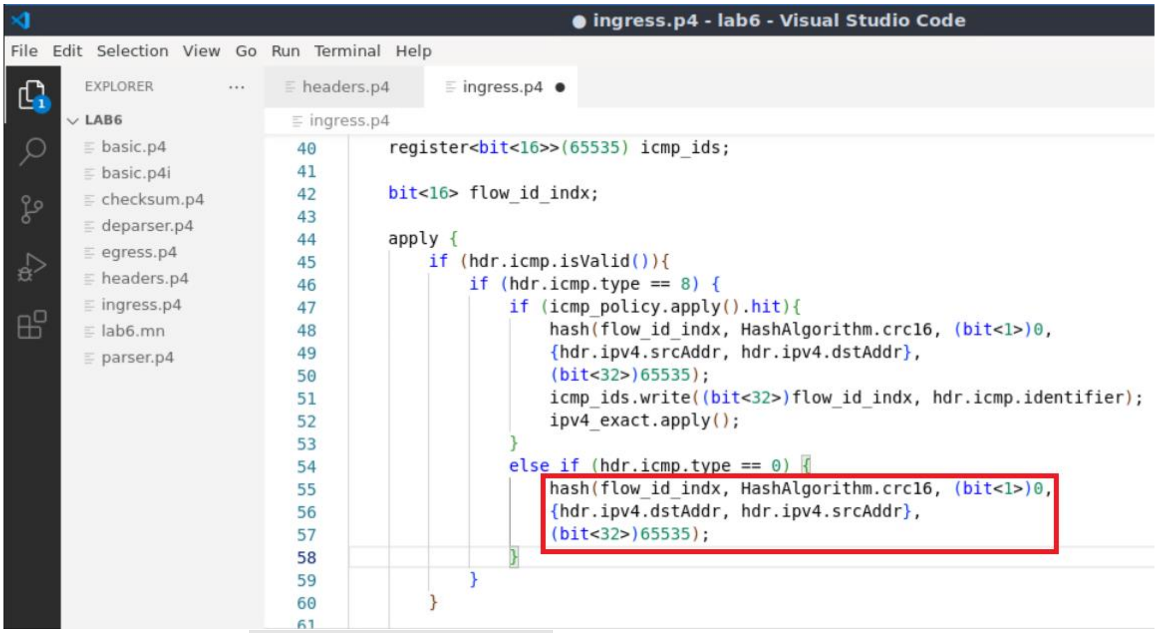
```
        size = 1024;
    }

    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
                else if (hdr.icmp.type == 0) {

                }
            }
        }
```

Figure 26. Checking if the ICMP packet is a reply.

When the header field `hdr.icmp.type` is equal to `0`, then the ICMP packet is a REPLY.

Step 12. Insert the code below to compute the flow identifier index that will be used to retrieve the ICMP identifier from the register.

```
hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
      {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
      (bit<32>)65535);
```



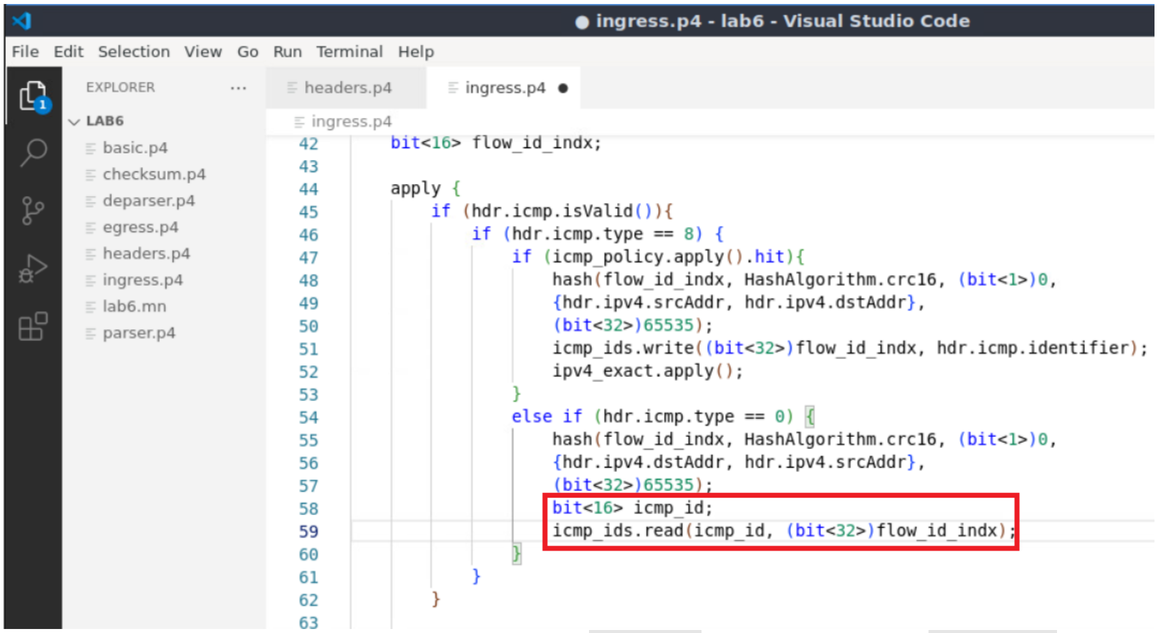
```
    register<bit<16>>(65535) icmp_ids;

    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
                else if (hdr.icmp.type == 0) {
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                }
            }
        }
```

Figure 27. Using the `HashAlgorithm.crc16` to compute the index of the register where the ICMP identifier will be retrieved.

The code in the figure above hashes flows based on their destination and source IP addresses. Note how the source and destination IP addresses are reversed in the hash function so that the retrieved index matches as the one used in the associated ICMP REQUEST.

Step 13. Insert the code below to retrieve the ICMP identifier from the register `icmp_ids` and store it inside `icmp_id` variable.

```
bit<16> icmp_id;
icmp_ids.read(flow_id, (bit<32>flow_id_indx);
```



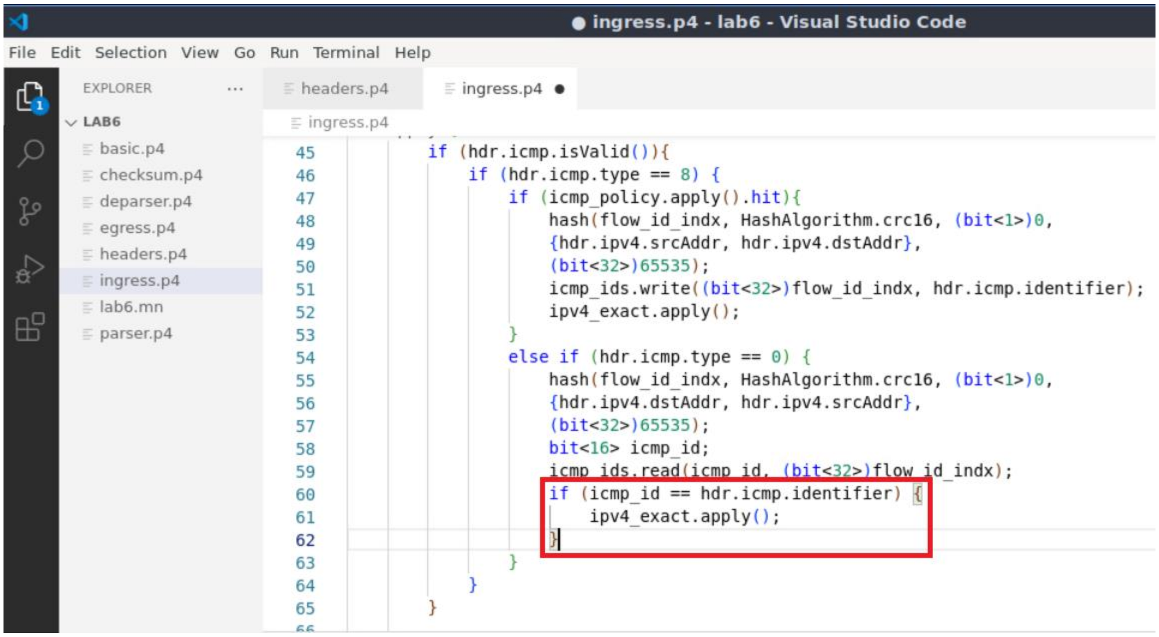
```
    bit<16> flow_id_indx;

    apply {
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
                else if (hdr.icmp.type == 0) {
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                    bit<16> icmp_id;
                    icmp_ids.read(icmp_id, (bit<32>)flow_id_indx);
                }
            }
        }
```

Figure 28. Retrieving the ICMP identifier (`icmp_id`) from the register `icmp_ids`.

The function `icmp_ids.read` retrieves the value at the index `flow_id_indx` of the `icmp_ids` register and stores it in the variable `icmp_id`.

Step 14. Insert the code below to check if the retrieved value `icmp_id` from the register is equal to the ICMP identifier of the ICMP REPLY packet (`hdr.icmp.identifier`) and to forward the packet by applying the `ipv4_exact` table.

```
if (icmp_id == hdr.icmp.identifier) {
        ipv4_exact.apply();
}
```



```
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
                else if (hdr.icmp.type == 0) {
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                    bit<16> icmp_id;
                    icmp_ids.read(icmp_id, (bit<32>)flow_id_indx);
                    if (icmp_id == hdr.icmp.identifier) {
                        ipv4_exact.apply();
                    }
                }
            }
        }
```

Figure 29. Forwarding ICMP REPLY packets that match the ICMP requests packets conforming to the enforced ICMP policy.

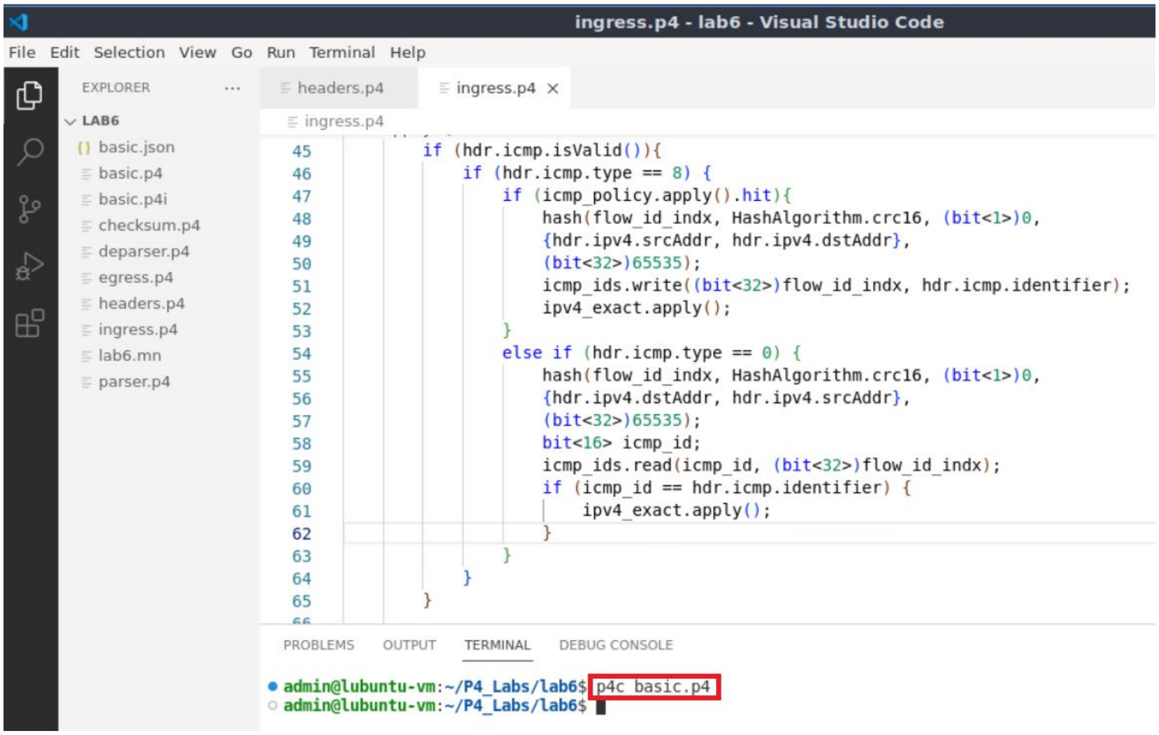
Step 15. Save the changes to the file by pressing `Ctrl + s`.

4 Loading the P4 program

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside the VS Code to compile the program.

```
p4c basic.p4
```



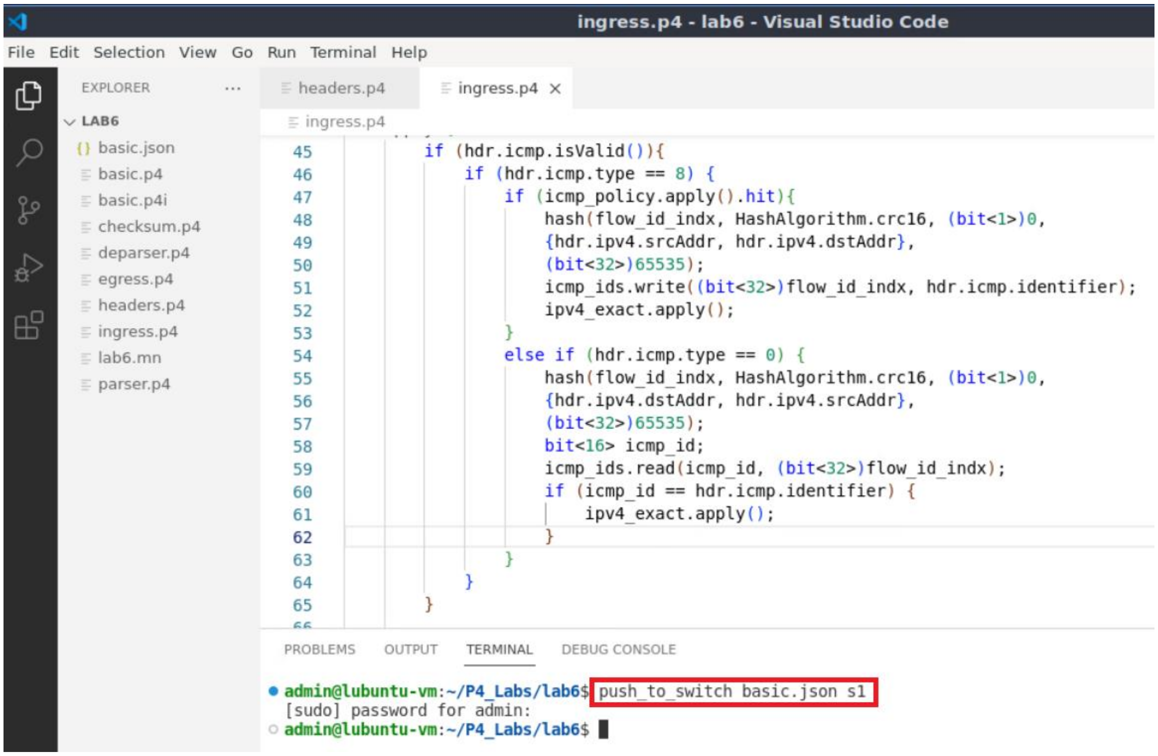
```
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
                else if (hdr.icmp.type == 0) {
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                    bit<16> icmp_id;
                    icmp_ids.read(icmp_id, (bit<32>)flow_id_indx);
                    if (icmp_id == hdr.icmp.identifier) {
                        ipv4_exact.apply();
                    }
                }
            }
        }
```

```
admin@lubuntu-vm:~/P4_Labs/lab6$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab6$
```

Figure 30. Compiling a P4 program.

Step 2. Type the command below in the terminal panel to push the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the p4c compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



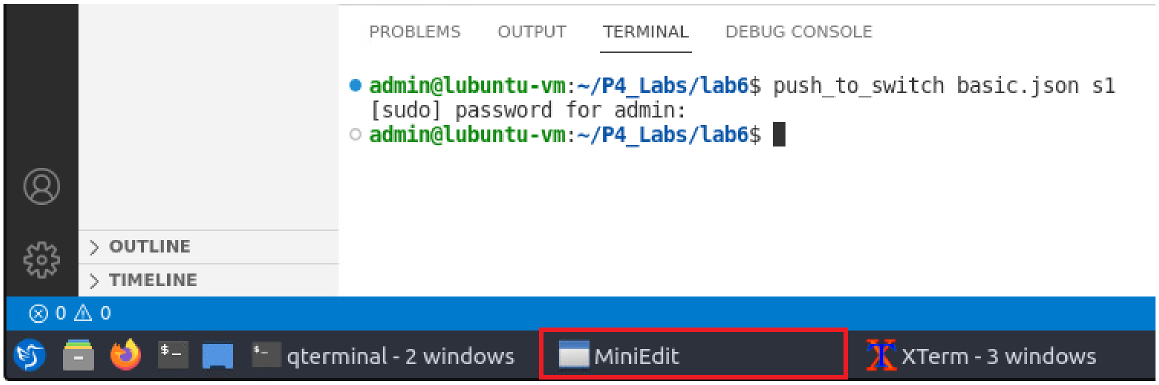
```
        if (hdr.icmp.isValid()){
            if (hdr.icmp.type == 8) {
                if (icmp_policy.apply().hit){
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                    icmp_ids.write((bit<32>)flow_id_indx, hdr.icmp.identifier);
                    ipv4_exact.apply();
                }
                else if (hdr.icmp.type == 0) {
                    hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                    bit<16> icmp_id;
                    icmp_ids.read(icmp_id, (bit<32>)flow_id_indx);
                    if (icmp_id == hdr.icmp.identifier) {
                        ipv4_exact.apply();
                    }
                }
            }
        }
```

```
admin@lubuntu-vm:~/P4_Labs/lab6$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab6$
```

Figure 31. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.



```
admin@lubuntu-vm:~/P4_Labs/lab6$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab6$
```

Figure 32. Maximizing the MiniEdit window.

Step 2. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

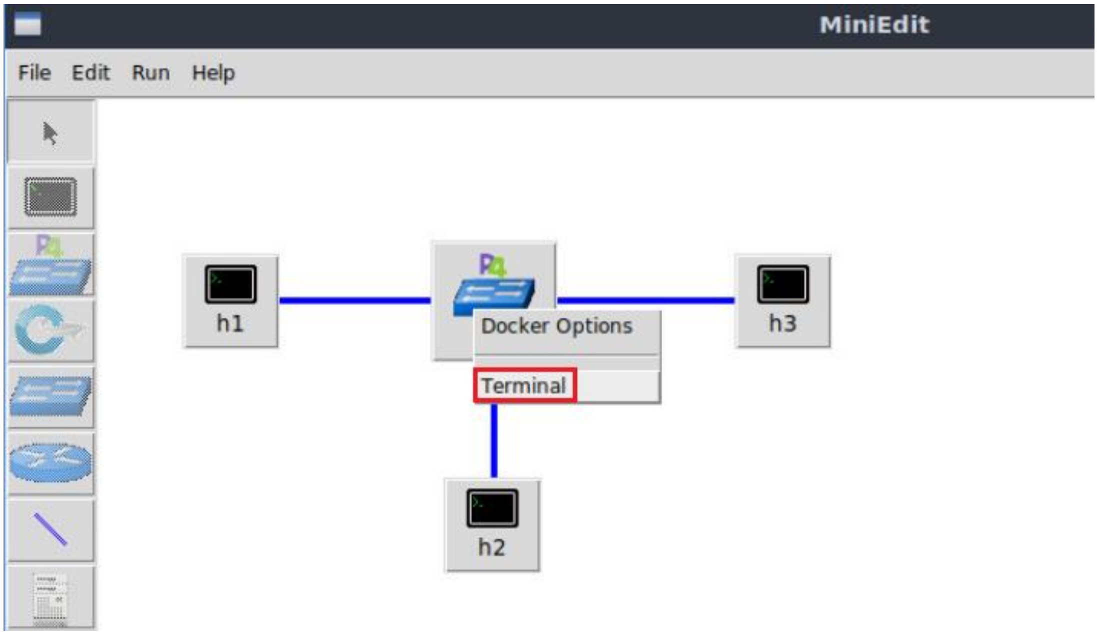
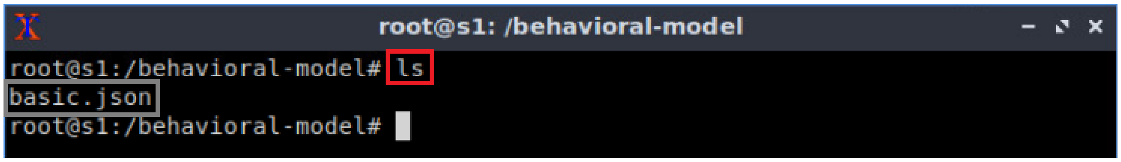


Figure 33. Opening switch s1 terminal.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

Step 3. Issue the following command on switch s1 terminal to inspect the content of the current folder.

```
ls
```



```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

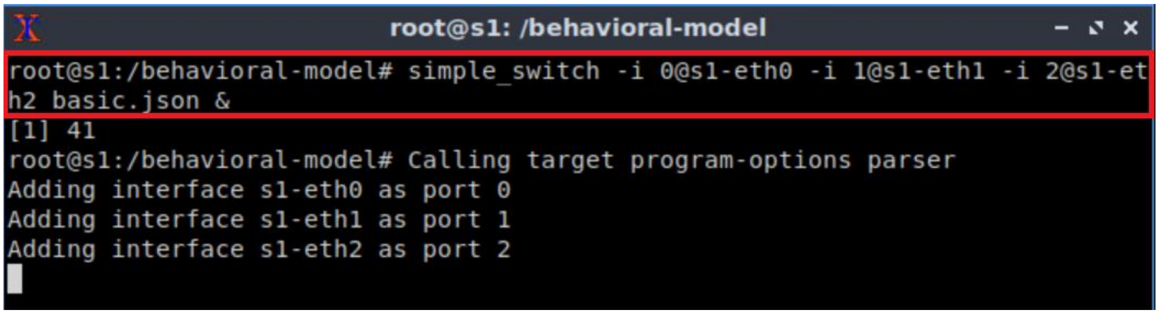
Figure 34. Displaying the content of the current directory in the switch s1.

We can see that the switch contains the *basic.json* file that was pushed previously after compiling the P4 program.

4.3 Mapping P4 program's ports

Step 1. Start the switch daemon and map the logical interfaces (i.e., ports) to the switch's interfaces by issuing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```



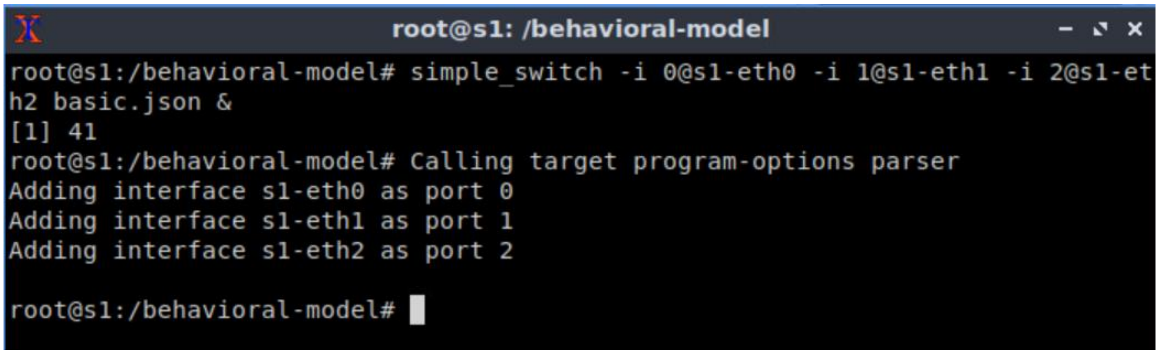
```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 41
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 35. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

5 Testing and verifying the P4 program

5.1 Configuring the policy rules

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

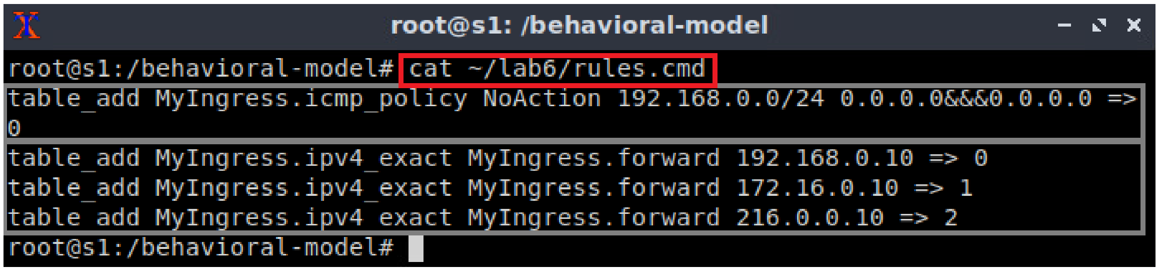


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 41
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model# 
```

Figure 36. Returning to switch s1 CLI.

Step 2. Type the command below to inspect the content of the `rules.cmd` file. The file contains the rules which will be inserted to the switch at runtime. `cat` utility prints the content into the standard output.

```
cat ~/lab6/rules.cmd
```



```
root@s1:/behavioral-model# cat ~/lab6/rules.cmd
table_add MyIngress.icmp_policy NoAction 192.168.0.0/24 0.0.0.0&&&0.0.0.0 =>
0
table_add MyIngress.ipv4_exact MyIngress.forward 192.168.0.10 => 0
table_add MyIngress.ipv4_exact MyIngress.forward 172.16.0.10 => 1
table_add MyIngress.ipv4_exact MyIngress.forward 216.0.0.10 => 2
root@s1:/behavioral-model#
```

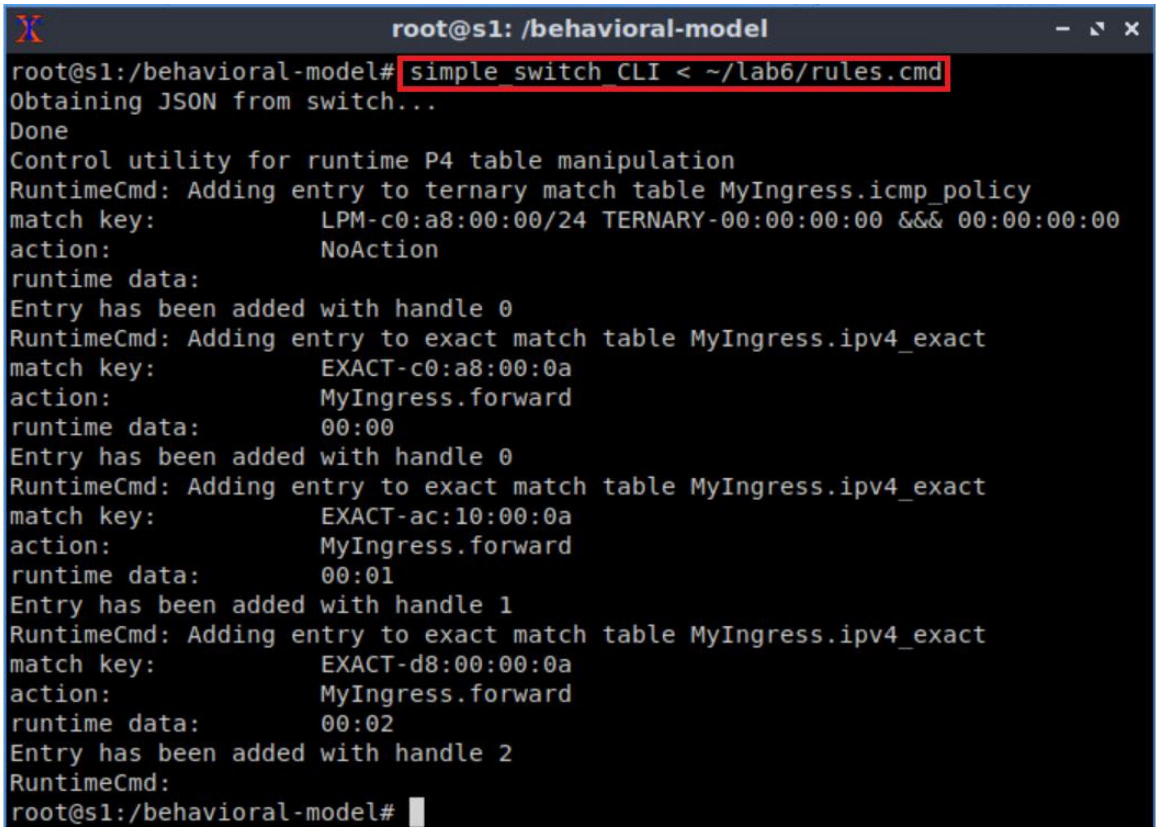
Figure 37. Inspecting the rules.

In the figure above, the first rule populates the `icmp_policy` table. `NoAction` keyword is used when we do not want to execute any specific action upon matching. `192.168.0.10/24` is the LPM key which matches packets coming from the internal network. `0.0.0.0&&&0.0.0.0` is the ternary key which matches on any packet (similar to 0.0.0.0/0). Thus, the `icmp_policy` table will hit on any packet originated from the internal network regardless of its destination IP address. Note that any ICMP packet originated from the external network, or the DMZ will not match the policy, and consequently, will be dropped.

The last three rules populate the `ipv4_exact` table with the forwarding rules.

Step 3. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab6/rules.cmd
```



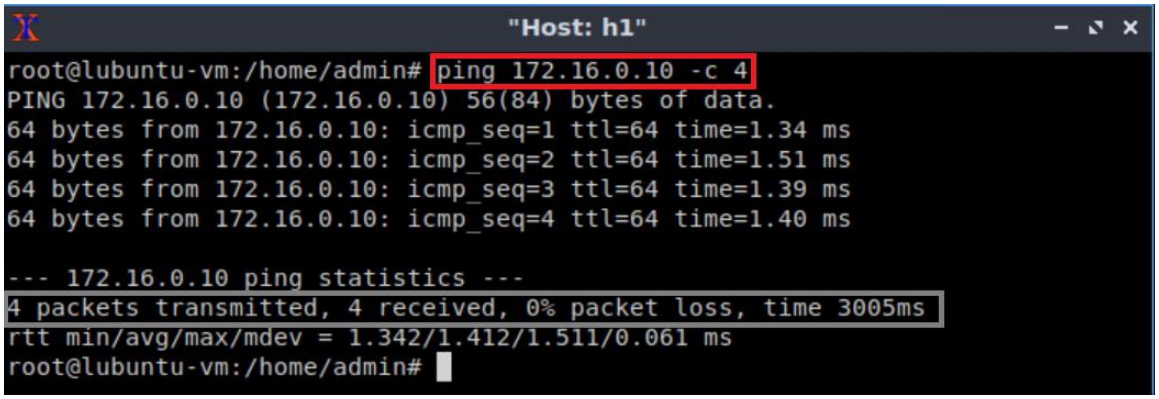
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab6/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to ternary match table MyIngress.icmp_policy
match key:           LPM-c0:a8:00:00/24 TERNARY-00:00:00:00 &&& 00:00:00:00
action:              NoAction
runtime data:
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-c0:a8:00:0a
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-ac:10:00:0a
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-d8:00:00:0a
action:              MyIngress.forward
runtime data:        00:02
Entry has been added with handle 2
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 38. Pushing the table entries to the switch.

5.2 Testing the P4 program

Step 1. On h1 terminal, type the command below to send four ICMP requests to h2.

```
ping 172.16.0.10 -c 4
```



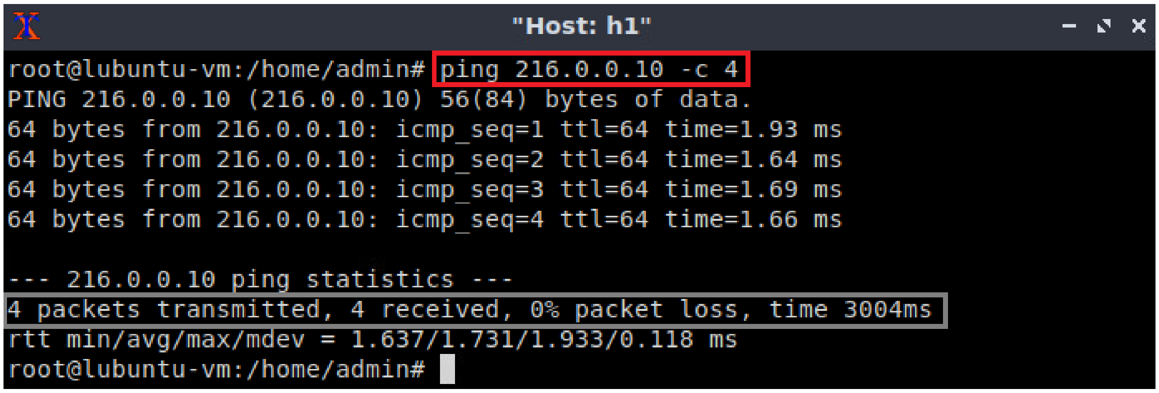
```
root@lubuntu-vm:/home/admin# ping 172.16.0.10 -c 4
PING 172.16.0.10 (172.16.0.10) 56(84) bytes of data.
64 bytes from 172.16.0.10: icmp_seq=1 ttl=64 time=1.34 ms
64 bytes from 172.16.0.10: icmp_seq=2 ttl=64 time=1.51 ms
64 bytes from 172.16.0.10: icmp_seq=3 ttl=64 time=1.39 ms
64 bytes from 172.16.0.10: icmp_seq=4 ttl=64 time=1.40 ms

--- 172.16.0.10 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.342/1.412/1.511/0.061 ms
root@lubuntu-vm:/home/admin#
```

Figure 39. Pinging h2 (DMZ) from h1 (internal network).

The figure above shows that the four ICMP packets were received successfully.

Step 2. On h1 terminal, type the command below to send four ICMP requests to h3.

```
ping 216.0.0.10 -c 4
```



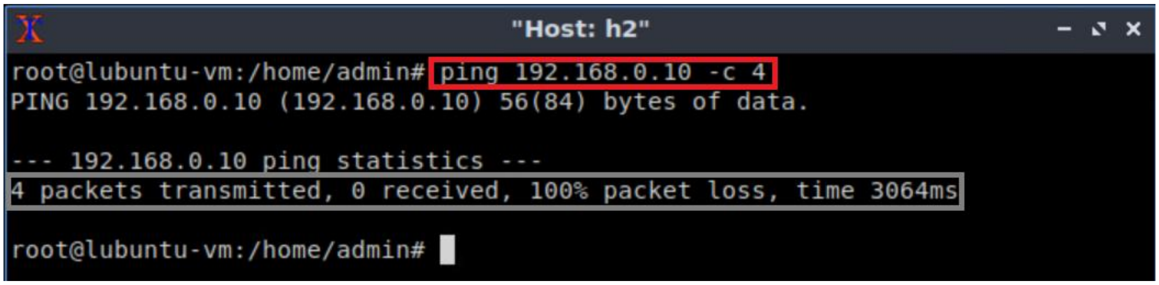
```
root@lubuntu-vm:/home/admin# ping 216.0.0.10 -c 4
PING 216.0.0.10 (216.0.0.10) 56(84) bytes of data.
64 bytes from 216.0.0.10: icmp_seq=1 ttl=64 time=1.93 ms
64 bytes from 216.0.0.10: icmp_seq=2 ttl=64 time=1.64 ms
64 bytes from 216.0.0.10: icmp_seq=3 ttl=64 time=1.69 ms
64 bytes from 216.0.0.10: icmp_seq=4 ttl=64 time=1.66 ms

--- 216.0.0.10 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.637/1.731/1.933/0.118 ms
root@lubuntu-vm:/home/admin# 
```

Figure 40. Pinging h3 (external network) from h1 (internal network).

The figure above shows that the four ICMP packets were received successfully.

Step 3. On h2 terminal, type the command below to send four ICMP requests to h1.

```
ping 192.168.0.10 -c 4
```



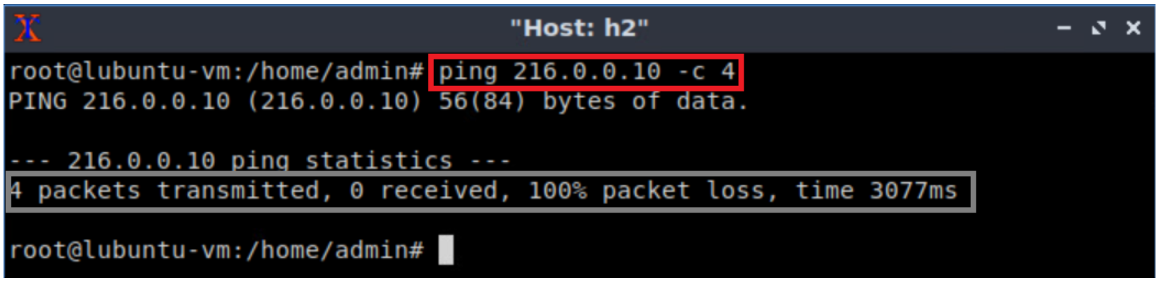
```
root@lubuntu-vm:/home/admin# ping 192.168.0.10 -c 4
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.

--- 192.168.0.10 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3064ms

root@lubuntu-vm:/home/admin# 
```

Figure 41. Pinging h1 (internal network) from h2 (DMZ).

The figure above shows that the four ICMP packets were lost. The switch dropped the packets because they did not originate from the internal network.

Step 4. On h2 terminal, type the command below send four ICMP requests to h3.

```
ping 216.0.0.10 -c 4
```



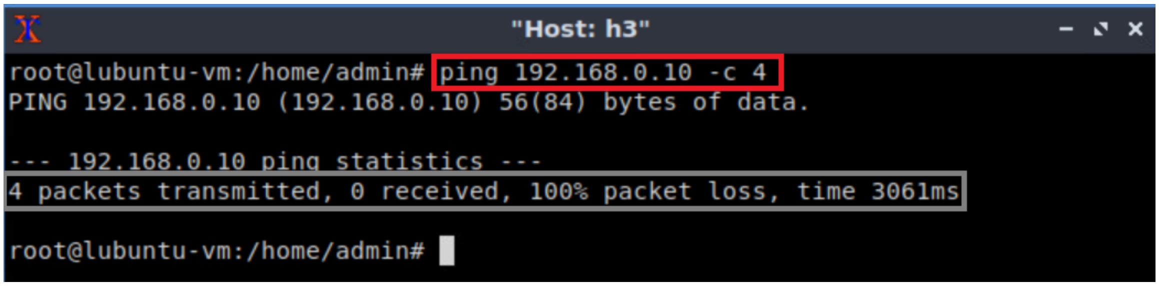
```
root@lubuntu-vm:/home/admin# ping 216.0.0.10 -c 4
PING 216.0.0.10 (216.0.0.10) 56(84) bytes of data.

--- 216.0.0.10 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3077ms

root@lubuntu-vm:/home/admin# 
```

Figure 42. Pinging h3 (external network) from h2 (DMZ).

The figure above shows that the four ICMP packets were lost. The switch dropped the packets because they did not originate from the internal network.

Step 5. On h3 terminal, type the command below to send four ICMP requests to h1.

```
ping 192.168.0.10 -c 4
```



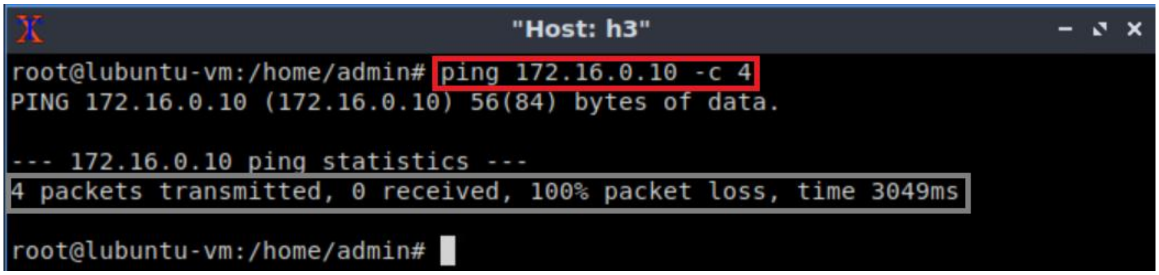
```
root@lubuntu-vm:/home/admin# ping 192.168.0.10 -c 4
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.

--- 192.168.0.10 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3061ms

root@lubuntu-vm:/home/admin#
```

Figure 43. Pinging h1 (internal network) from h3 (external network).

The figure above shows that the four ICMP packets were lost. The switch dropped the packets because they did not originate from the internal network.

Step 6. On h3 terminal, type the command below to send four ICMP requests to h2.

```
ping 172.16.0.10 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 172.16.0.10 -c 4
PING 172.16.0.10 (172.16.0.10) 56(84) bytes of data.

--- 172.16.0.10 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3049ms

root@lubuntu-vm:/home/admin#
```

Figure 44. Pinging h2 (DMZ) from h3 (external network).

The figure above shows that the four ICMP packets were lost. The switch dropped the packets because they did not originate from the internal network.

This concludes lab 6. Stop the emulation and then exit out of MiniEdit.

References

1. M. Rouse, “Packet Filtering.” [Online]. Available: https://tinyurl.com/8z4a2yp6
2. Diyaroy, “Stateless vs Stateful Packet Filtering Firewalls” Online]. Available: https://tinyurl.com/3s2twcdp
3. P4-guide github repository, *“Demo Global Register P4$_{16}$.”* [Online]. Available: https://tinyurl.com/mrytj9ad
4. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.
5. P4lang/behavioral-model github repository*, “The BMv2 simple switch target*.” [Online]. Available: https://tinyurl.com/vrasamm.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 7: Implementing a Stateful Packet Filter for the TCP Protocol

Document Version: 04-18-2023

Contents

Overview

This lab is an introduction to stateful packet filter in P4, a technique by which a network administrator can implement network-based access control. In particular, the lab uses P4 registers to store the state of a connection. The lab further implements a stateful packet filter for Transmission Control Protocol (TCP) via a policy defined by the network administrator.

Objectives

By the end of this lab, students should be able to:

1. Understand stateful packet filters.
2. Understand what registers are used for.
3. Implement stateful packet filters in P4 using registers.
4. Test the defined policy for the stateful packet filter.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client machine.

Device	**Account**	**Password**
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Creating a P4 program that performs stateful packet filtering.
4. Section 4: Loading the P4 program.
5. Section 5: Testing and verifying the P4 program.

1 Introduction

Packet filters control and manage the flow of data across a network by filtering and analyzing outgoing and incoming packets[1]. They are commonly implemented in firewalls or routers to protect networks from unauthorized access and malicious activities. Packet filters can be broadly classified into two categories: stateless and stateful.

Stateless packet filters operate on a per-packet basis, examining each packet individually without considering any previous packets[2]. Stateless filters use predefined rules based on packet header information, such as source and destination IP addresses, port numbers, and protocols. Based on these rules, the filter decides whether to allow or deny the packet. Stateless filters are relatively simple and fast, as they do not maintain any information about ongoing connections or packet history. However, their simplicity can also be a disadvantage, as they are unable to recognize the context of a network connection and may be less effective in detecting complex attacks or handling certain protocols.

Stateful packet filters, on the other hand, maintain a state table that tracks the status of ongoing network connections[2]. By keeping track of connection states, stateful filters can make more informed decisions about whether to allow or deny a packet. When a new packet arrives, the stateful filter examines both the packet header and the current state of the connection in its state table. If the packet is part of an existing, legitimate connection, it is allowed through; otherwise, it may be denied based on the filter's rules. Stateful packet filters provide a higher level of security compared to stateless filters, as they can better handle connection-oriented protocols and detect malicious activities that span multiple packets. However, they can be more resource-intensive and slower due to the additional overhead of maintaining and updating the state table.

1.1 P4 registers

P4 targets implement registers to save arbitrary data. Multiple packets can access the data stored in the registers. Registers in P4 are organized into named arrays of cells. Registers can be read and written by both the control and the data plane. In P4, registers are global memory resources meaning that any match-action tables can reference them.

The syntax below shows how to declare a register array in P4. The register array R1 contains `M` values of `N` bits.

```
register<bit<N>>(M) R1;
```

Figure 1 depicts a graphical representation of the register `R1`. The functions `write` and `read` are used to store and retrieve values from a specific position, where an index specifies the position[3]. For example, the programmer invokes the following function to store the value `val` in position 0 in the register array R1.

```
R1.write(0,val)
```

Similarly, the user invokes the function shown below to read a value stored in position 3. Note that the retrieved value is stored in the variable `res`.

```
R1.read(res,3)
```

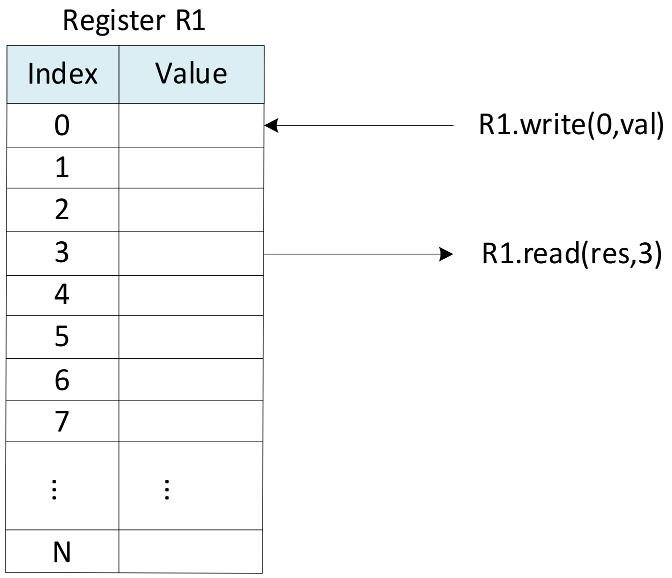


Figure 1. Register array R1. The register array contains N entries of M bits. The index indicates the position of the value. Using the functions `read` and `write`, programmers can retrieve and modify values in the register array.

1.2 Hashes in P4

P4 targets implement hash functions to map arbitrary data to a hash value. For example, the V1Model implements hash functions as externs[4]. The following code shows how to call a hash function in P4.

```
hash(hash_val, algo, min_val, {val_1, val_2, ..., val_N}, (n_bits, max_val))
```

The parameters of the hash function are as follows:

- `hash_val`: variable used to store the hash value.
- `algo`: indicates the hashing algorithm. For example, the V1Model supports crc16, crc32, universal hashing (i.e., random), xor32, and others.
- `min_val`: establishes the minimum hash value.
- `{val_1,val_2,…,val_N}`: values to be hashed.
- `n_bit`: number of bits of the output (i.e., width).
- `max_val`: maximum hash value.

1.3 Lab scenario

This lab shows how to implement a stateful packet filter for the TCP protocol using registers. Hashes are used to identify a flow, and registers are used to store the flow's state. The stateful packet filter will only allow hosts to initiate a TCP session to the DMZ server, thus, TCP flows that are not destined to the DMZ server are dropped.

The P4 program presented in this lab performs the following:

1- If the TCP header matches the assigned policy, the state of the flow (i.e., the TCP source and destination ports) is stored in a register.
2- If the TCP header does not match the assigned policy, the source and destination ports of the flow are extracted from the registers. If the extracted source and destination ports match the destination and source ports of the packet, respectively, the packet is accepted.

2 Lab topology

Let's get started by opening a simple Mininet topology using MiniEdit. The topology comprises three end hosts and one P4 programmable switch. Host h1 is in the internal network, host h2 is in the DMZ network, and host h3 is in the external network.

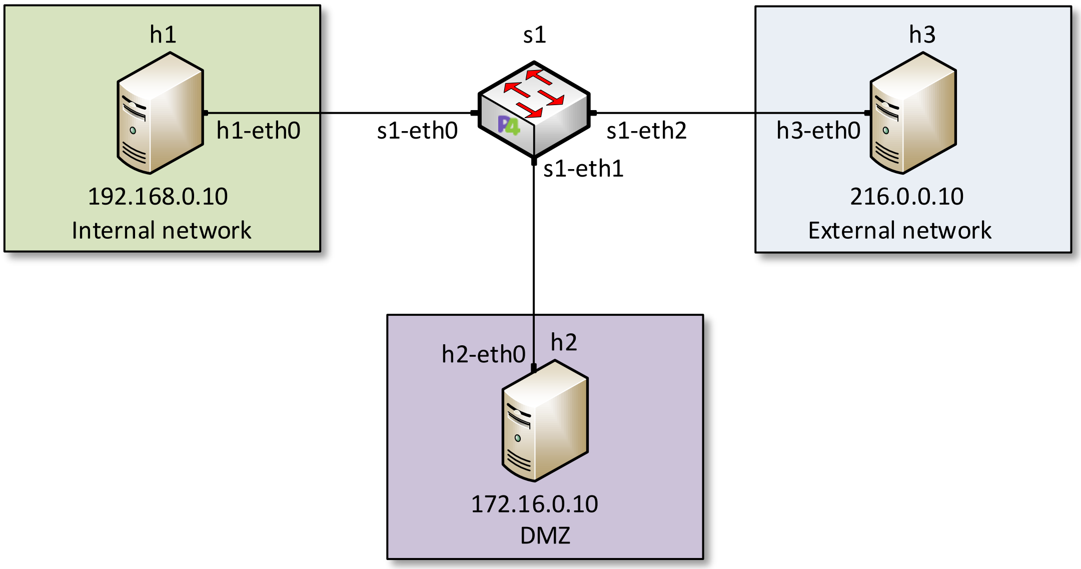


Figure 2. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

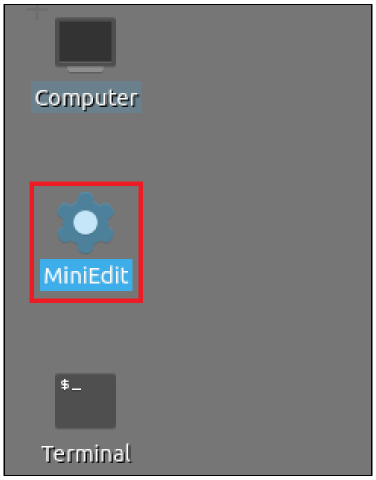
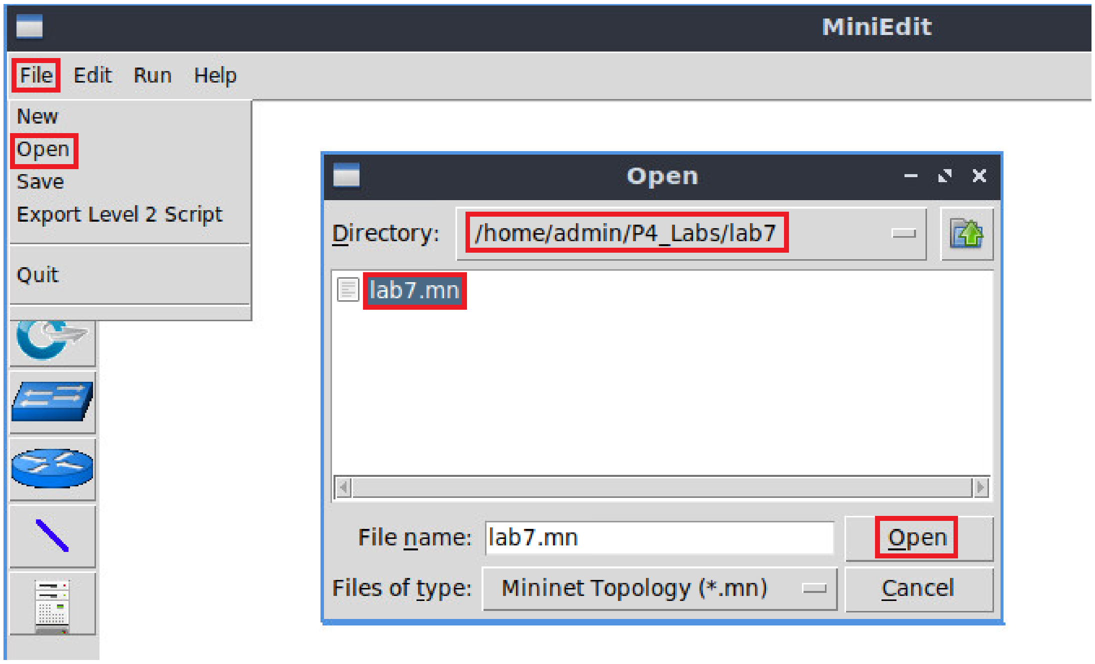


Figure 3. MiniEdit shortcut.

Step 2. In the MiniEdit application, load the topology by clicking on *File* then *Open*. Navigate to the *lab7* folder and search for the topology file called *lab7.mn* and click on *Open*. A new topology will be loaded to MiniEdit.



Directory: /home/admin/P4_Labs/lab7

lab7.mn

File name: lab7.mn

Files of type: Mininet Topology (*.mn)

Figure 4. MiniEdit's *Open* dialog.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

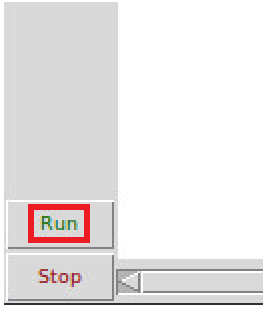


Figure 5. Running the emulation.

2.1 Verifying the configuration of the end hosts

Step 1. Right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

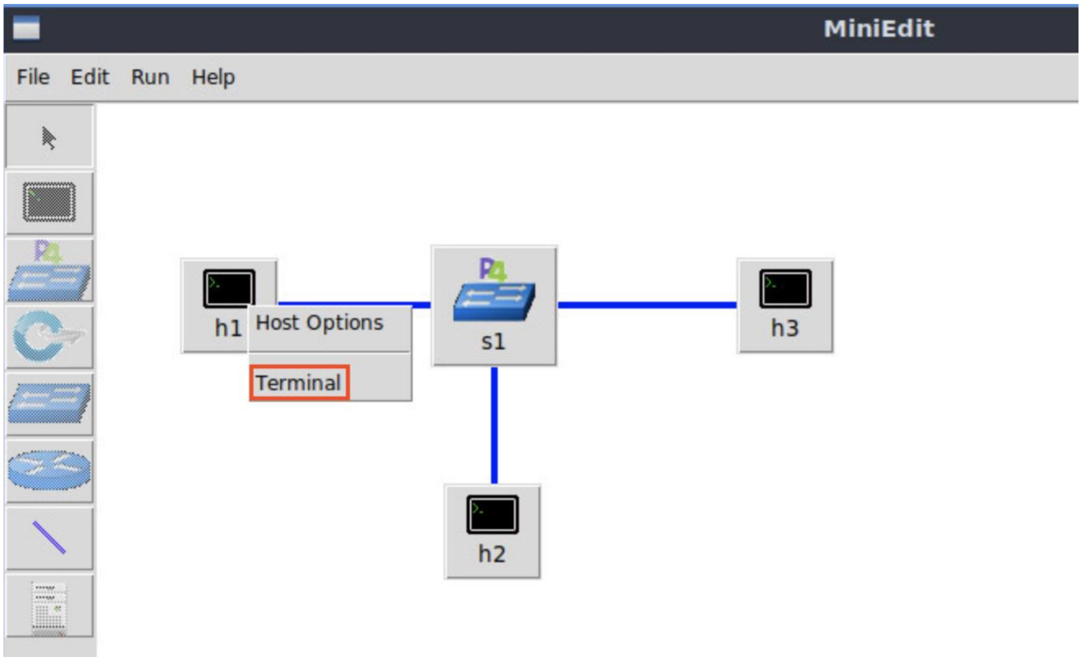
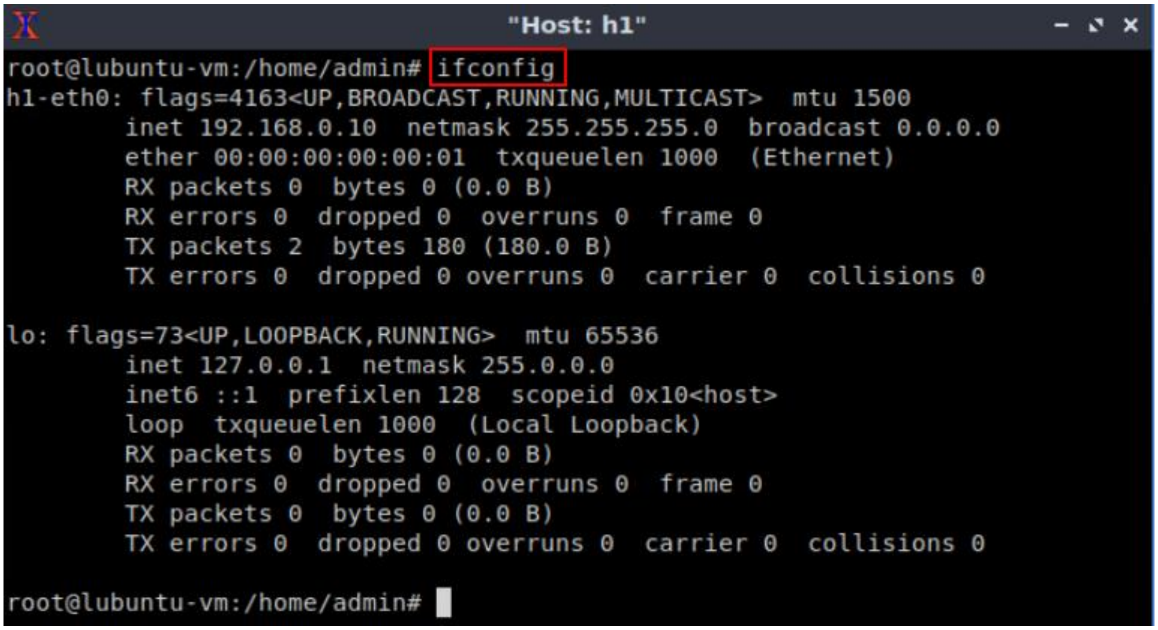


Figure 6. Opening a terminal on host h1.

Step 2. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 180 (180.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin#
```

Figure 7. Verifying the configuration host h1 interfaces.

Step 3. Hold the right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

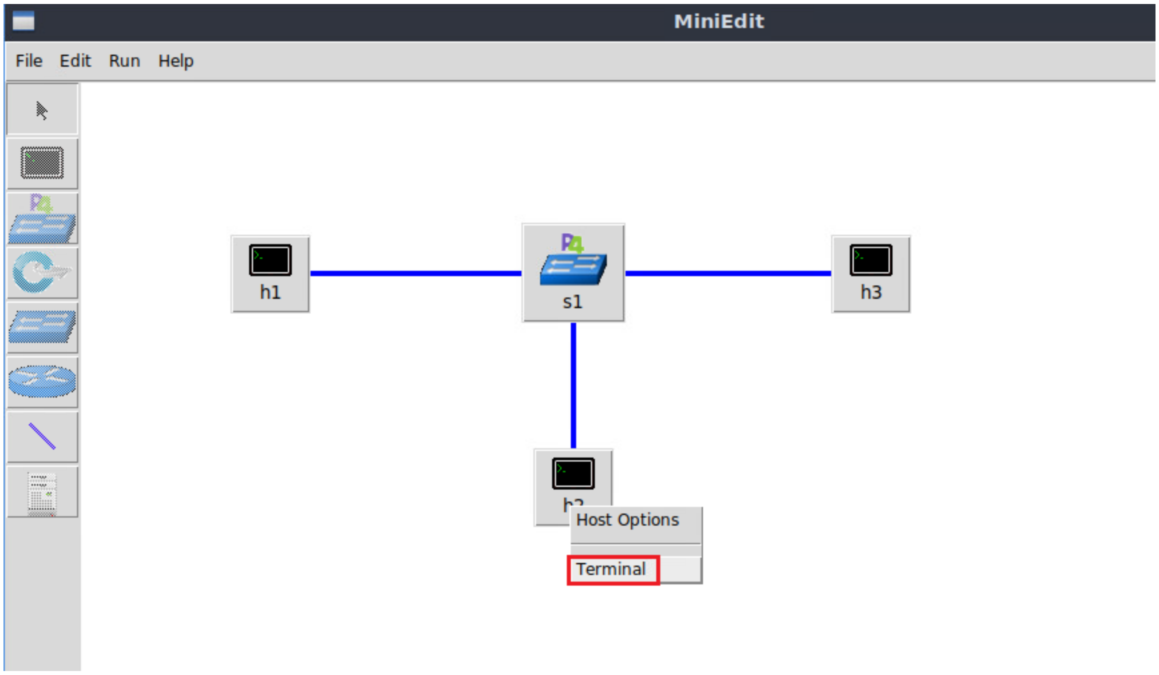
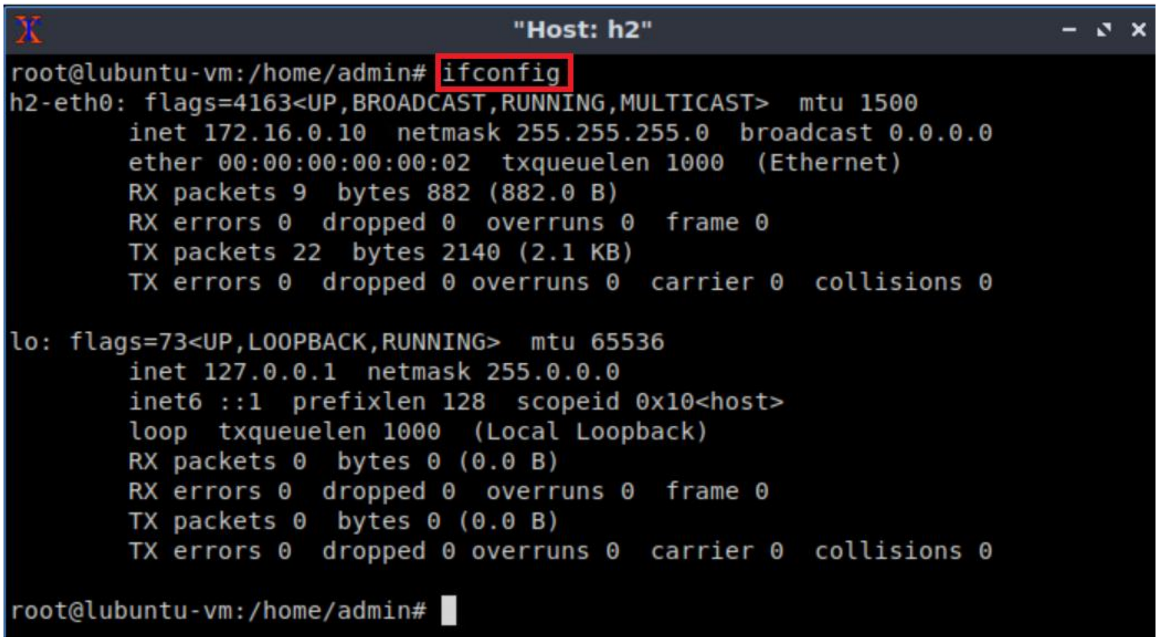


Figure 8. Opening a terminal on host h2.

Step 4. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.16.0.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:02  txqueuelen 1000  (Ethernet)
        RX packets 9  bytes 882 (882.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2140 (2.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin# 
```

Figure 9. Verifying the configuration host h2 interfaces.

Step 5. Hold the right-click on host h3 and select *Terminal*. This opens the terminal of host h3 and allows the execution of commands on that host.

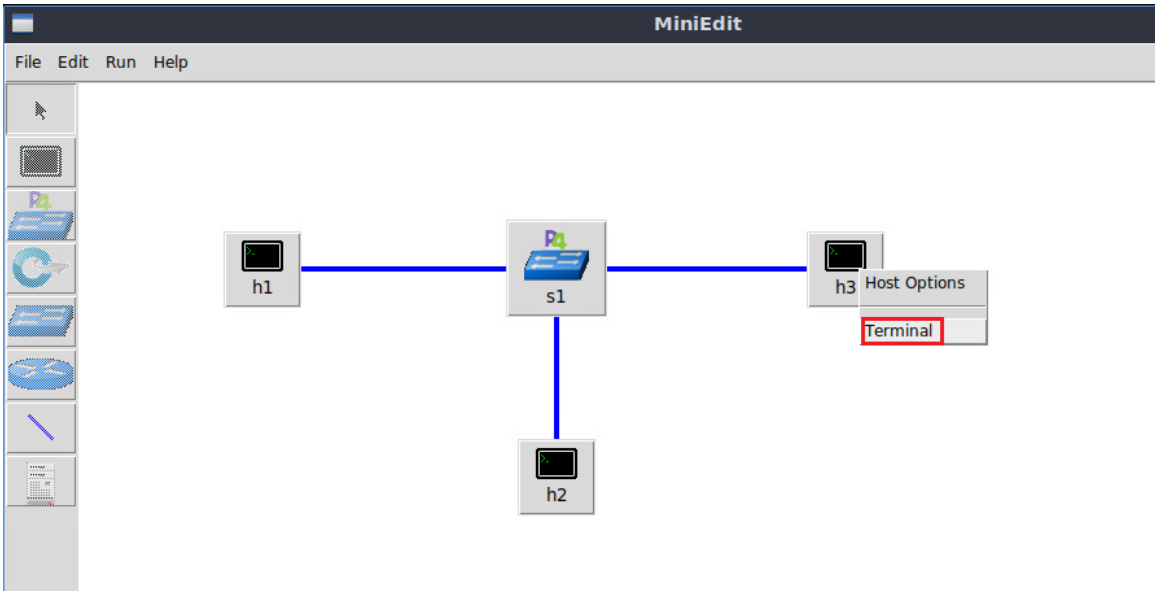
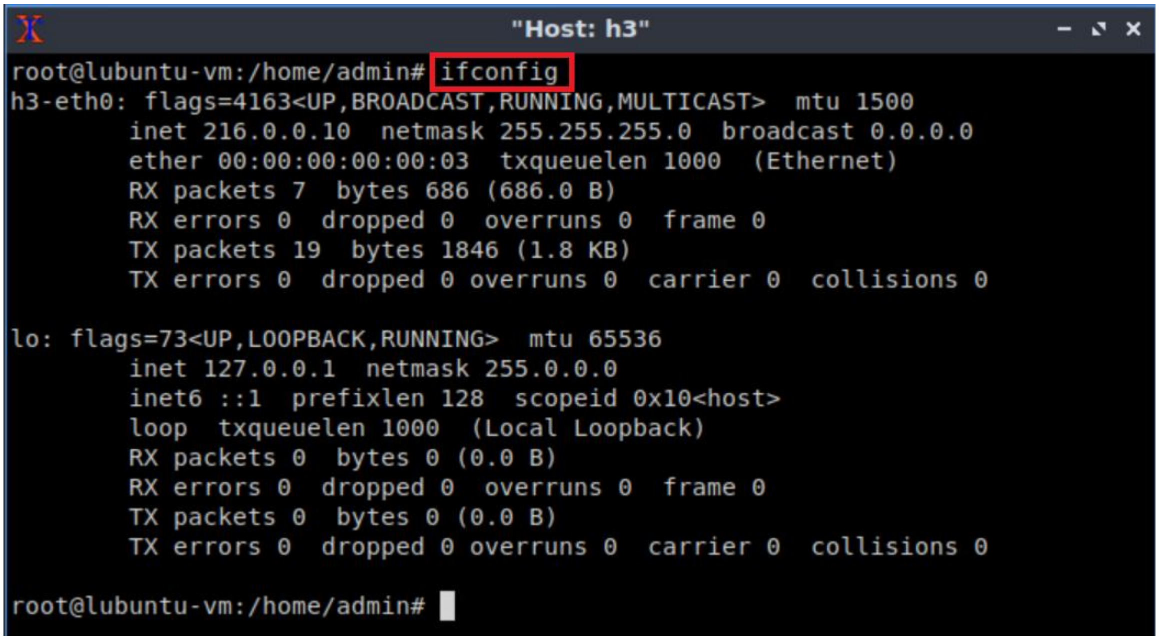


Figure 10. Opening a terminal on host h3.

Step 6. Verify the interfaces' configuration by issuing the following command.

```
ifconfig
```



```
root@lubuntu-vm:/home/admin# ifconfig
h3-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 216.0.0.10  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:03  txqueuelen 1000  (Ethernet)
        RX packets 7  bytes 686 (686.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19  bytes 1846 (1.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@lubuntu-vm:/home/admin#
```

Figure 11. Verifying the configuration host h3 interfaces.

3 Creating a P4 program that performs stateful packet filtering

This section demonstrates how to implement a stateful packet filter in P4 using registers. The stateful packet filter will be applied on TCP. First, you will load the programming environment. Then, you will define the headers to parse TCP. Following, you will create P4 tables to apply the desired policies, as well as registers to store the state of the flow. The flow ID is produced by a hashing algorithm that computes the source and destination IPv4 addresses to produce an index. This index will be used to access the state of the flow and decide whether to forward or block packet based on the policy.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the icon located on the desktop. Alternatively, click on the terminal icon in taskbar located in the lower left-hand side.

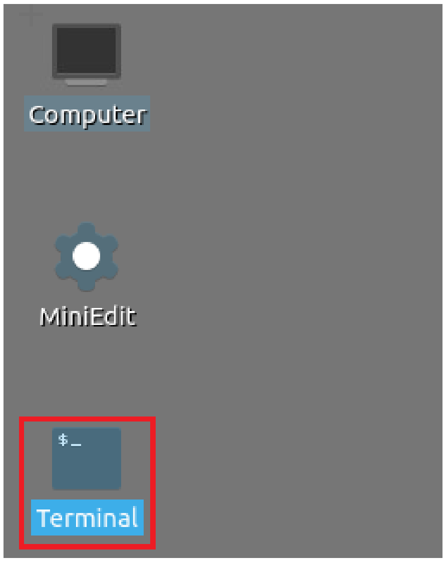
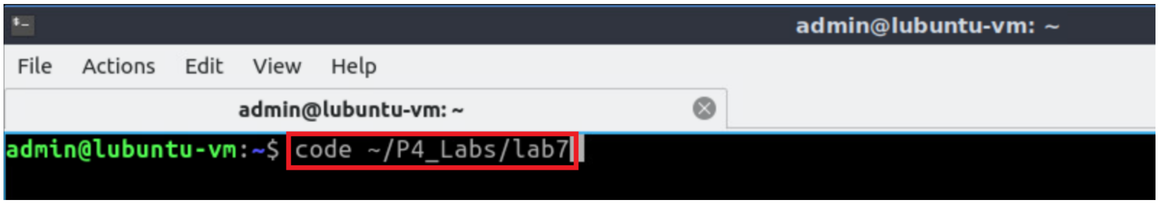


Figure 12. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the Visual Studio Code (VS Code) and opens the directory where the P4 program for this lab is located.

```
code ~/P4_Labs/lab7
```

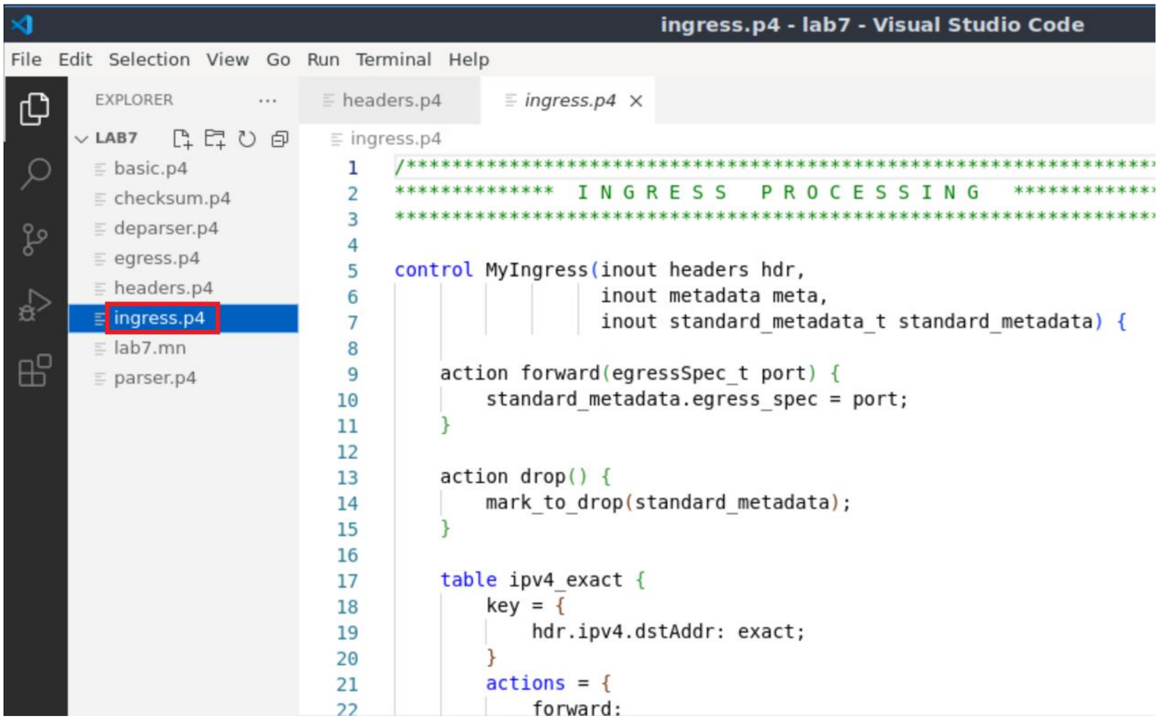


```
admin@lubuntu-vm:~$ code ~/P4_Labs/lab7
```

Figure 13. Launching the editor and opening the lab7 directory.

3.2 Implementing a TCP stateful packet filter

Step 1. Click on the *ingress.p4* file to display the contents of the file. Use the file explorer on the left-hand side of the screen to locate the file.



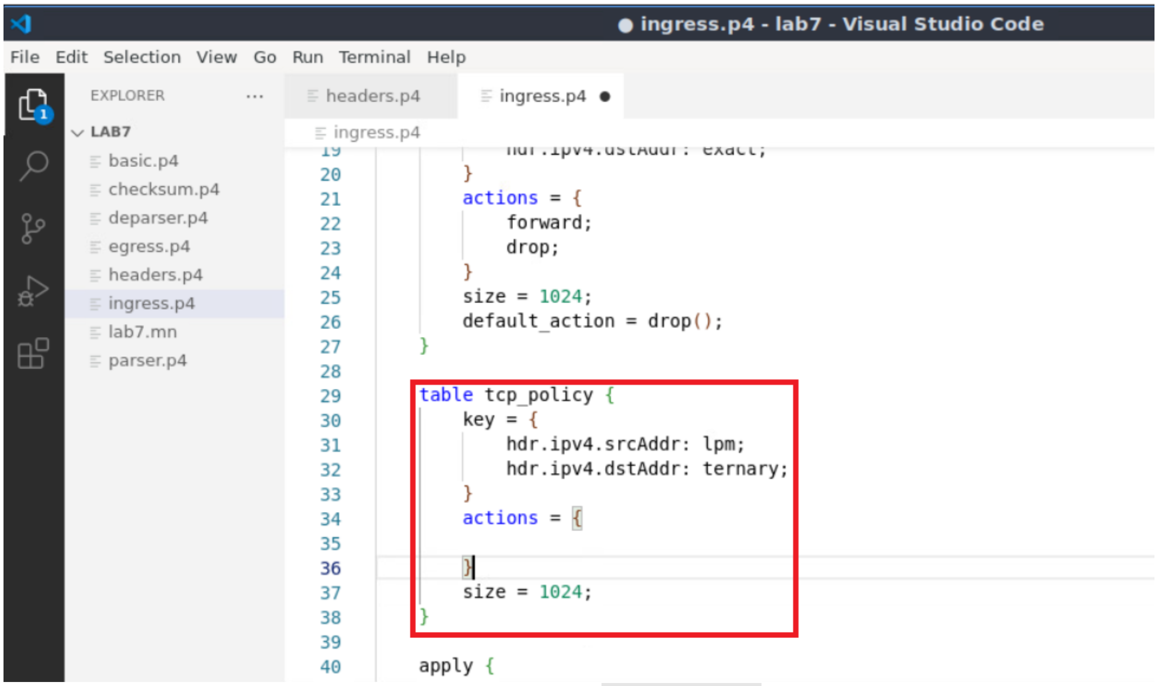
```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *************
*************************************************************************

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action forward(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table ipv4_exact {
        key = {
            hdr.ipv4.dstAddr: exact;
        }
        actions = {
            forward;
```

Figure 14. Inspecting the *ingress.p4* file.

We can see that the *ingress.p4* declares a control block named `MyIngress`. Inside this block, a table `ipv4_exact` is defined which matches on the destination IP address and can invoke the `forward` action to forward the packet out of a port, or the `drop` action to drop the packet.

Step 2. Add the code below under the *ipv4_exact* table to define `tcp_policy` table. The table is responsible for checking incoming TCP packets against the access rules defined by the policy.

```
table tcp_policy {
    key = {
        hdr.ipv4.srcAddr: lpm;
        hdr.ipv4.dstAddr: ternary;
    }
    actions = {

    }
    size = 1024;
}
```



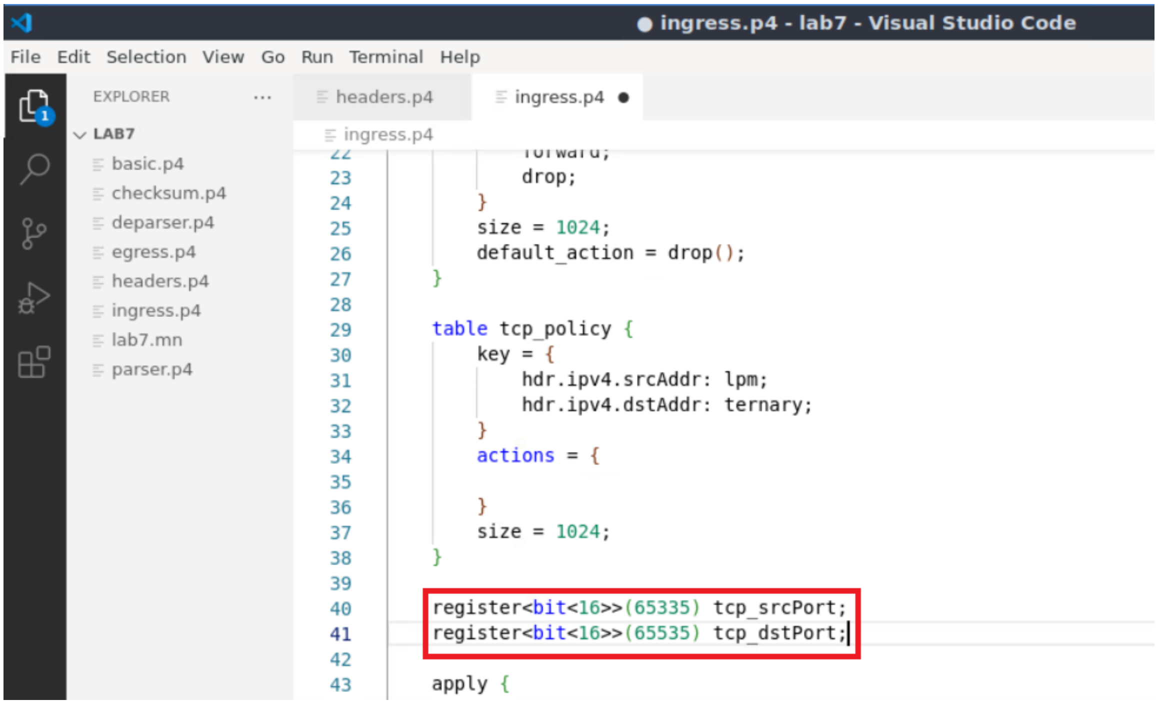
```
            hdr.ipv4.dstAddr: exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    table tcp_policy {
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    apply {
```

Figure 15. Defining the `tcp_policy` table.

In the code above, the table `tcp_policy` uses the IP source and destination addresses as the table keys, i.e., the table checks if the incoming packets belong to a defined rule by inspecting the source and destination IP addresses. The keys are populated and configured by the control plane at runtime based on the policy in place. `lpm` and `ternary` matching types are used to allow the administrator to define a policy on a range of IP addresses. Note that BMv2 compiler (p4c) does not allow a table that has more than one LPM key field, thus, the ternary matching is used[5].

Step 3. Define two registers `tcp_srcPort` and `tcp_dstPort` by typing the code below. The registers are responsible for storing the source and destination ports of the allowed TCP sessions, enabling stateful packet filter.

```
register<bit<16>>(65535) tcp_srcPort;
register<bit<16>>(65535) tcp_dstPort;
```

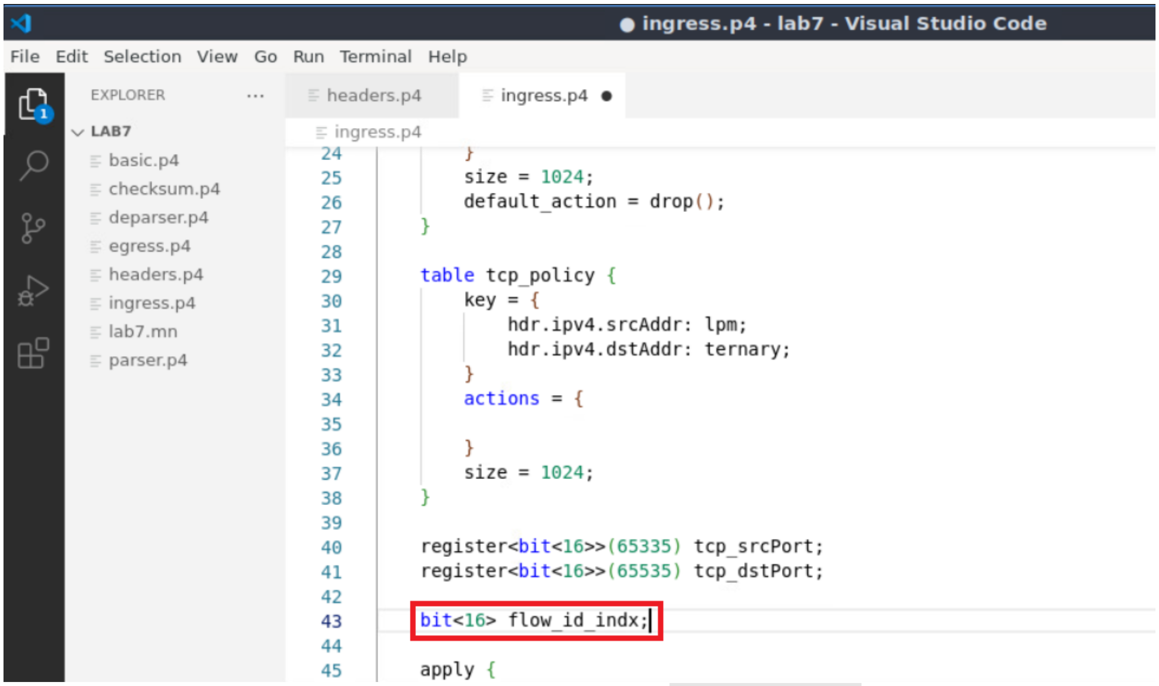


```
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    table tcp_policy {
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    apply {
```

Figure 16. Defining the registers store TCP ports.

Step 4. Insert the code below to define a 16-bit variable to store the hash index of the flow.

```
bit<16> flow_id_indx;
```

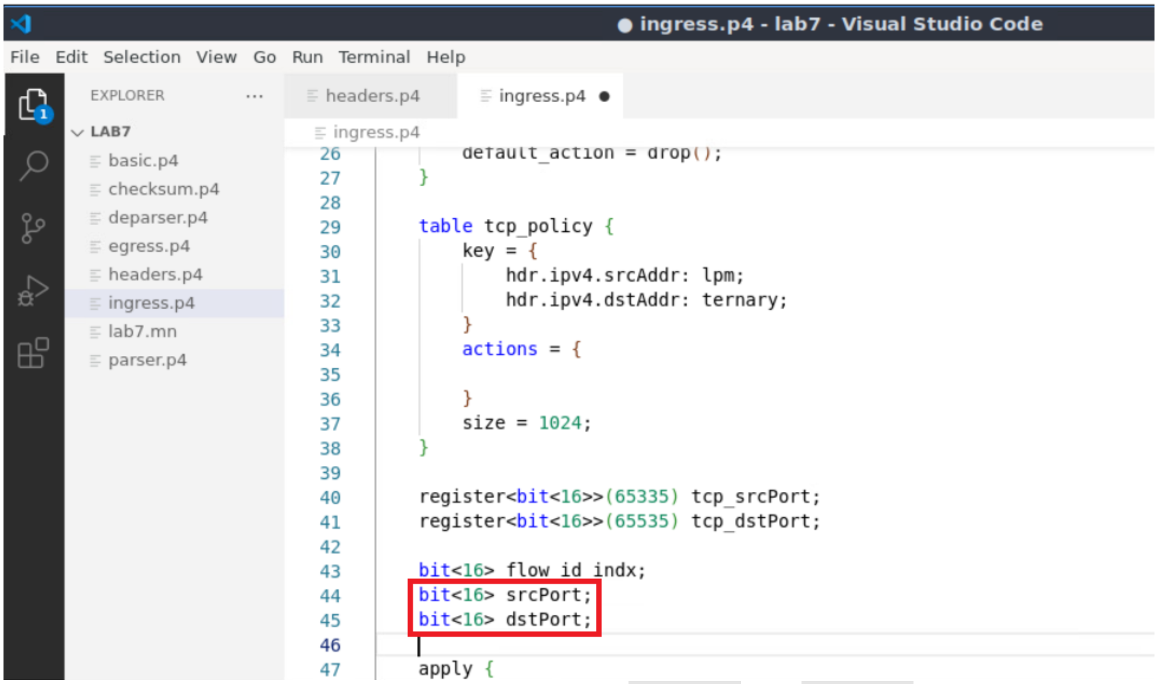


```
        }
        size = 1024;
        default_action = drop();
    }

    table tcp_policy {
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;

    apply {
```

Figure 17. Defining the variable `flow_id_indx`.

Step 5. Insert the code below to define two 16-bit variables to store the TCP source (`srcPort`) and destination (`dstPort`) ports.

```
bit<16> srcPort;
bit<16> dstPort;
```

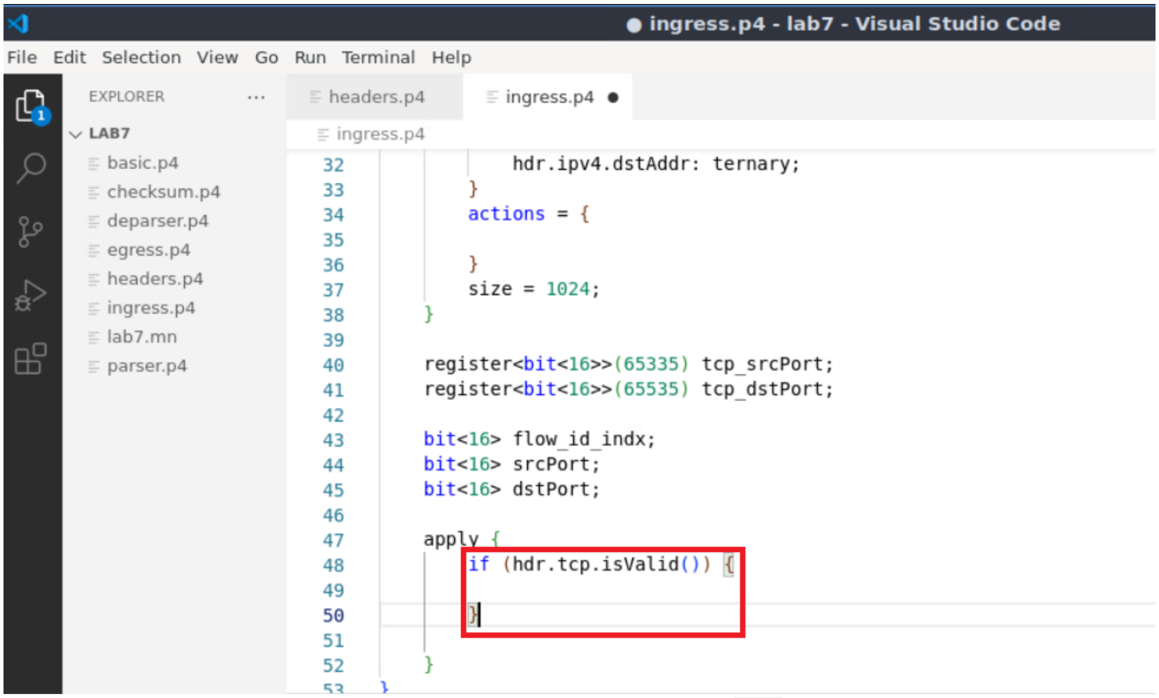


```
        default_action = drop();
    }

    table tcp_policy {
        key = {
            hdr.ipv4.srcAddr: lpm;
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
```

Figure 18. Defining the variables `srcPort` and `dstPort`.

Step 6. Add the following code inside the *apply* block of the *MyIngress* control to check if the packet is an TCP packet. The *apply* block defines the sequential flow of packet processing. It is required in every control block, otherwise the program will not compile. It describes the sequence of tables to be invoked, in addition to other packet processing instructions.

```
if (hdr.tcp.isValid()) {

}
```

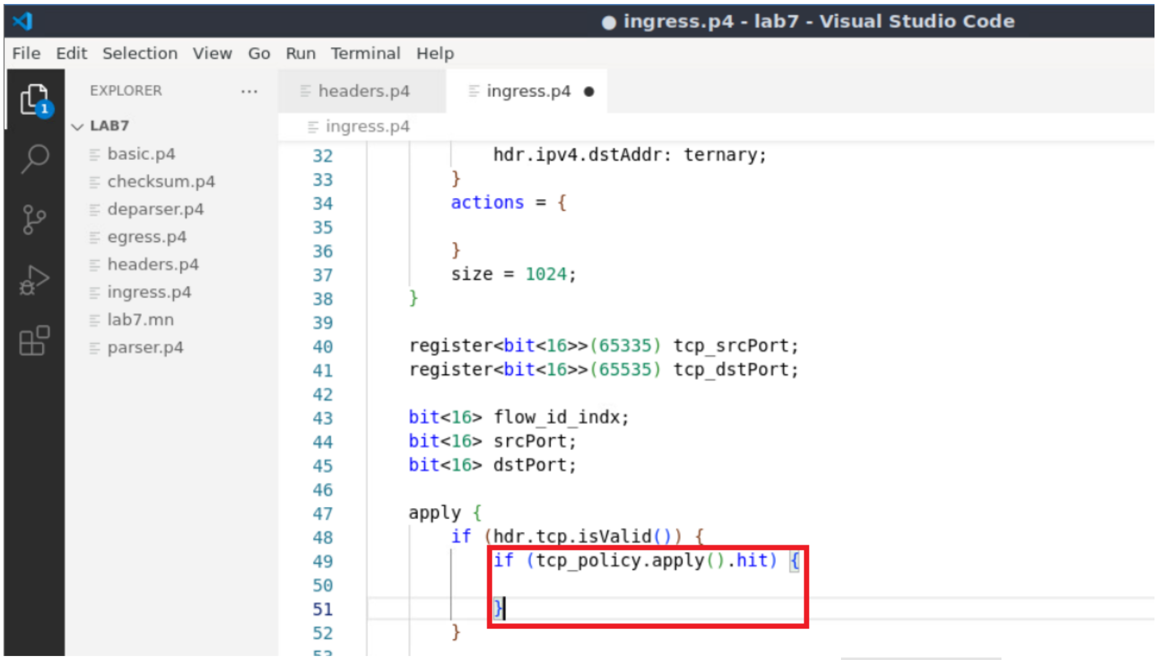


```
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {

        }
    }
}
```

Figure 19. Checking the validity of the `tcp` header.

Step 7. Add the following code to check if the TCP packet matches one of the access rules defined by the `tcp_policy` table.

```
if (tcp_policy.apply().hit) {

}
```



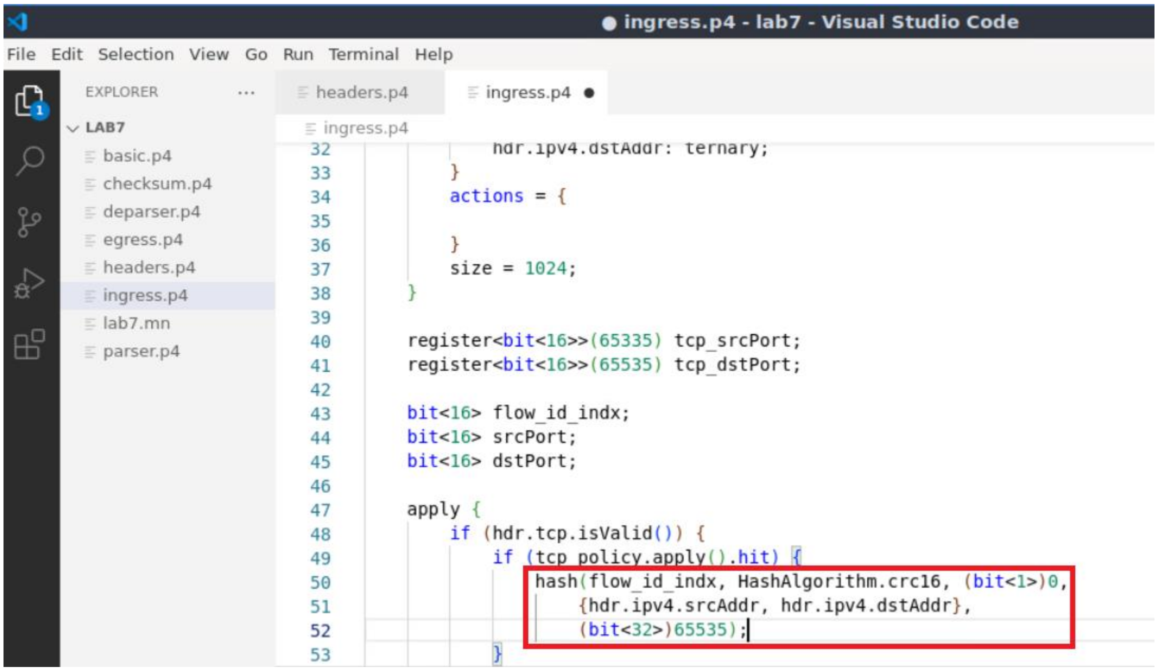
```
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {

            }
        }
```

Figure 20. Checking if the packet matches the policy defined by `tcp_policy` table.

Note that a table hit occurs only if a packet matches against the table's keys. In the code above, the if statement applies to the `tcp_policy` table and checks if the source and destination IP addresses of the packet match the table's keys, i.e., the packet matches the policy. If the packet matches the policy, then its source and destinations ports should be stored inside stateful registers. The registers should be indexed using the hash of the source and destinations ports.

Step 8. Add the following code to calculate the hash of the source and destination ports and store it inside `flow_id_indx` variable. The hash will be used to index `tcp_srcPort` and `tcp_dstPort` registers.

```
hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
     {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
     (bit<32>)65535);
```



```
            hdr.ipv4.dstAddr: ternary;
        }
        actions = {

        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
            }
```

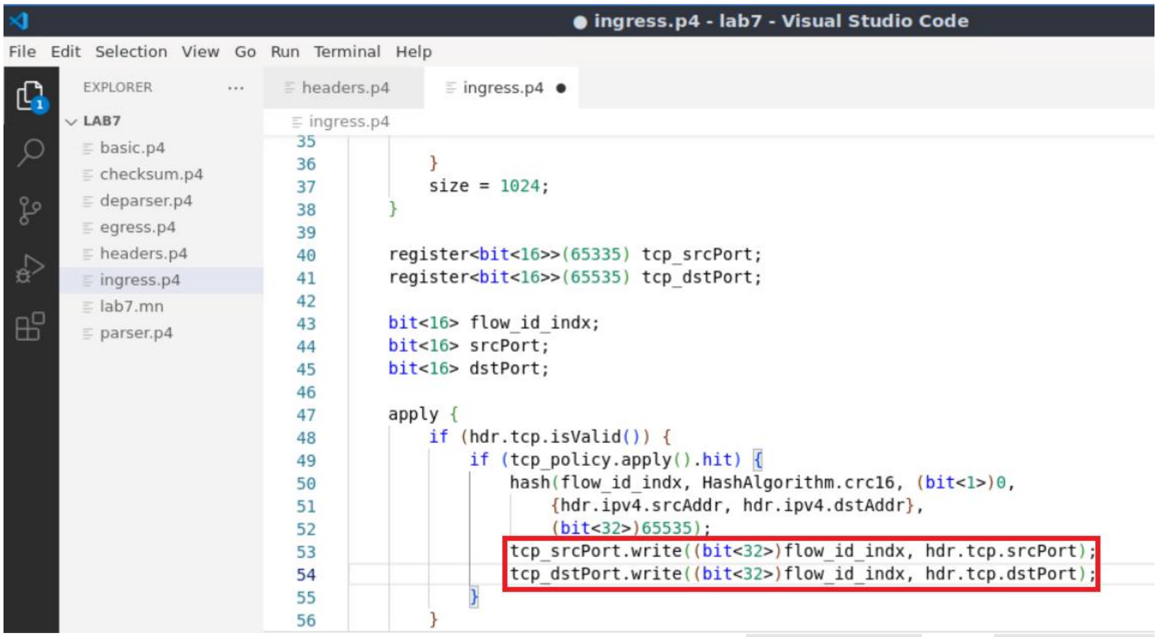
Figure 21. Calculating the index of the registers where the source and destination ports will be stored.

The code in the figure above hashes flows based on their source and destination IP addresses. The hash function produces a 16-bits output using the following parameters:

- `flow_id_indx`: The variable used to store the output.
- `HashAlgorithm.crc16`: the hash algorithm.
- `(bit<1>)0`: the minimum (or base) value produced by the hash algorithm.
- `hdr.ipv4.srcAddr` and `hdr.ipv4.dstAddr`: the data to be hashed.
- `(bit<32>)65535`: the maximum value produced by the hash algorithm.

Step 9. Add the following code to store the source and destination ports inside the `tcp_srcPort` and `tcp_dstPort` registers, respectively.

```
tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
```

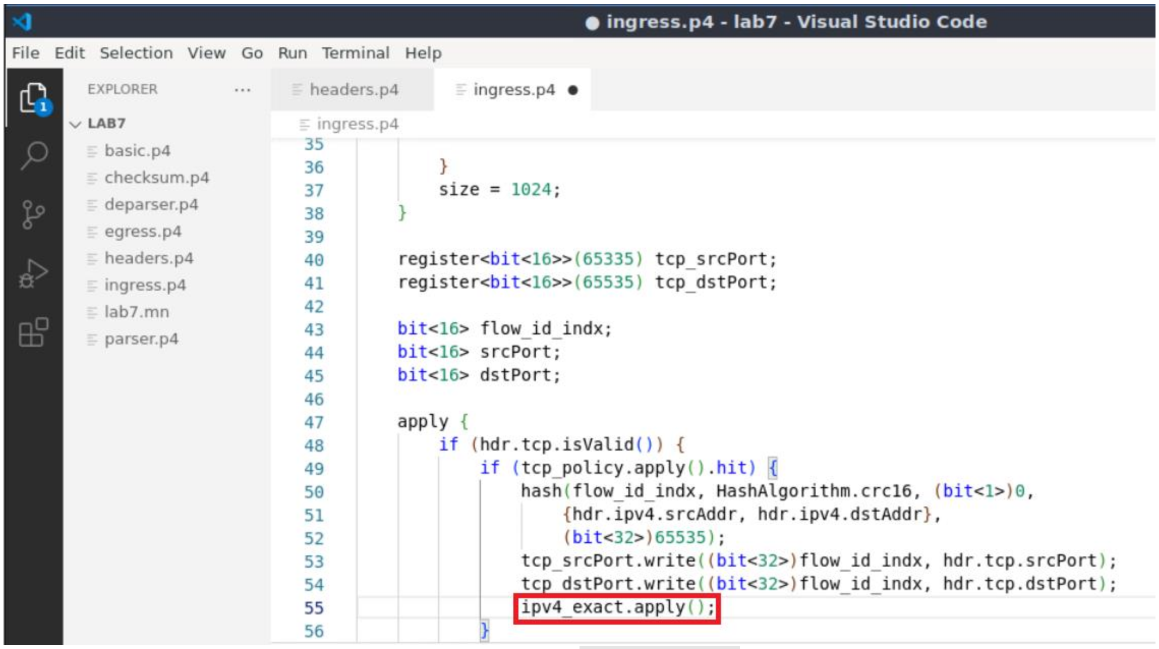


```
        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            }
        }
```

Figure 22. Storing source and destination ports inside the `tcp_srcPort` and `tcpdstPort` registers.

Step 10. Insert the code below to apply the `ipv4_exact` table and forward the packet to the destination host.

```
ipv4_exact.apply();
```



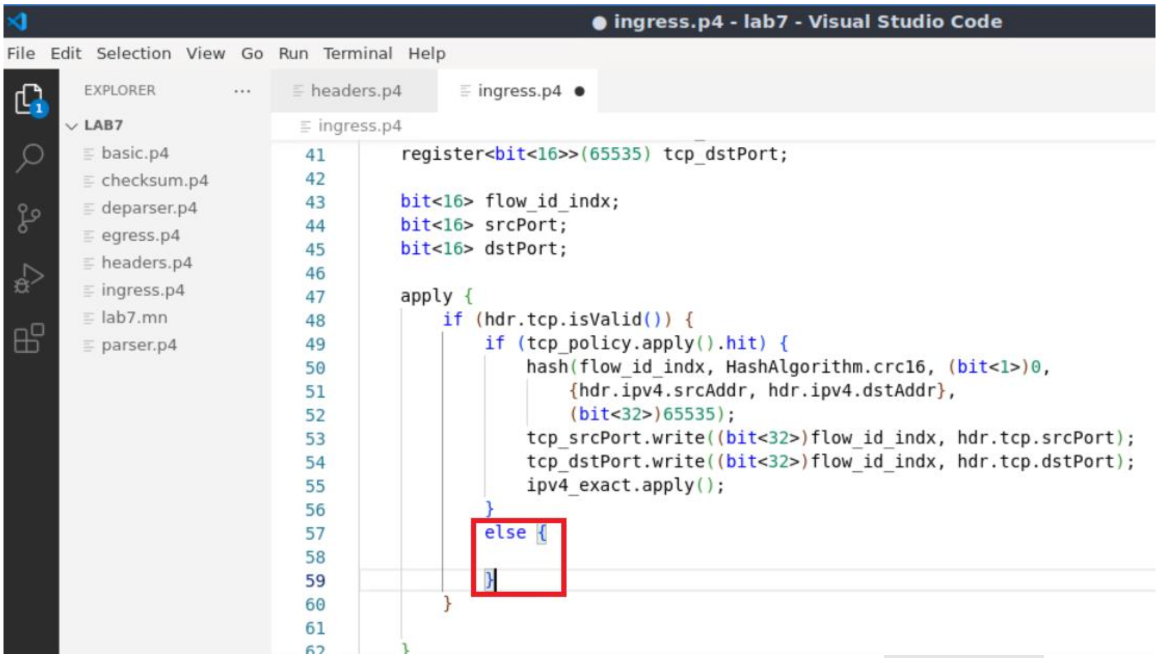
```
        }
        size = 1024;
    }

    register<bit<16>>(65335) tcp_srcPort;
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
                ipv4_exact.apply();
            }
```

Figure 23. Applying `ipv4_exact` table.

At this stage, the source and destination ports of packets that match the policy are stored inside two stateful registers thar are indexed by the hash of the source and destination ports. The next step is to write a code to process the packets that do not match the policy.

Step 11. Add the following code to check if the TCP packet does not match one of the access rules defined by the `tcp_policy` table.

```
else {

}
```

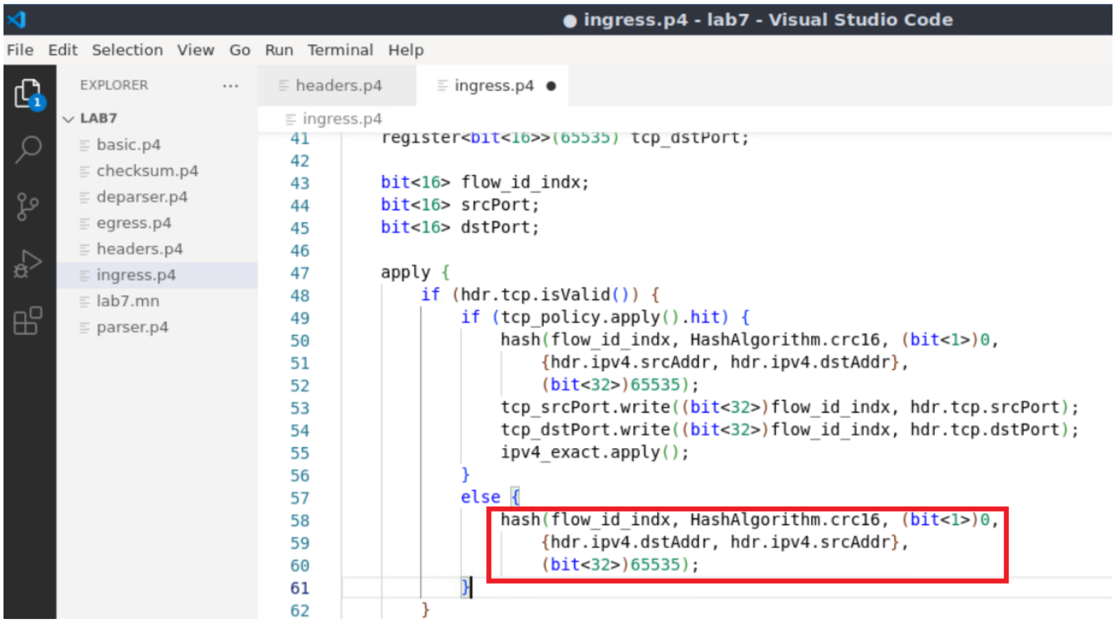


```
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
                ipv4_exact.apply();
            }
            else {

            }
        }

    }
```

Figure 24. Checking if the packet does not match the policy defined by `tcp_policy` table.

Step 12. Add the following code to calculate the hash of the source and destination ports and store it inside `flow_id_indx` variable. The hash will be used to index `tcp_srcPort` and `tcp_dstPort` registers.

```
hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
     {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
     (bit<32>)65535);
```



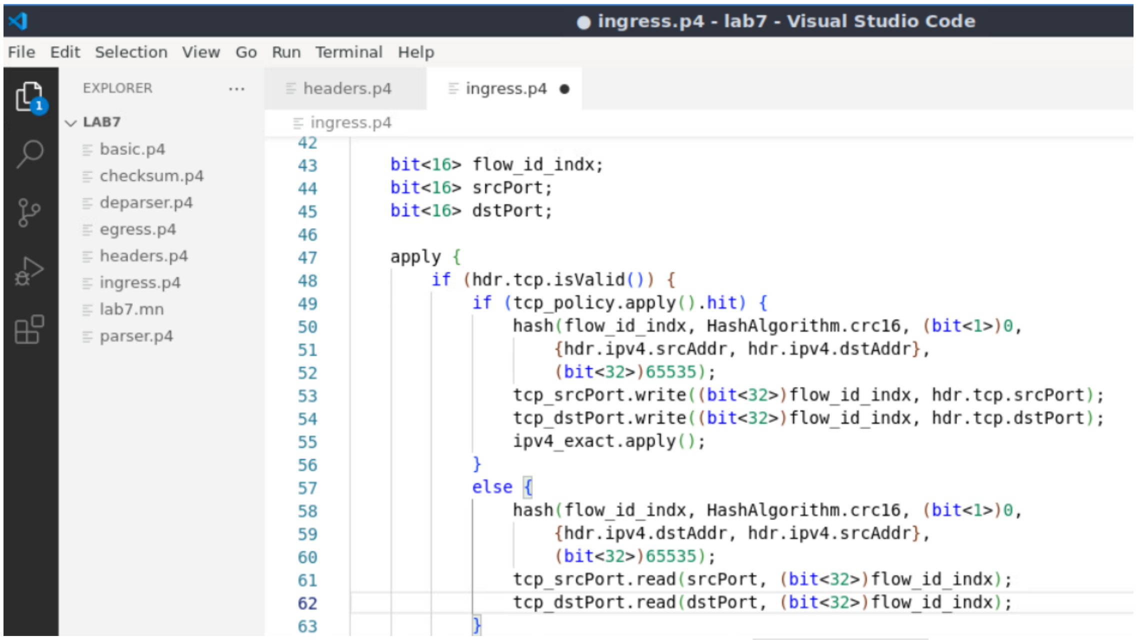
```
    register<bit<16>>(65535) tcp_dstPort;

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
                ipv4_exact.apply();
            }
            else {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
            }
        }
```

Figure 25. Calculating the index of the registers where the source and destination ports will be retrieved from.

The code in the figure above hashes flows based on their destination and source IP addresses. Notice how the source and destination IP addresses are inverted in the hash function so that the retrieved index is the same as the one used in the associated TCP session.

Step 13. Add the following code to retrieve the source and destination ports from the `tcp_srcPort` and `tcp_dstPort` registers.

```
tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
```

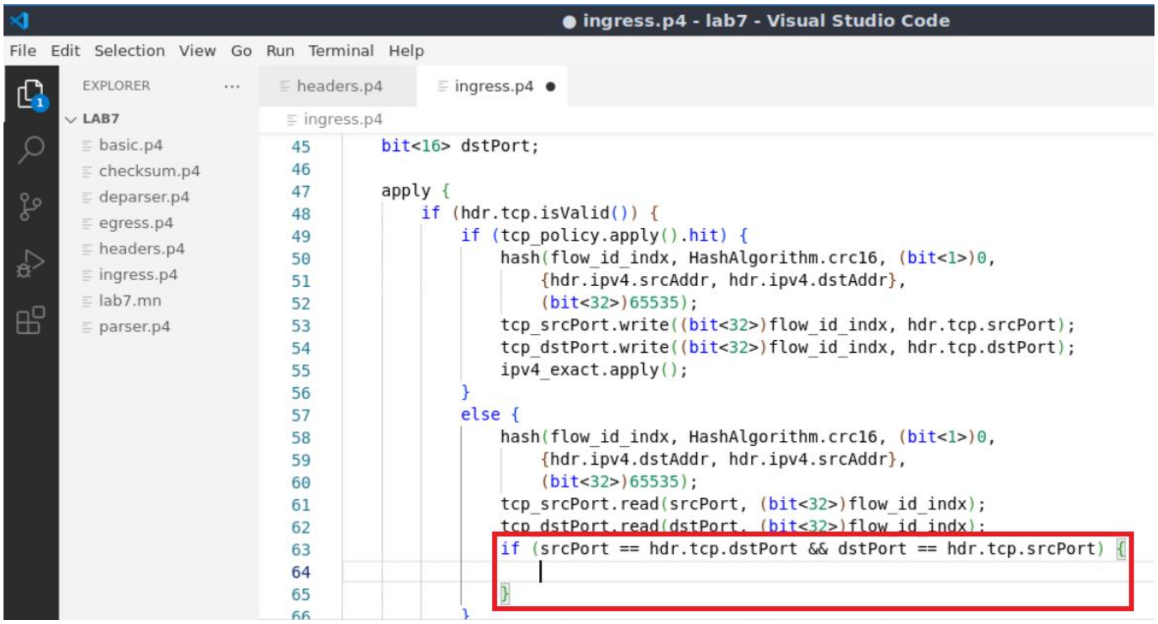


```

    bit<16> flow_id_indx;
    bit<16> srcPort;
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
                ipv4_exact.apply();
            }
            else {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
                tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            }
```

Figure 26. Retrieving the source and destination ports from the `tcp_srcPort` and `tcpdstPort` registers.

Step 14. Add the following code to check if the retrieved source and destination ports match the packet's ports.

```
if (srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {

}
```



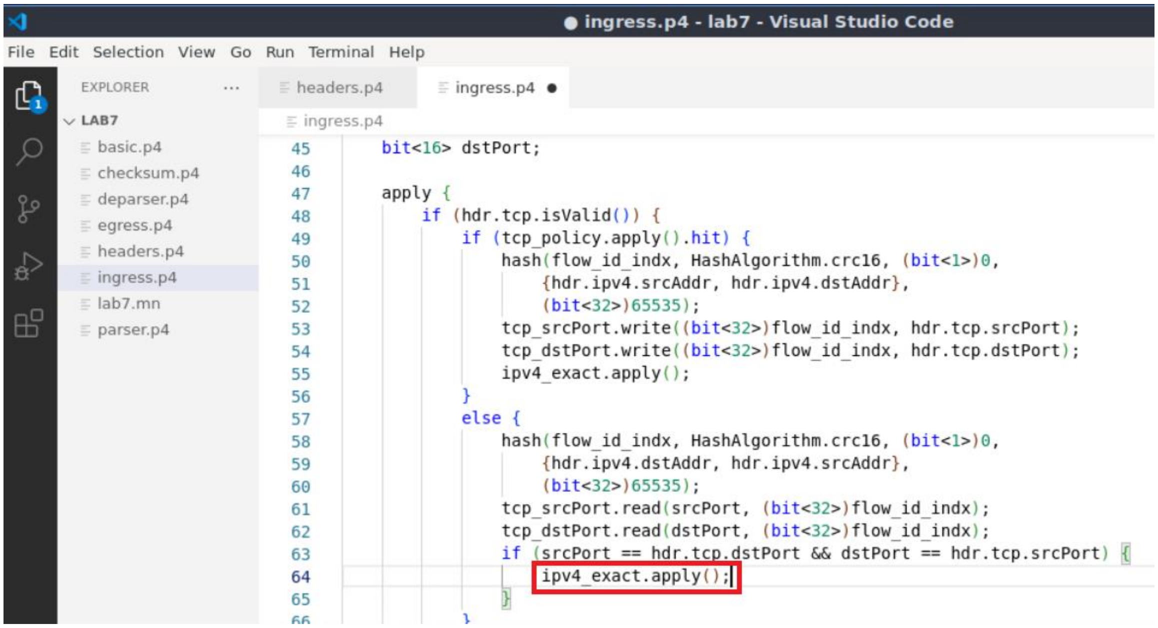
```
    bit<16> dstPort;

    apply {
        if (hdr.tcp.isValid()) {
            if (tcp_policy.apply().hit) {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
                ipv4_exact.apply();
            }
            else {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
                tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
                if (srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {

                }
            }
```

Figure 27. Checking if the packet belongs to an existing TCP session.

In the code above, the source and destination ports of the packets are compared against the retrieved ports from the registers. The two pairs should match only if the packet belongs to an existing TCP session, and consequently, the packet should be forwarded. Otherwise, the packet should be dropped.

Step 15. Add the following code to forward the packet if it belongs to an existing TCP session.

```
ipv4_exact.apply();
```

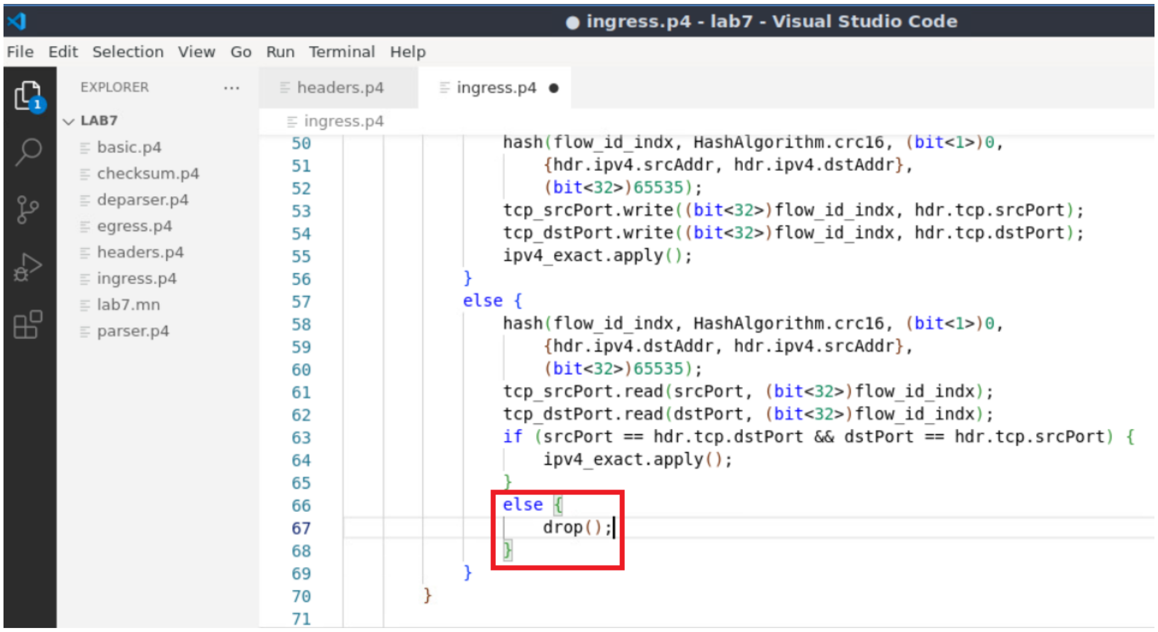


```
bit<16> dstPort;

apply {
    if (hdr.tcp.isValid()) {
        if (tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if (srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            }
        }
```

Figure 28. Forwarding TCP packets that belong to an existing session.

Step 16. Add the following code to drop the packet if it does not belong to an existing TCP session.

```
else {
    drop();
}
```



```
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if (srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            }
            else {
                drop();
            }
        }
    }

```

Figure 29. Dropping TCP packets that do not belong to an existing session.

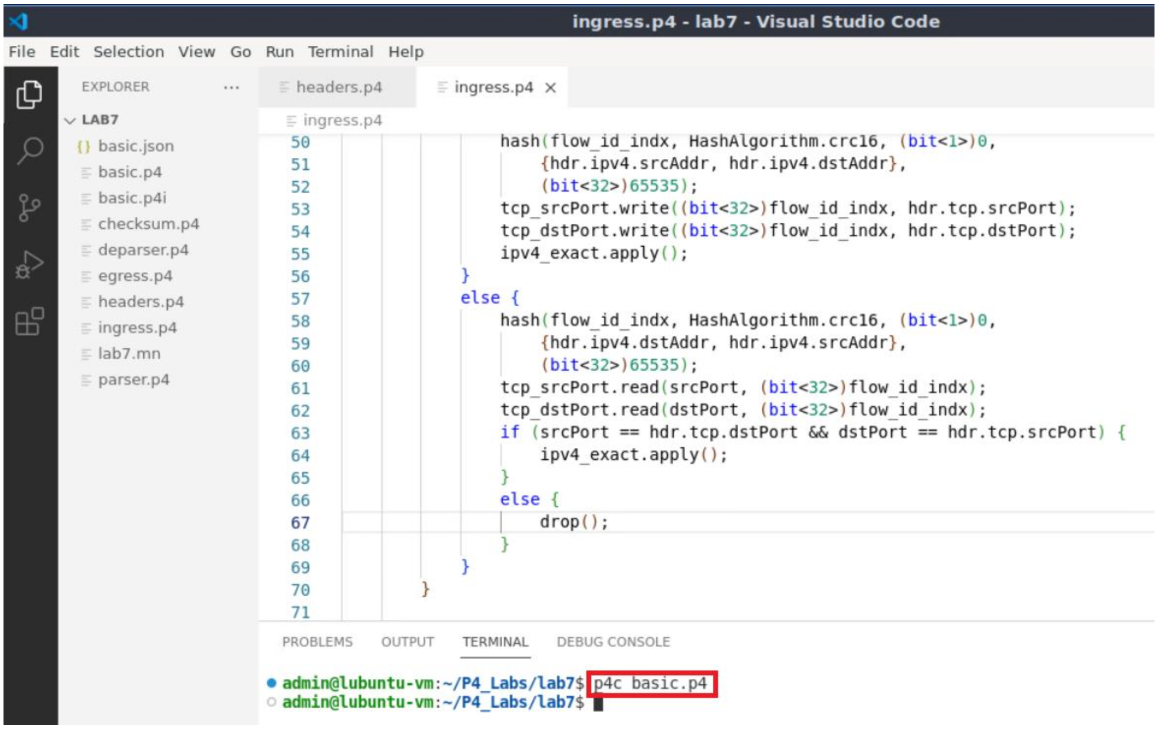
Step 17. Save the changes to the file by pressing `Ctrl + s`.

4 Loading the P4 program

4.1 Compiling and loading the P4 program to switch s1

Step 1. Issue the following command in the terminal panel inside the VS Code to compile the program.

```
p4c basic.p4
```



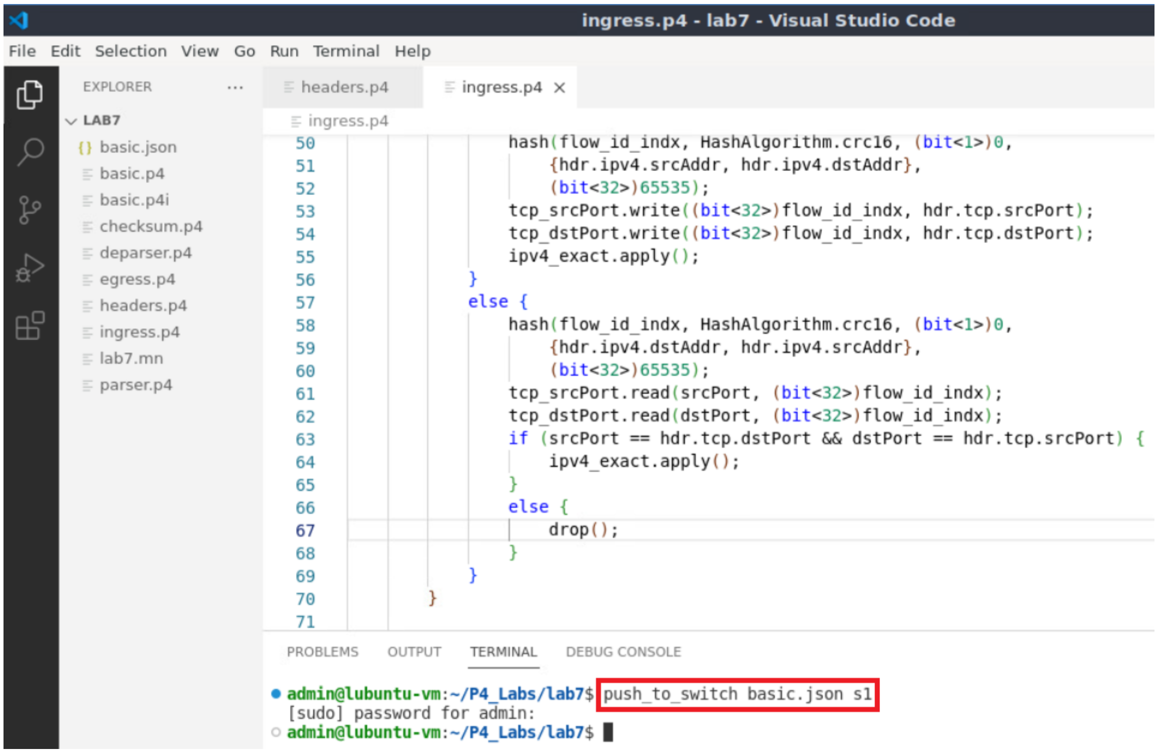
```
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if (srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            }
            else {
                drop();
            }
        }
    }
```

```
admin@lubuntu-vm:~/P4_Labs/lab7$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab7$
```

Figure 30. Compiling the P4 program.

Step 2. Type the command below in the terminal panel to push the *basic.json* file to the switch s1's filesystem. The script accepts as input the JSON output of the p4c compiler, and the target switch name. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



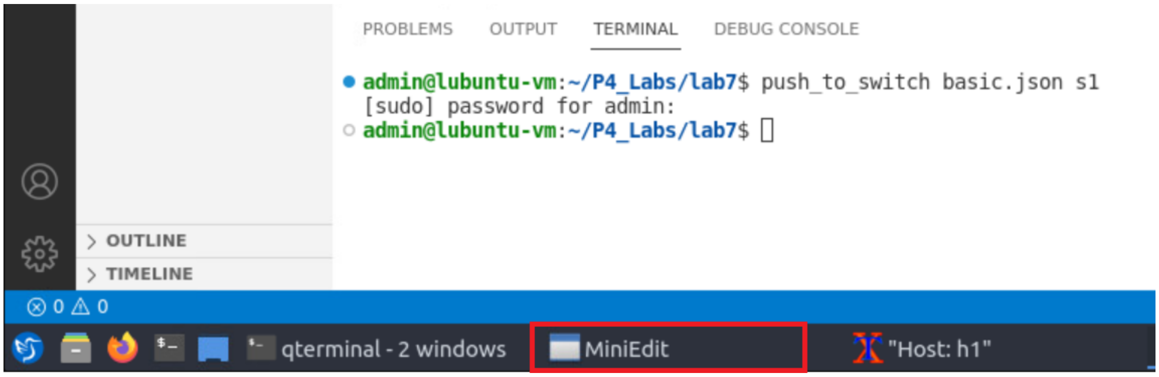
```
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr},
                    (bit<32>)65535);
                tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
                tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
                ipv4_exact.apply();
            }
            else {
                hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0,
                    {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr},
                    (bit<32>)65535);
                tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
                tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
                if (srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                    ipv4_exact.apply();
                }
                else {
                    drop();
                }
            }
        }
```

```
admin@lubuntu-vm:~/P4_Labs/lab7$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab7$
```

Figure 31. Pushing the *basic.json* file to switch s1.

4.2 Verifying the configuration

Step 1. Click on the MinEdit tab in the start bar to maximize the window.



```
admin@lubuntu-vm:~/P4_Labs/lab7$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab7$
```

Figure 32. Maximizing the MiniEdit window.

Step 2. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

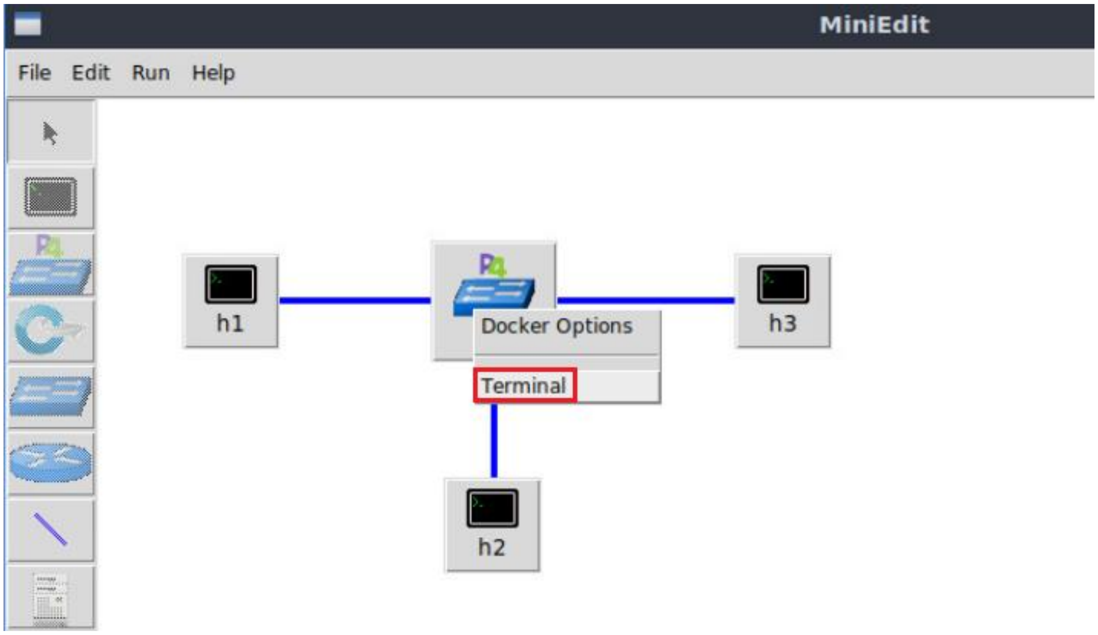
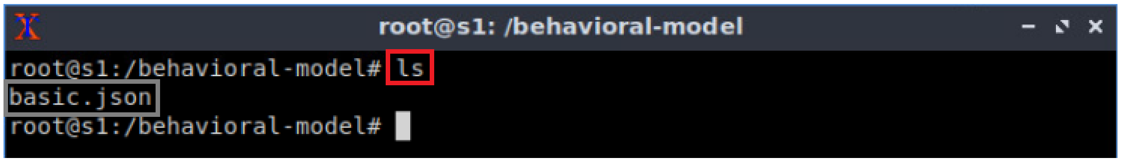


Figure 33. Opening switch s1 terminal.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch terminal.

Step 3. Issue the following command on switch s1 terminal to inspect the content of the current folder.

```
ls
```



```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

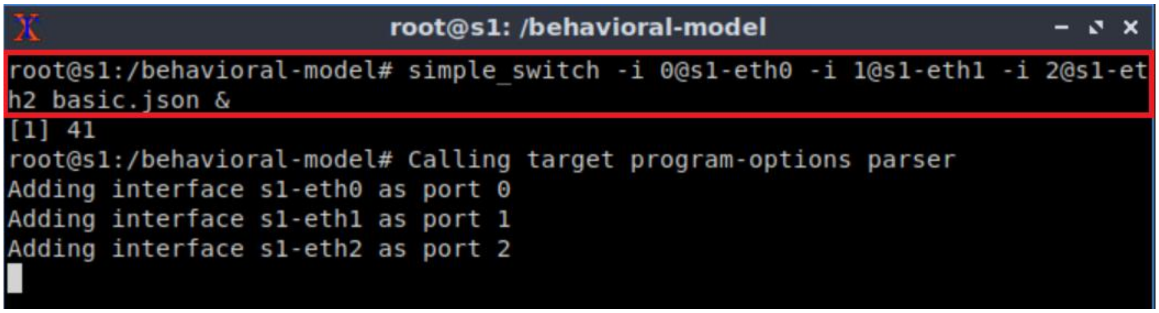
Figure 34. Displaying the content of the current directory in the switch s1.

We can see that the switch contains the *basic.json* file that was pushed previously after compiling the P4 program.

4.3 Mapping P4 program’s ports

Step 1. Start the switch daemon and map the logical interfaces (i.e., ports) to the switch’s interfaces by issuing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```



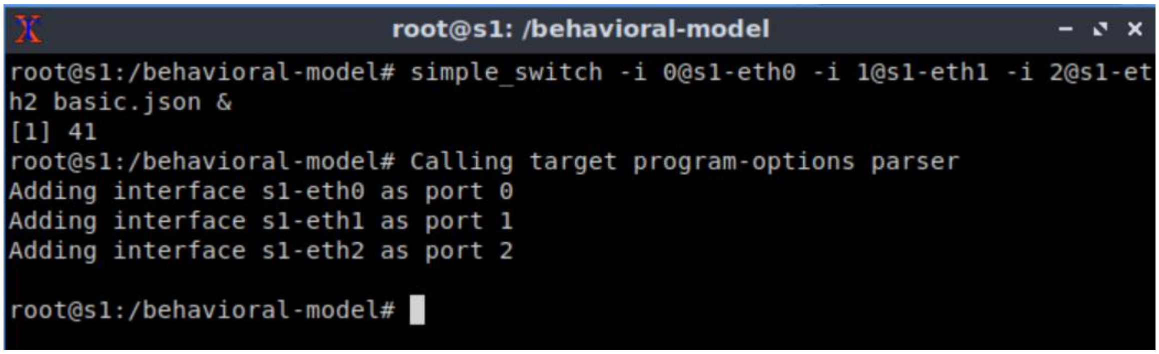
```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 41
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 35. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

5 Testing and verifying the P4 program

5.1 Configuring the policy rules

Step 1. In switch s1 terminal, press *Enter* to return the CLI.

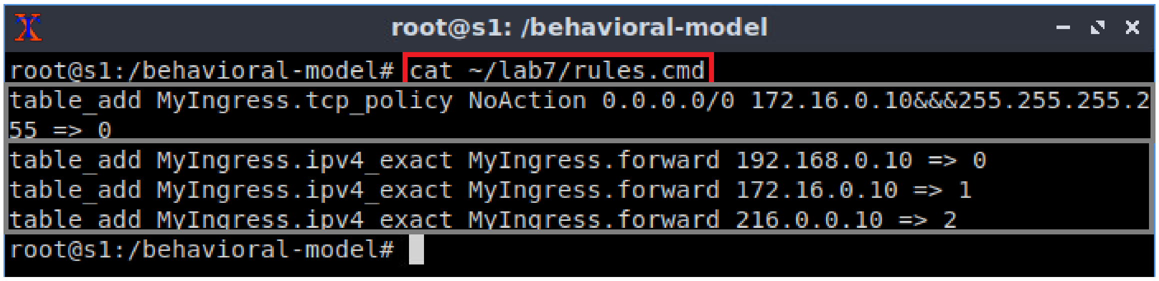


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 41
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model#
```

Figure 36. Returning to switch s1 CLI.

Step 2. Type the command below to inspect the content of the `rules.cmd` file. The file contains the rules which will be inserted to the switch at runtime. `cat` utility prints the content into the standard output.

```
cat ~/lab7/rules.cmd
```



```
root@s1:/behavioral-model# cat ~/lab7/rules.cmd
table_add MyIngress.tcp_policy NoAction 0.0.0.0/0 172.16.0.10&&&255.255.255.2
55 => 0
table_add MyIngress.ipv4_exact MyIngress.forward 192.168.0.10 => 0
table_add MyIngress.ipv4_exact MyIngress.forward 172.16.0.10 => 1
table_add MyIngress.ipv4_exact MyIngress.forward 216.0.0.10 => 2
root@s1:/behavioral-model#
```

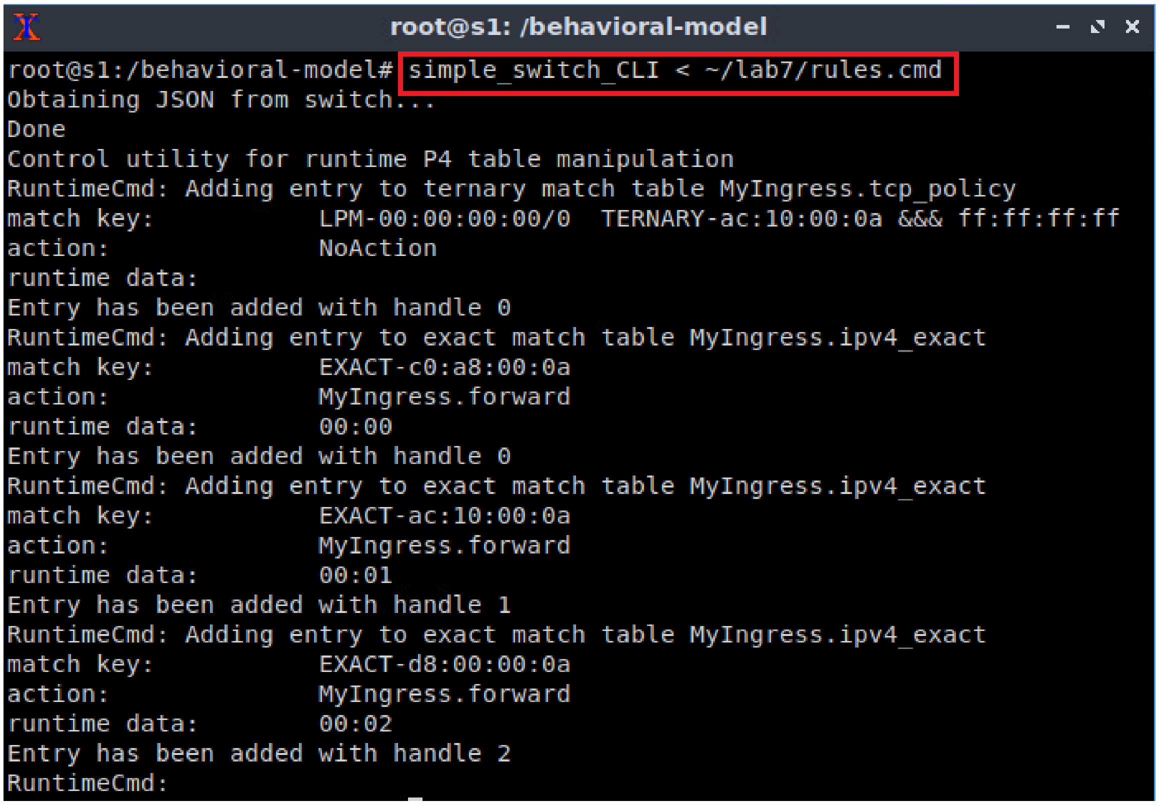
Figure 37. Inspecting the rules.

In the figure above, the first rule populates the `tcp_policy` table. `NoAction` keyword is used when a table does not include any actions in its definition. `0.0.0.0/0` is the LPM key which matches on any incoming packets. `172.16.0.10&&&255.255.255.255` is the ternary key which matches on any packet destined to the DMZ. Thus, the `tcp_policy` table will hit on any packet destined to the DMZ regardless of its source IP address. Note that any connection destined to the internal or external networks will not match the policy, and consequently, will be dropped.

The last three rules populate the `ipv4_exact` table with the forwarding rules.

Step 3. Push the table entries to the switch by typing the following command.

```
simple_switch_CLI < ~/lab7/rules.cmd
```



```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab7/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to ternary match table MyIngress.tcp_policy
match key:           LPM-00:00:00:00/0  TERNARY-ac:10:00:0a &&& ff:ff:ff:ff
action:              NoAction
runtime data:
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-c0:a8:00:0a
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-ac:10:00:0a
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.ipv4_exact
match key:           EXACT-d8:00:00:0a
action:              MyIngress.forward
runtime data:        00:02
Entry has been added with handle 2
RuntimeCmd:
```

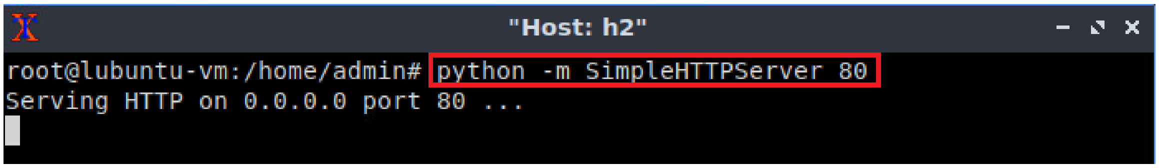
Figure 38. Pushing the table entries to the switch.

5.2 Testing TCP connections destined to the DMZ

In this section, an HTTP server will be configured on the DMZ. GET requests will be initiated from the internal and external networks. The requests should be successful because the policy accepts any connection destined to the DMZ.

Step 1. On h2 terminal, type the command below to start an HTTP server using Python. `-m` is used to run a module as a script, allowing the execution of Python module directly from the command line. `SimpleHTTPServer` is a Python 2 module that provides a basic HTTP server capable of serving static files from the current directory. The server will be listening on port `80` for incoming packets.

```
python -m SimpleHTTPServer 80
```

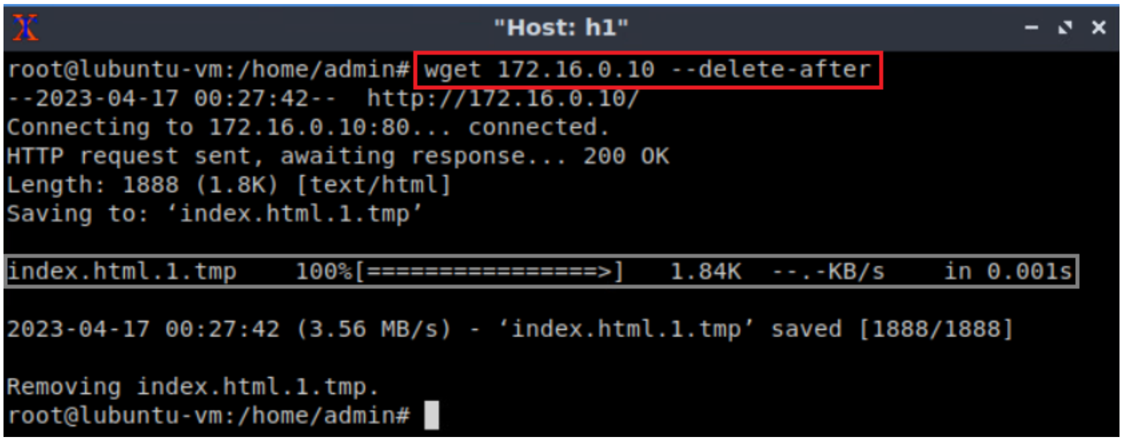


```
root@lubuntu-vm:/home/admin# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Figure 39. Starting HTTP server on h2 (DMZ).

Step 2. On h1 terminal, type the command below to issue an HTTP GET request. `wget` is a utility for non-interactive download of files from the Web. `172.16.0.10` is the IP address of the HTTP server running on host h2. `--delete-after` option tells Wget to delete every single file it downloads, after having done so.

```
wget 172.16.0.10 --delete-after
```



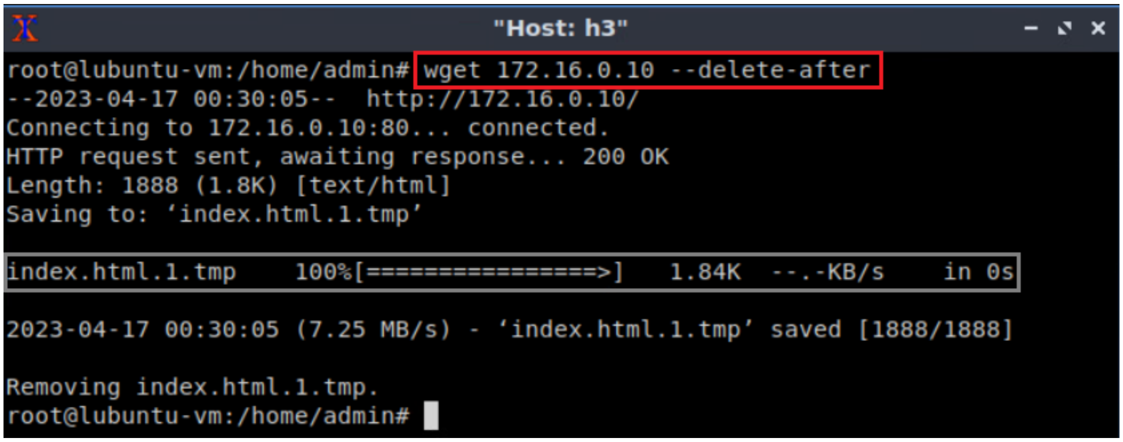
```
root@lubuntu-vm:/home/admin# wget 172.16.0.10 --delete-after
--2023-04-17 00:27:42--  http://172.16.0.10/
Connecting to 172.16.0.10:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1888 (1.8K) [text/html]
Saving to: ‘index.html.1.tmp’

index.html.1.tmp    100%[================>]   1.84K  --.-KB/s    in 0.001s

2023-04-17 00:27:42 (3.56 MB/s) - ‘index.html.1.tmp’ saved [1888/1888]

Removing index.html.1.tmp.
root@lubuntu-vm:/home/admin#
```

Figure 40. Issuing HTTP GET requests from h1 (internal network).

The figure above shows that the request was successful.

Step 3. On h3 terminal, type the command below to issue an HTTP Get request.

```
wget 172.16.0.10 --delete-after
```



```
root@lubuntu-vm:/home/admin# wget 172.16.0.10 --delete-after
--2023-04-17 00:30:05--  http://172.16.0.10/
Connecting to 172.16.0.10:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1888 (1.8K) [text/html]
Saving to: ‘index.html.1.tmp’

index.html.1.tmp    100%[================>]   1.84K  --.-KB/s    in 0s

2023-04-17 00:30:05 (7.25 MB/s) - ‘index.html.1.tmp’ saved [1888/1888]

Removing index.html.1.tmp.
root@lubuntu-vm:/home/admin#
```

Figure 41. Issuing HTTP GET requests from h3 (external network).

The figure above shows that the request was successful.

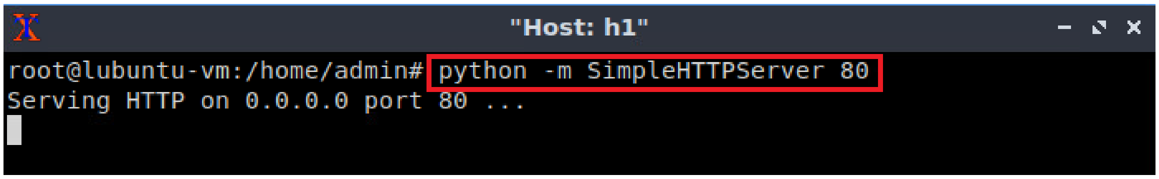
Step 4. On h2 terminal, press `Ctrl + c` to stop the server.

5.3 Testing TCP connections destined to the internal network

In this section, an HTTP server will be configured on the internal network. GET requests will be initiated from the external network and the DMZ. The requests should not be successful because the policy drops any connection destined to the internal network.

Step 1. On h1 terminal, type the command below to start an HTTP server using Python.

```
python -m SimpleHTTPServer 80
```

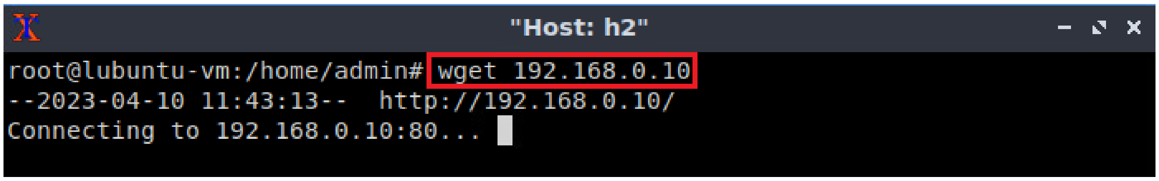


```
root@lubuntu-vm:/home/admin# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Figure 42. Starting HTTP server on h1 (internal network).

Step 2. On h2 terminal, type the command below to issue an HTTP GET request.

```
wget 192.168.0.10
```



```
root@lubuntu-vm:/home/admin# wget 192.168.0.10
--2023-04-10 11:43:13--  http://192.168.0.10/
Connecting to 192.168.0.10:80...
```

Figure 43. Issuing HTTP GET requests from h2 (DMZ).

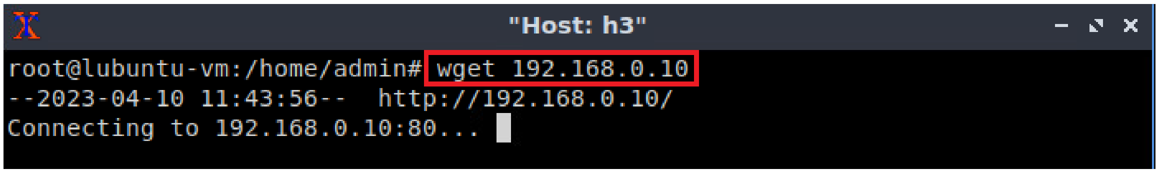
The figure above shows that the request was not successful because the switch blocked the connection.

Note that the connection will not be dropped by h2, as it will retry to connect multiple times. You should manually terminate the connection.

Step 3. On host h2, press `Ctrl + c` to terminate the HTTP GET request.

Step 4. On h3 terminal, type the command below to issue an HTTP Get request.

```
wget 192.168.0.10
```



```
root@lubuntu-vm:/home/admin# wget 192.168.0.10
--2023-04-10 11:43:56--  http://192.168.0.10/
Connecting to 192.168.0.10:80...
```

Figure 44. Issuing HTTP GET requests from h3 (external network).

The figure above shows that the request was not successful because the switch blocked the connection.

Step 5. On host h3, press `Ctrl + c` to terminate the HTTP GET request.

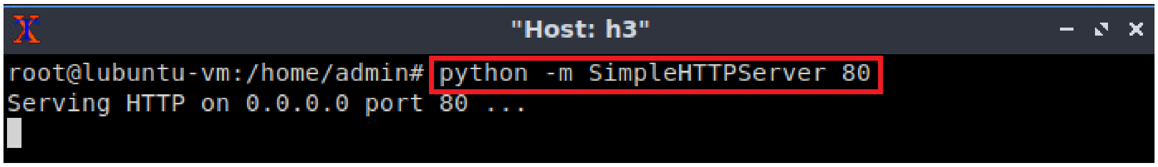
Step 6. On h1 terminal, press `Ctrl + c` to stop the server.

5.4 Testing TCP connections destined to the external network

In this section, an HTTP server will be configured on the internal network. GET requests will be initiated from the internal network and the DMZ. The requests should not be successful because the policy drops any connection destined to the external network.

Step 1. On h3 terminal, type the command below to start an HTTP server using Python.

```
python -m SimpleHTTPServer 80
```

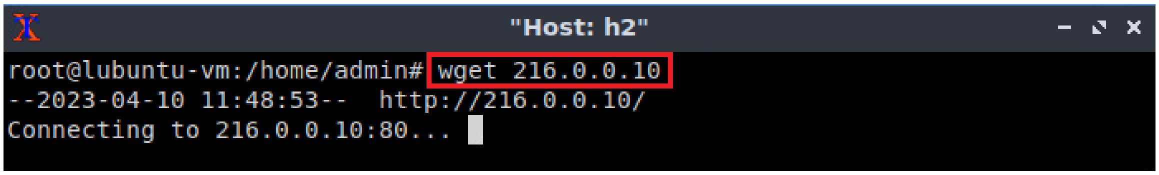


```
root@lubuntu-vm:/home/admin# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Figure 45. Starting HTTP server on h3 (external network).

Step 2. On h2 terminal, type the command below to issue an HTTP Get request.

```
wget 216.0.0.10
```



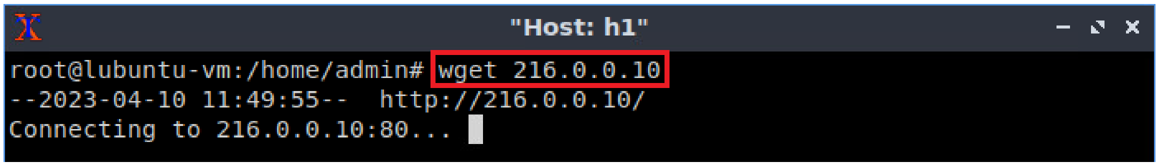
```
root@lubuntu-vm:/home/admin# wget 216.0.0.10
--2023-04-10 11:48:53--  http://216.0.0.10/
Connecting to 216.0.0.10:80...
```

Figure 46. Issuing HTTP GET requests from h2 (DMZ).

The figure above shows that the request was not successful because the switch blocked the connection.

Step 3. On h1 terminal, type the command below to issue an HTTP Get request.

```
wget 216.0.0.10
```



```
root@lubuntu-vm:/home/admin# wget 216.0.0.10
--2023-04-10 11:49:55--  http://216.0.0.10/
Connecting to 216.0.0.10:80...
```

Figure 47. Issuing HTTP GET requests from h1 (internal network).

The figure above shows that the request was not successful because the switch blocked the connection.

This concludes lab 7. Stop the emulation and then exit out of MiniEdit.

References

1. M. Rouse. “Packet Filtering.” [Online]. Available: https://tinyurl.com/8z4a2yp6
2. Diyaroy. “Stateless vs Stateful Packet Filtering Firewalls” Online]. Available: https://tinyurl.com/3s2twcdp
3. P4-guide github repository*. “Demo Global Register P4$_{16}$*.” [Online]. Available: https://tinyurl.com/mrytj9ad
4. P4 Language Tutorial. [Online]. Available: https://tinyurl.com/2p9cen9e.
5. P4lang/behavioral-model github repository*. “The BMv2 simple switch target*.” [Online]. Available: https://tinyurl.com/vrasamm.

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 8: Detecting and Mitigating the DNS Amplification Attack

Document Version: 04-18-2023

Contents

Overview

This lab introduces the DNS amplification attack and provides the steps to implement a P4 program to mitigate the attack. In a DNS amplification attack, the attacker floods the victim with DNS responses by utilizing a DNS resolver. To mitigate this attack, the user will use P4' registers to store the transaction ID of the DNS queries issued by the victim. Any DNS response with transaction ID not stored by the switch will be dropped.

Objectives

By the end of this lab, students should be able to:

1. Define DNS amplification attack.
2. Understand the workflow of the DNS amplification attack.
3. Perform a DNS amplification attack.
4. Write a P4 program that mitigates the DNS amplification attack.

Lab settings

Table 1 contains the credentials of the virtual machine used for this lab.

Table 1**.** Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Loading a basic P4 program.
4. Section 4: Performing DNS amplification attack.
5. Section 5: Modifying the P4 program to mitigate DNS amplification.

1 Introduction

Domain Name System (DNS) is an essential component of the internet, responsible for translating human-readable domain names (e.g., www.example.com) into IP addresses (e.g., 192.0.2.1) that can be understood by devices connected to the internet[1]. Recursive

DNS servers are responsible for resolving these domain names on behalf of clients and caching the results for subsequent requests.

DNS Amplification is a type of Distributed Denial of Service (DDoS) attack that exploits the DNS infrastructure to amplify the amount of traffic directed towards a target system, overwhelming its resources, and causing it to become unresponsive[2]. In a DNS amplification attack, the attacker sends a large number of spoofed DNS query packets to vulnerable, open recursive DNS servers. These packets have a forged source IP address set to the target's IP. As a result, when the DNS server responds to the query, it sends a much larger response packet to the target, rather than the actual source of the query.

Attackers typically use small query packets with a high amplification factor, meaning that the response packets are considerably larger in size than the query packets[2]. This amplification effect allows attackers to generate a massive volume of traffic with relatively minimal resources, amplifying the impact of the attack on the targeted system. Mitigating DNS Amplification attacks requires a combination of strategies, including securing open recursive DNS servers, implementing rate limiting on DNS queries, and employing traffic filtering techniques to identify and block malicious traffic[3].

1.1 Lab scenario

In this lab, a P4 programmable switch will mitigate the DNS amplification attack by dropping the DNS responses that do not match DNS requests. The switch will use the hash of the 5-tuple (source IP, destination IP, source port, destination port, and transport protocol) to index flows in a register. The DNS transaction ID will be stored in the cell of the register. The transaction ID is generated by the client sending a DNS request.

Figure 1 depicts a DNS amplification attack scenario. The attacker is performing DNS amplification attack by using the IP of the victim as the source IP of the packets. The DNS server responds to malicious requests and sends the replies to the victim. The victim is flooded with DNS replies from the server.

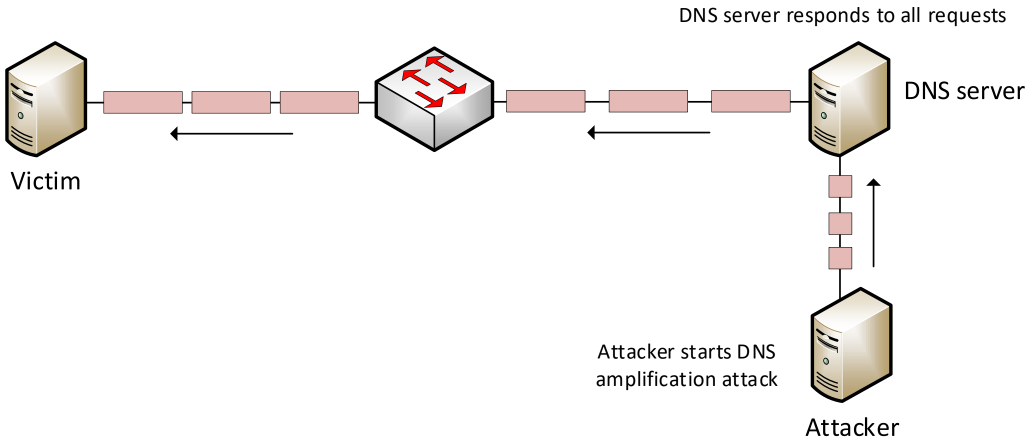

DNS server responds to all requests
DNS server
Victim
Attacker starts DNS amplification attack
Attacker

Figure 1. DNS amplification attack.

Figure 2 depicts the DNS amplification attack mitigation using P4 switch. The attacker is performing DNS amplification attack by using the IP of the victim as the source IP of the packets. The DNS server responds to the malicious requests and sends the replies to the victim. The P4 switch drops all the replies as they do not associate with DNS requests issued by the victim. After that, the victim sends a DNS query to the DNS server. The DNS server responds to the legitimate request and the P4 switch forward the legitimate reply to the victim. Note that the P4 switch forwarded the legitimate reply while dropping all the replies resulting from the attack.

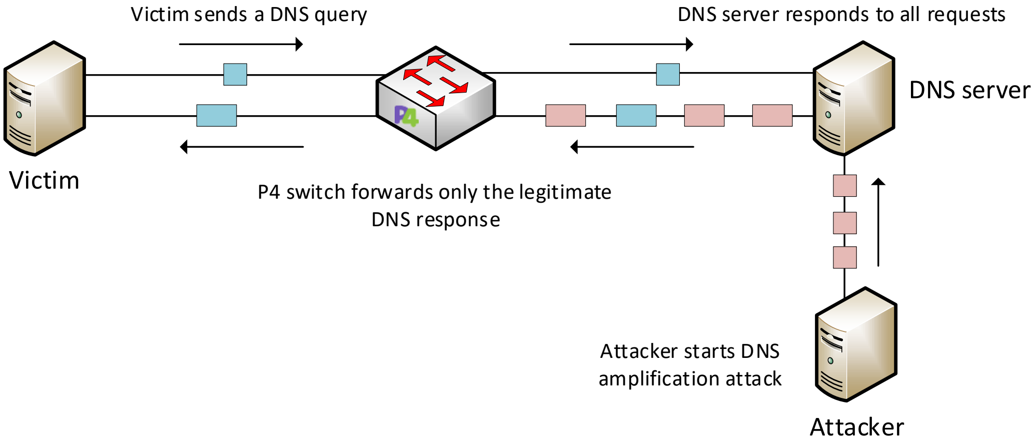

Victim sends a DNS query
DNS server responds to all requests
DNS server
Victim
P4
P4 switch forwards only the legitimate DNS response
Attacker starts DNS amplification attack
Attacker

Figure 2. DNS amplification attack mitigation using a P4 switch.

The P4 programmable switch identifies DNS requests by inspecting the *dns_qr* header of DNS packets[4]. *dns_qr* = 0 means that the packet is a DNS request. *dns_qr* = 1 means that the packet is a DNS response. For DNS requests, the switch stores their transaction IDs in register cells. The hash of the 5-tuple of the request will be used as an index to the register. The 5-tuple is hashed in the following order: source IP, destination IP, source port, destination port, and transport protocol. For DNS responses, the switch hashes the 5-tuple. The 5-tuple is hashed in the following order: destination IP, source IP, source port, destination port, and transport protocol. The source IP and destination IP are reversed so that the DNS response maps to the same cell as the DNS request. After retrieving the transaction ID from the cell, the ID is compared to the transaction ID of the DNS response. If the values match, the switch forwards the packet. Otherwise, the switch drops the packet.

2 Lab topology

Let’s get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

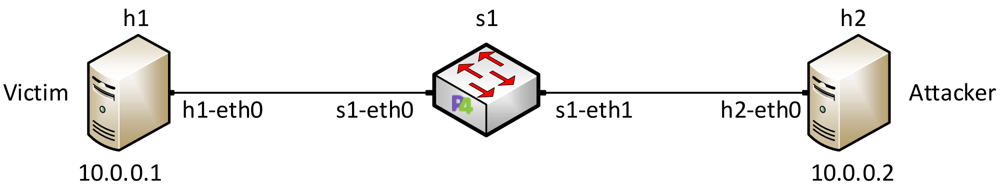


Figure 3. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

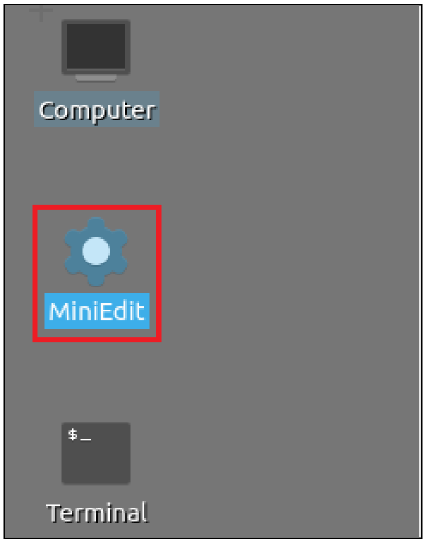
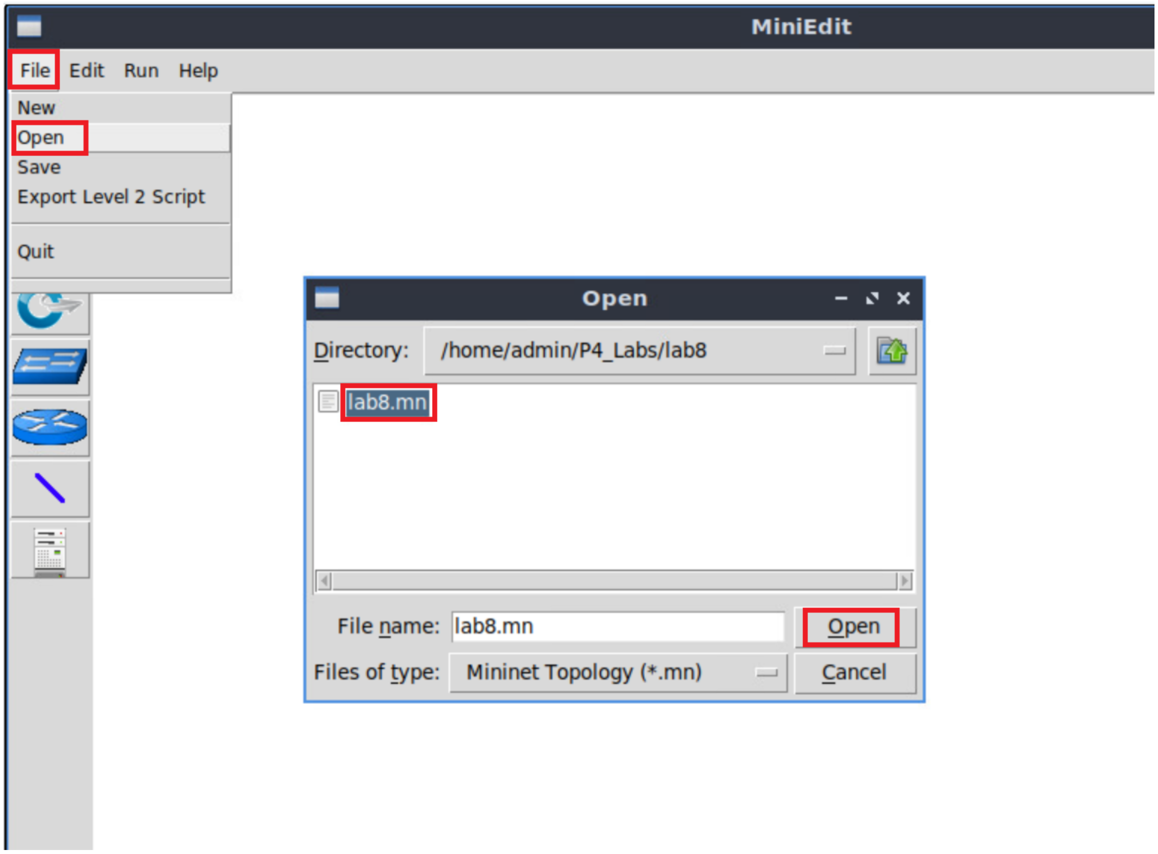


Figure 4. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab8*, select the file *lab8.mn,* and click on *Open*.



Directory: /home/admin/P4_Labs/lab8

lab8.mn

File name: lab8.mn

Files of type: Mininet Topology (*.mn)

Figure 5. Opening a topology in MiniEdit.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

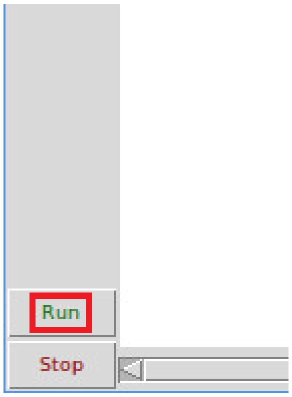


Figure 6. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

Step 1. Hold the right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

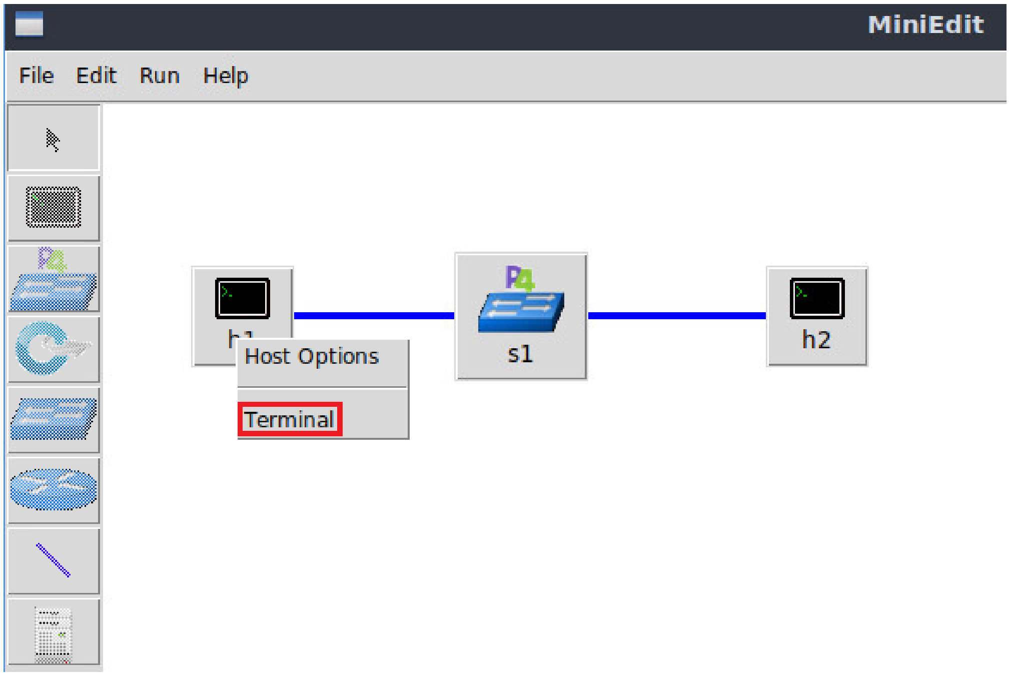
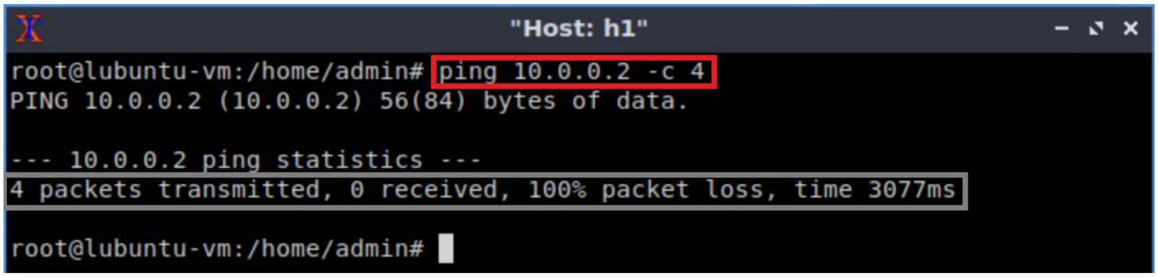


Figure 7. Opening a terminal on host h1.

Step 2. Test the connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3077ms

root@lubuntu-vm:/home/admin#
```

Figure 8. Performing a connectivity test between host h1 and host h2.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded into the switch.

3 Loading the basic P4 program

In this section, the user will compile and run a P4 program that implements the basic forwarding functionality. The switch will then be configured by mapping the P4 program's ports and loading the rules to the switch.

Step 1. Launch a Linux terminal by double-clicking on the Linux terminal icon located on the desktop.

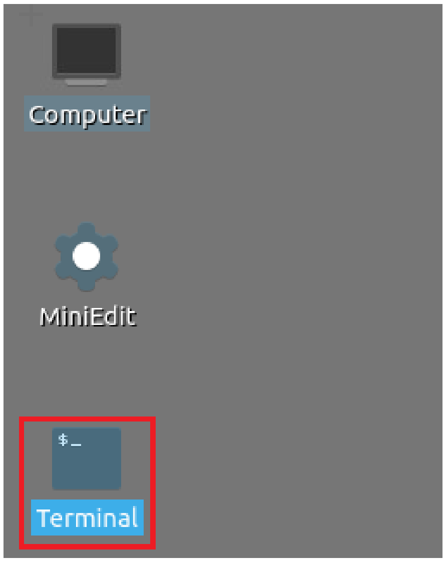
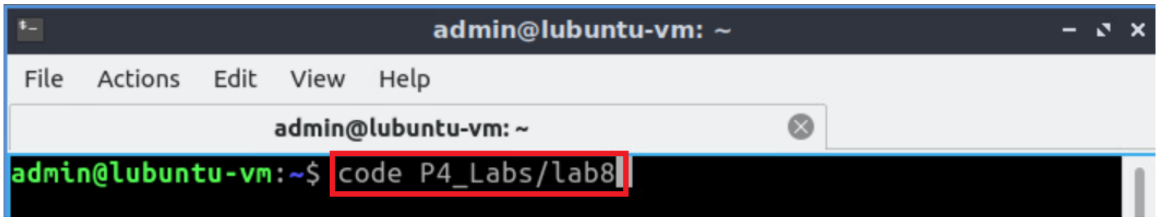


Figure 9. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the VS Code and opens the directory where the P4 program for this lab is located.

```
code P4_Labs/lab8
```

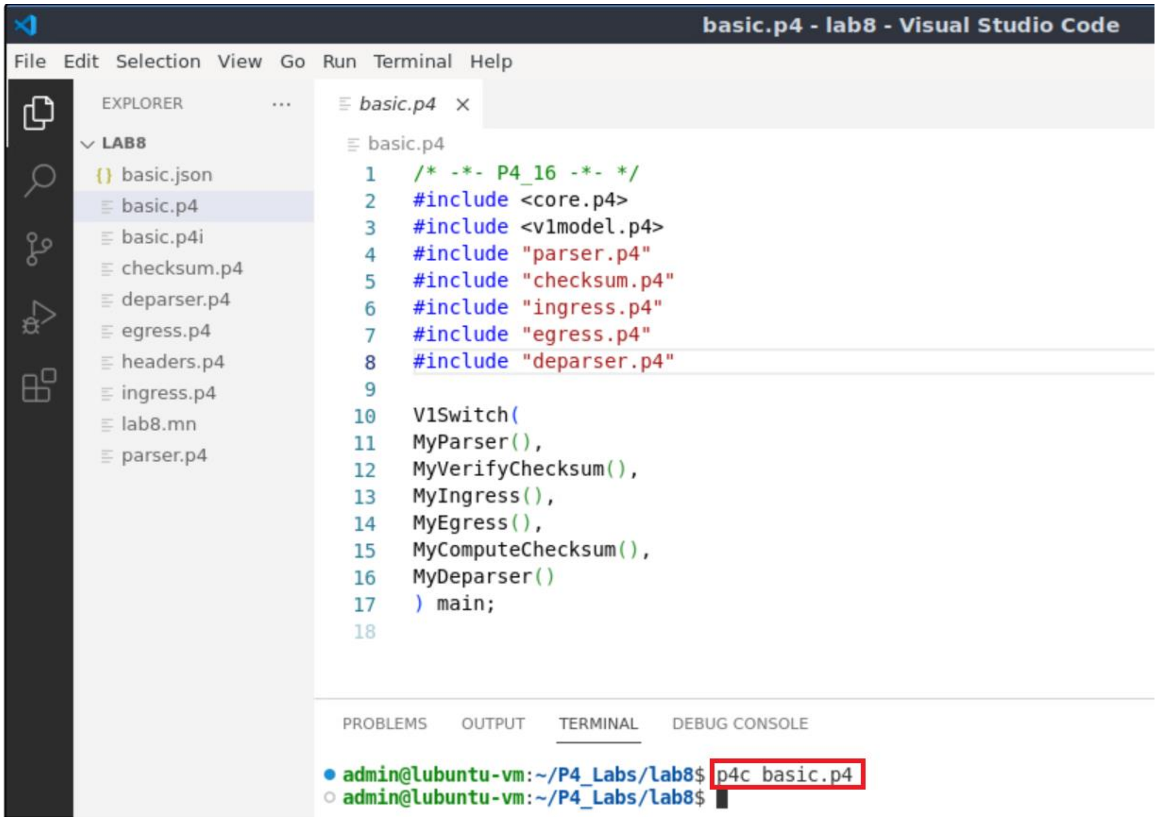


```
admin@lubuntu-vm:~$ code P4_Labs/lab8
```

Figure 10. Launching the editor and opening the lab8 directory.

Step 3. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```


basic.p4 - lab8 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER ...
LAB8
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab8.mn
parser.p4
basic.p4
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

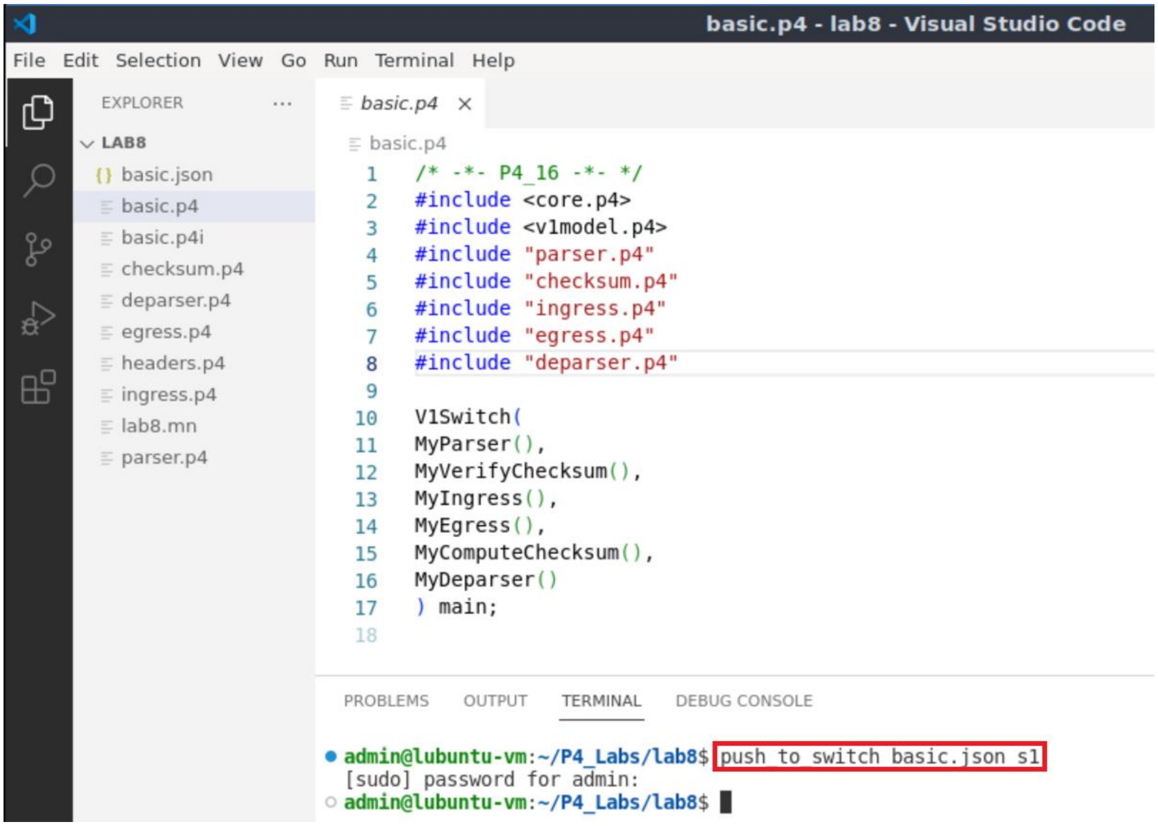
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab8$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab8$
```

Figure 11. Compiling the P4 program using the VS Code terminal.

Step 4. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. The script accepts as input the JSON output of the *p4c* compiler, and the target switch name (e.g., s1). If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```

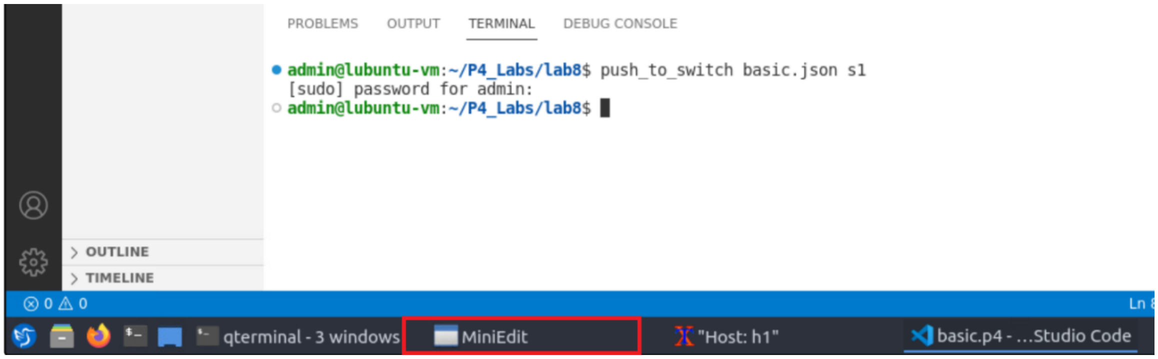

basic.p4 - lab8 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB8
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab8.mn
parser.p4
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab8$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab8$
```

Figure 12. Downloading the compiled program to switch s1.

Step 5. Click on the MinEdit tab in the start bar to maximize the window.


PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
OUTLINE
TIMELINE
⊗ 0 ⚠ 0
qterminal - 3 windows
MiniEdit
"Host: h1"
basic.p4 - ...Studio Code

```
admin@lubuntu-vm:~/P4_Labs/lab8$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab8$
```

Figure 13. Maximizing the MiniEdit window.

Step 6. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

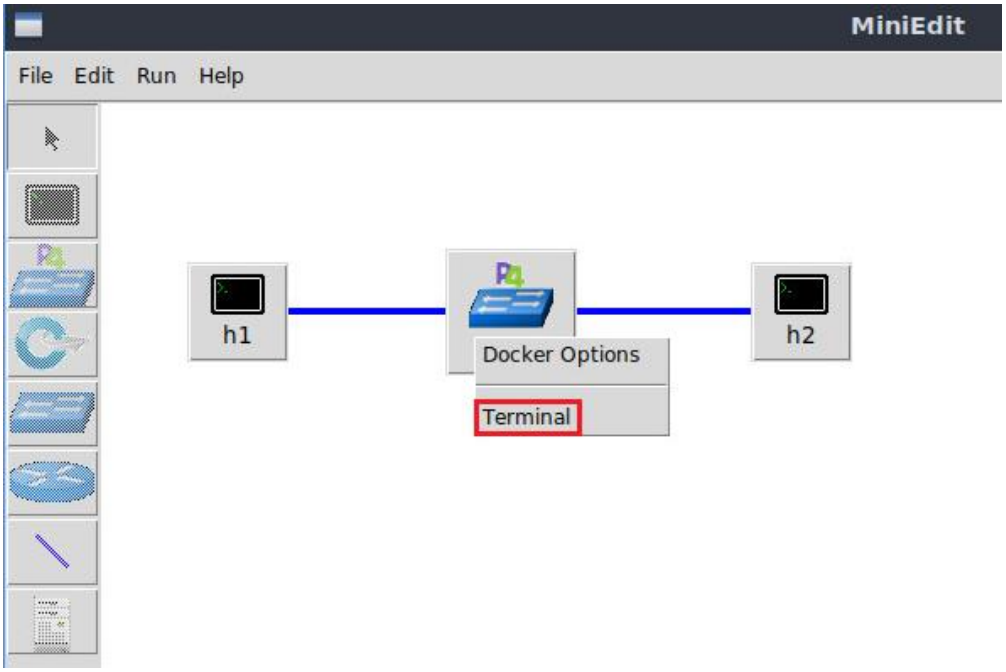
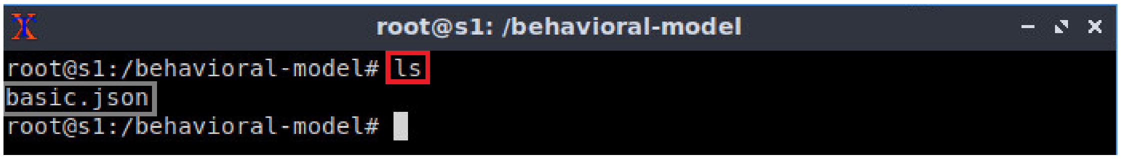


Figure 14. Starting the terminal on switch s1.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch's terminal.

Step 7. Issue the following command to list the files in the current directory.

```
ls
```



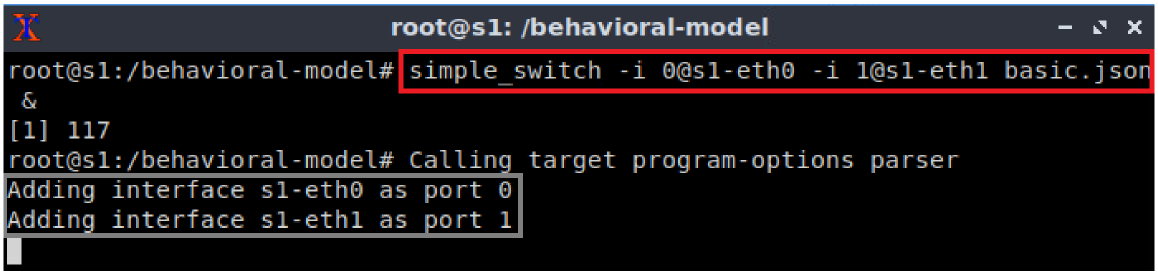
```
root@s1: /behavioral-model
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

Figure 15. Displaying the contents of the current directory in the switch s1.

We can see that the switch contains the *basic.json* file that was downloaded after compiling the P4 program.

Step 8. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
```



```
root@s1: /behavioral-model
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 117
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 16. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

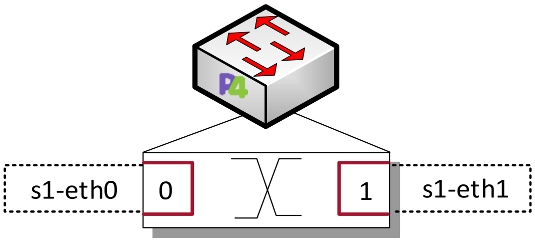
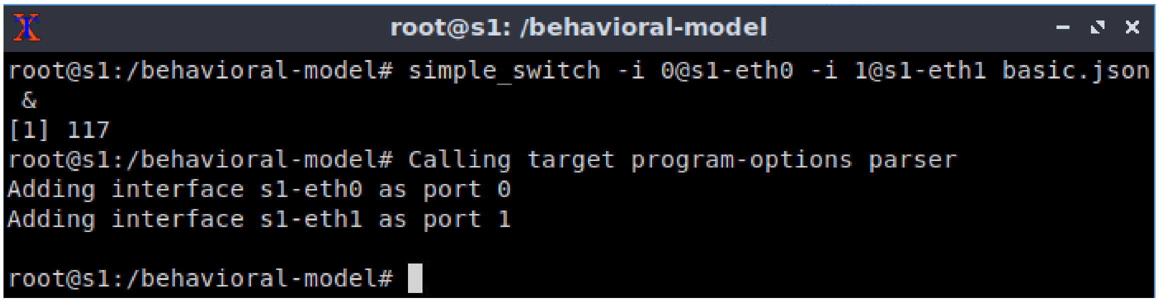


Figure 17. Ports 0 and 1 are mapped to the interfaces *s1-eth0* and *s1-eth1* of switch s1.

Step 9. In switch s1 terminal, press *Enter* to return the CLI.

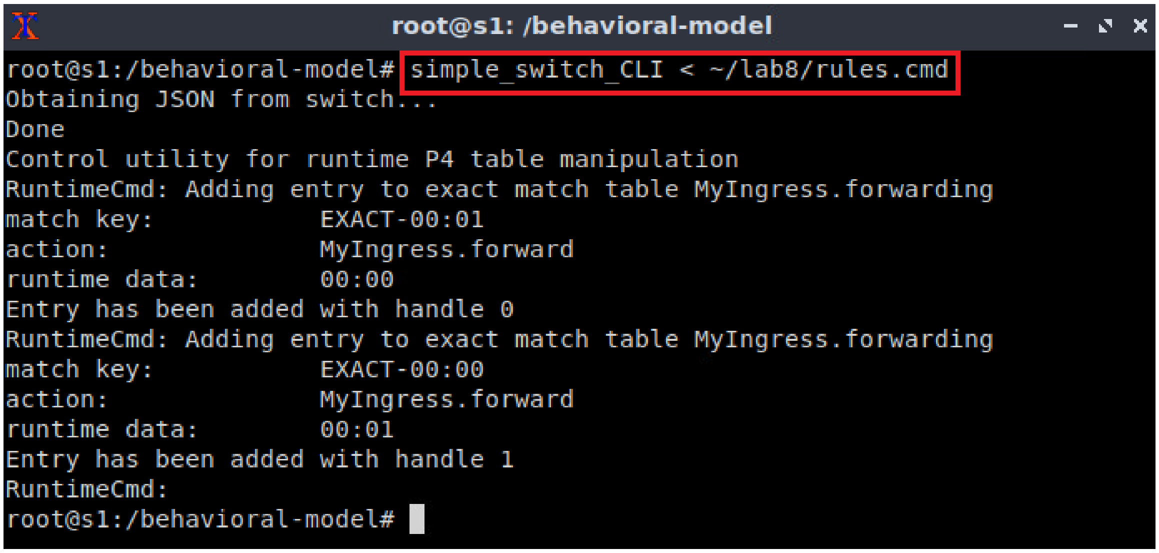


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 117
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 18. Returning to switch s1 CLI.

Step 10. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab8/rules.cmd
```



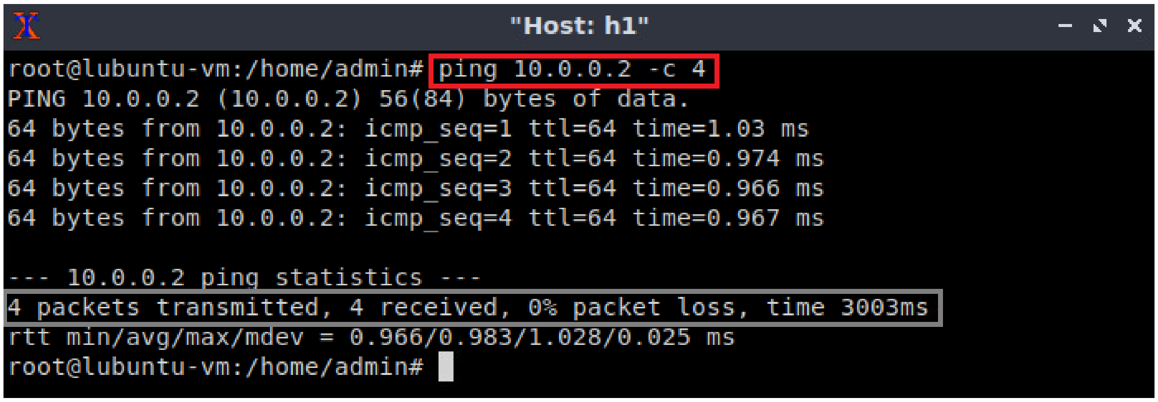
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab8/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# 
```

Figure 19. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

Step 11. Go back to host h1 terminal to test the connectivity between host h1 and host h2 by issuing the following command.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.03 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.974 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.966 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.967 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 0.966/0.983/1.028/0.025 ms
root@lubuntu-vm:/home/admin# 
```

Figure 20. Performing a connectivity test between host h1 and host h2.

The figure above shows that there is connectivity between the two hosts.

4 Performing DNS amplification attack

4.1 Starting and testing the DNS server

Step 1. Hold the right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

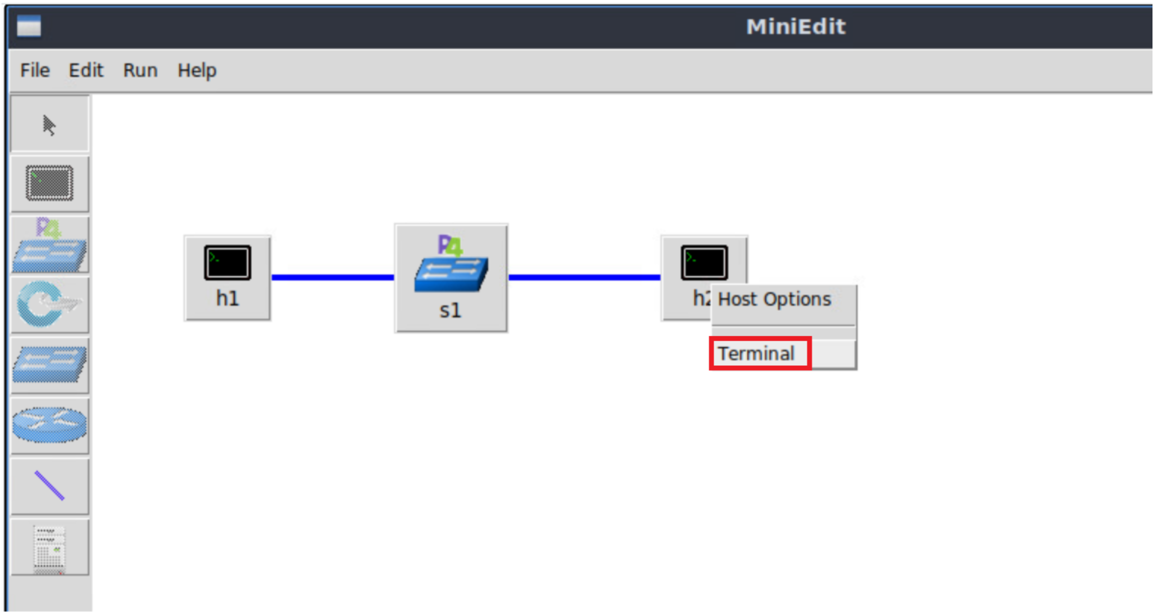
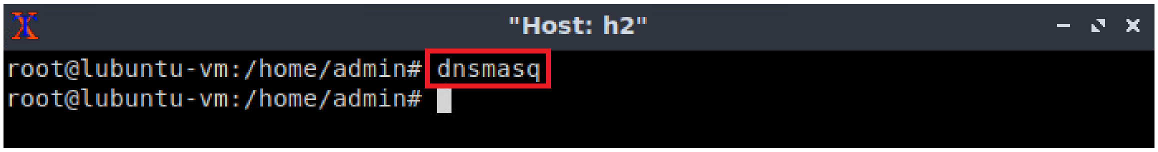


Figure 21. Opening a terminal on host h2.

Step 2. Start a DNS server on h2 by issuing the command below. `dnsmasq` command starts a lightweight DNS server.

```
dnsmasq
```

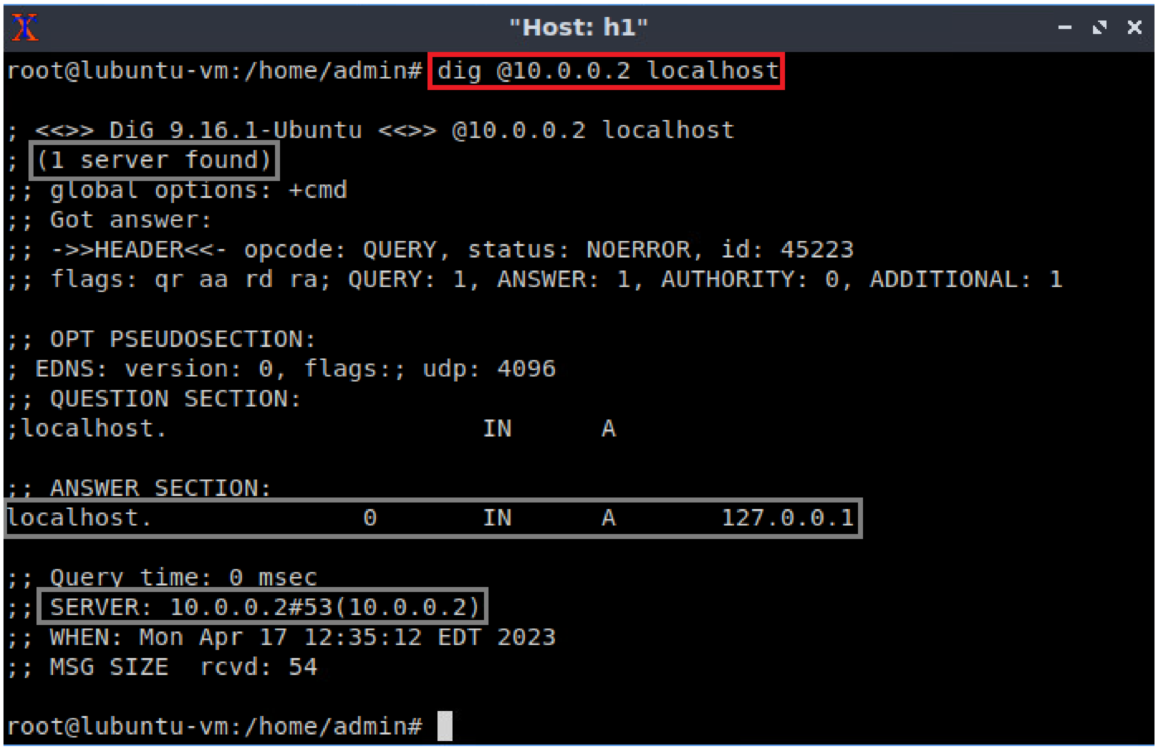


```
root@lubuntu-vm:/home/admin# dnsmasq
root@lubuntu-vm:/home/admin# 
```

Figure 22. Starting the DNS server on h2.

Step 3. On h1 terminal, type the command below to validate that h2 operates as a DNS server. `dig` (domain information groper) is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. `@10.0.0.2` is the IP address of the DNS server which is running on h2. `localhost` is the target of the DNS query.

```
dig @10.0.0.2 localhost
```



```
root@lubuntu-vm:/home/admin# dig @10.0.0.2 localhost

; <<>> DiG 9.16.1-Ubuntu <<>> @10.0.0.2 localhost
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45223
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;localhost.                     IN      A

;; ANSWER SECTION:
localhost.              0       IN      A       127.0.0.1

;; Query time: 0 msec
;; SERVER: 10.0.0.2#53(10.0.0.2)
;; WHEN: Mon Apr 17 12:35:12 EDT 2023
;; MSG SIZE  rcvd: 54

root@lubuntu-vm:/home/admin# 
```

Figure 23. Issuing DNS query.

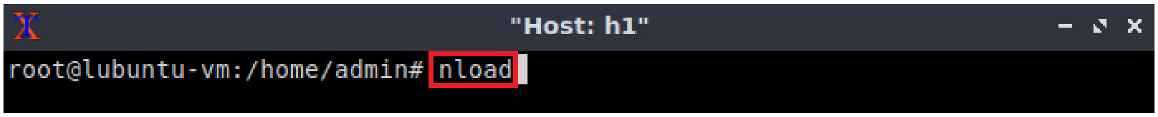
The figure above shows that 1 DNS server is detected. The IP address of the server is 10.0.0.2. The answer section contains the IP address of the localhost (127.0.0.1).

4.2 Performing the attack

In this section, the DNS server will flood h1 with DNS responses. The responses are stored inside a PCAP file.

Step 1. On h1 terminal, type the command below to display the current network usage. `nload` is a console application that monitors network traffic and bandwidth usage in real time.

```
nload
```

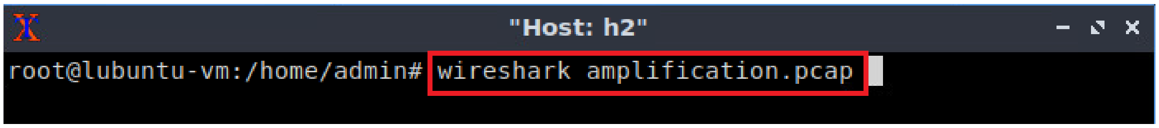


```
root@lubuntu-vm:/home/admin# nload
```

Figure 24. Staring nload on h1.

Step 2. On h2 terminal, type the command below to open the file `amplification.pcap` using Wireshark.

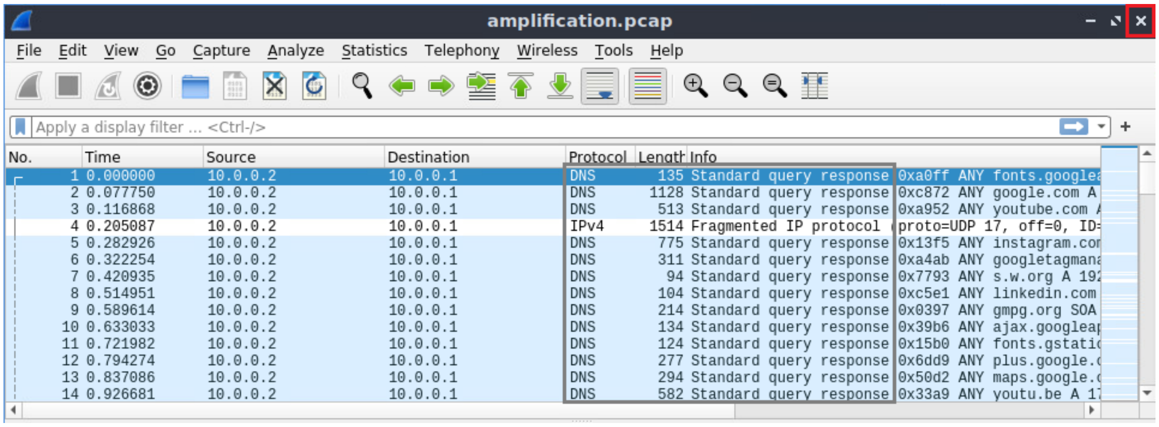
```
wireshark amplification.pcap
```



```
root@lubuntu-vm:/home/admin# wireshark amplification.pcap
```

Figure 25. Opening amplification.pcan file using Wireshark.

Step 3. Inspect the content of the *amplification.pcap* file. Close Wireshark by clicking on the ☒ icon at the top right corner.



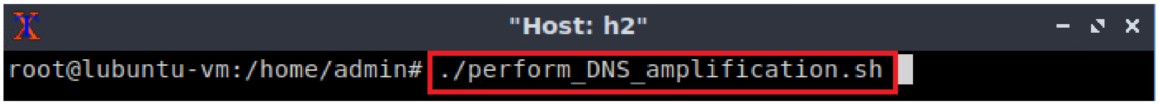
```
No.  Time      Source    Destination Protocol Length Info
   1 0.000000  10.0.0.2  10.0.0.1    DNS       135 Standard query response 0xa0ff ANY fonts.googlea
   2 0.077750  10.0.0.2  10.0.0.1    DNS      1128 Standard query response 0xc872 ANY google.com A
   3 0.116868  10.0.0.2  10.0.0.1    DNS       513 Standard query response 0xa952 ANY youtube.com A
   4 0.205087  10.0.0.2  10.0.0.1    IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=
   5 0.282926  10.0.0.2  10.0.0.1    DNS       775 Standard query response 0x13f5 ANY instagram.com
   6 0.322254  10.0.0.2  10.0.0.1    DNS       311 Standard query response 0xa4ab ANY googletagmana
   7 0.420935  10.0.0.2  10.0.0.1    DNS        94 Standard query response 0x7793 ANY s.w.org A 192
   8 0.514951  10.0.0.2  10.0.0.1    DNS       104 Standard query response 0xc5e1 ANY linkedin.com
   9 0.589614  10.0.0.2  10.0.0.1    DNS       214 Standard query response 0x0397 ANY gmpg.org SOA
  10 0.633033  10.0.0.2  10.0.0.1    DNS       134 Standard query response 0x39b6 ANY ajax.googleap
  11 0.721982  10.0.0.2  10.0.0.1    DNS       124 Standard query response 0x15b0 ANY fonts.gstatic
  12 0.794274  10.0.0.2  10.0.0.1    DNS       277 Standard query response 0x6dd9 ANY plus.google.c
  13 0.837086  10.0.0.2  10.0.0.1    DNS       294 Standard query response 0x50d2 ANY maps.google.c
  14 0.926681  10.0.0.2  10.0.0.1    DNS       582 Standard query response 0x33a9 ANY youtu.be A 17
```

Figure 26. Inspecting the content of *amplification.pcap* file.

The figure above shows that all the packets inside the file are DNS responses. The packets of this file will be replayed to emulate a DNS amplification attack.

Step 4. On h2 terminal, type the command below to perform DNS amplification attack.

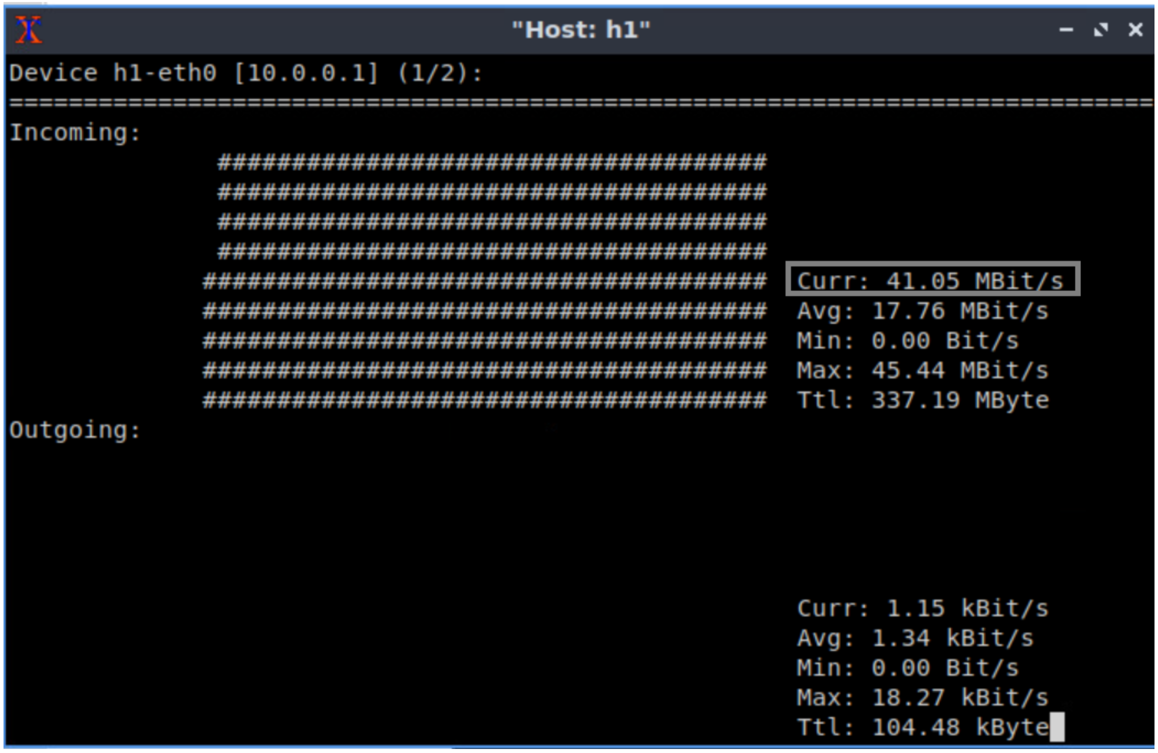
```
./perform_DNS_amplification.sh
```



```
root@lubuntu-vm:/home/admin# ./perform_DNS_amplification.sh
```

Figure 27. Performing the DNS amplification attack.

Step 5. Inspect the network usage at h1.



```
Device h1-eth0 [10.0.0.1] (1/2):
=============================================================================
Incoming:
              #####################################
              #####################################
              #####################################
              #####################################
             ######################################  Curr: 41.05 MBit/s
             ######################################  Avg: 17.76 MBit/s
             ######################################  Min: 0.00 Bit/s
             ######################################  Max: 45.44 MBit/s
             ######################################  Ttl: 337.19 MByte
Outgoing:




                                                     Curr: 1.15 kBit/s
                                                     Avg: 1.34 kBit/s
                                                     Min: 0.00 Bit/s
                                                     Max: 18.27 kBit/s
                                                     Ttl: 104.48 kByte
```

Figure 28. Inspecting resource usage at h1.

The figure above shows the increase in network usage caused by the DNS amplification attack.

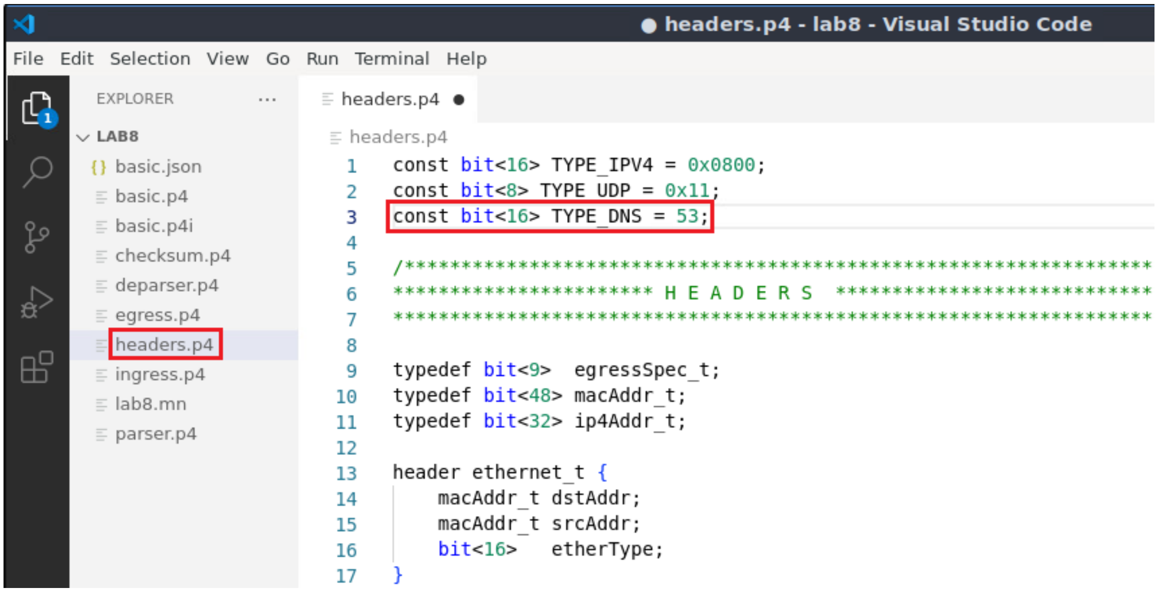
5 Modifying the P4 program to mitigate DNS amplification

In this section, the P4 program will be modified to mitigate DNS amplification attacks. To do this, the DNS header will be added to the header file. Then, the parser will be modified to extract the DNS header from incoming packets. After that the ingress block will be modified to drop all the DNS responses that do not belong to DNS requests initiated by h1. Finally, the P4 program will be tested by performing the DNS amplification attack and observing the network usage at h1.

5.1 Modifying the headers file

Step 1. Use VScode to access the *header.p4* file. In the *header.p4* file, add the following constant.

```
const bit<16> TYPE_DNS = 53;
```



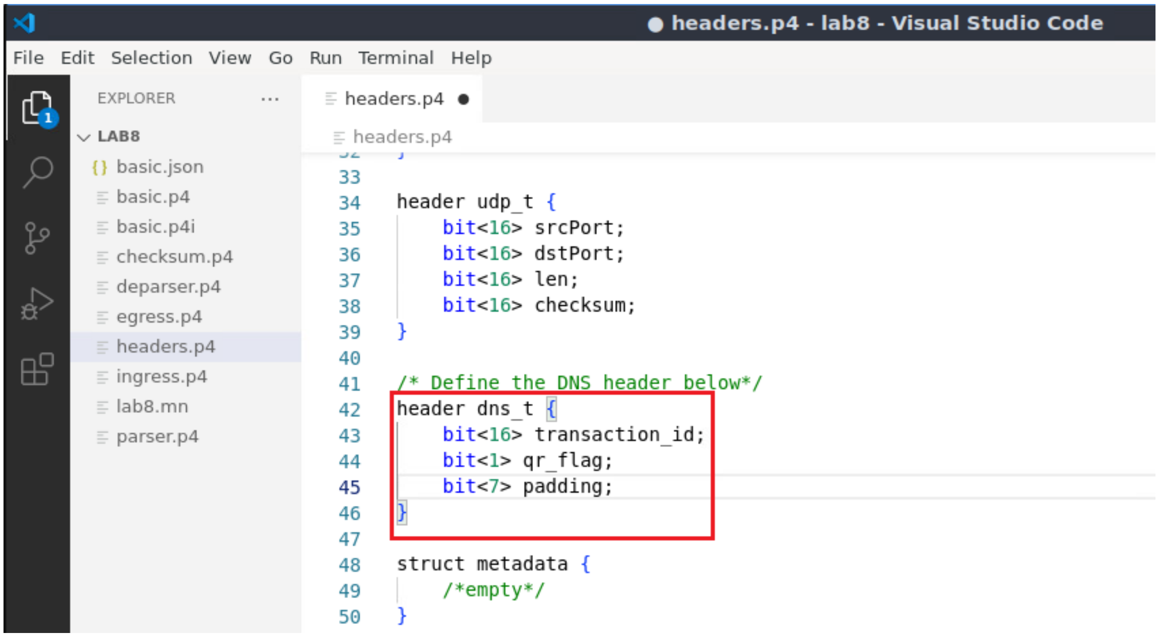
```
const bit<16> TYPE_IPV4 = 0x0800;
const bit<8> TYPE_UDP = 0x11;
const bit<16> TYPE_DNS = 53;

/*************************************************************************
*********************** 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;
}
```

Figure 29. Adding TYPE_DNS constant.

In the figure above, `DNS_TYPE` represents the port number used by DNS queries. All UDP packets with source port or destination port of `DNS_TYPE` (i.e., 53) are DNS packets.

Step 2. In the *header.p4* file, define the DNS header under the UDP header by adding the following code.

```
header dns_t{
   bit <16> transaction_id;
   bit <1> qr_flag;
   bit <7> padding;
}
```



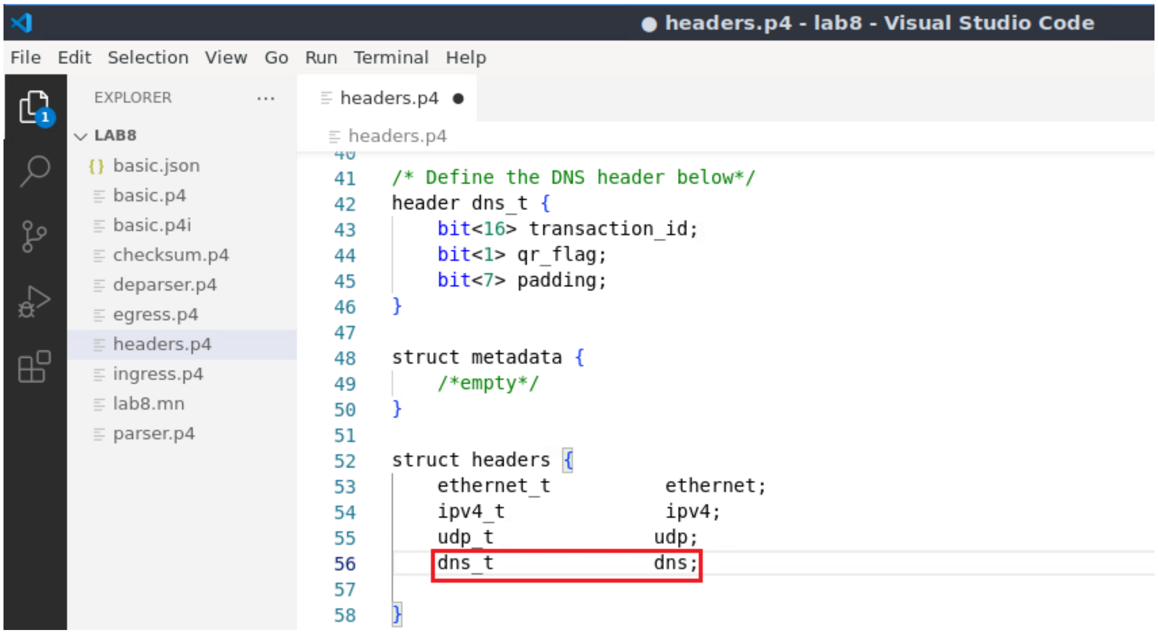
```

header udp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> len;
    bit<16> checksum;
}

/* Define the DNS header below*/
header dns_t {
    bit<16> transaction_id;
    bit<1> qr_flag;
    bit<7> padding;
}

struct metadata {
    /*empty*/
}
```

Figure 30. Defining the DNS header.

Note that in the code above, the padding field was added because headers in P4 should be byte aligned (the length of headers in P4 should be a multiple of 8).

Step 3. Add the DNS header to the headers struct by typing the following code.

```
dns_t dns;
```



```
/* Define the DNS header below*/
header dns_t {
    bit<16> transaction_id;
    bit<1> qr_flag;
    bit<7> padding;
}

struct metadata {
    /*empty*/
}

struct headers {
    ethernet_t           ethernet;
    ipv4_t               ipv4;
    udp_t               udp;
    dns_t               dns;

}
```

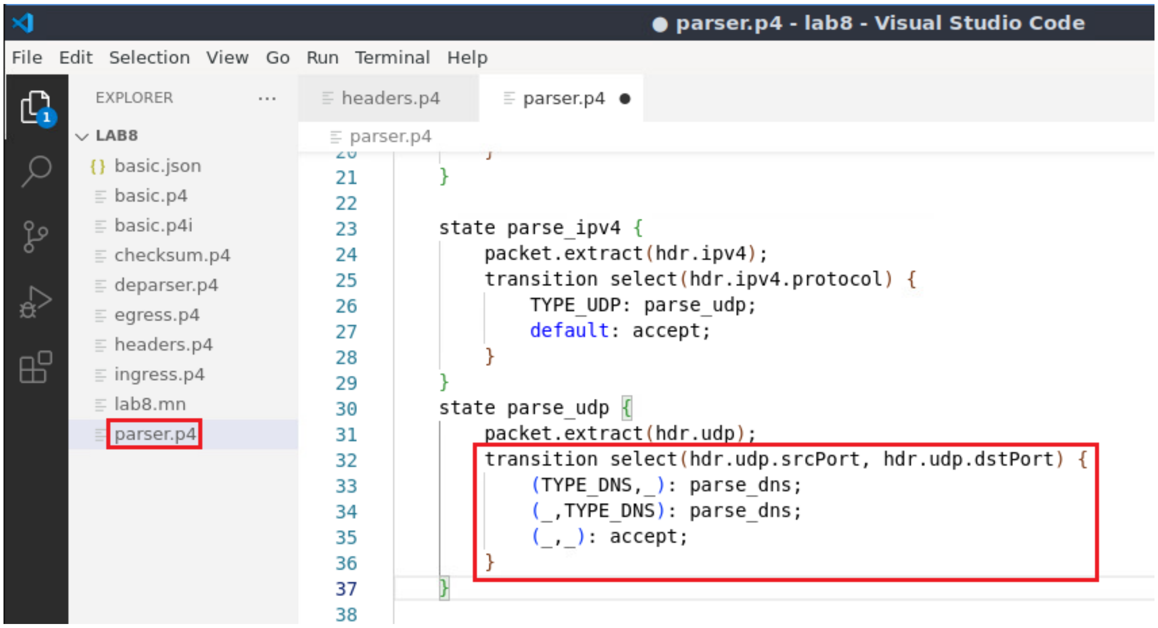
Figure 31. Adding the DNS header to the headers struct.

Step 4. Save the changes to the file by pressing `Ctrl + s`.

5.2 Modifying the parser file

Step 1. Use VScode to access the *parser.p4* file. In the *parser.p4* file, modify the state *parse_udp* by adding the following code to extract the DNS header if either the source port or the destination port of a UDP packet is *TYPE_DNS* (i.e., 53). The code must replace the `transition accept;` statement.

```
transition select(hdr.udp.srcPort, hdr.udp.dstPort){
    (TYPE_DNS,_): parse_dns;
    (_,TYPE_DNS): parse_dns;
    (_,_): accept;
}
```

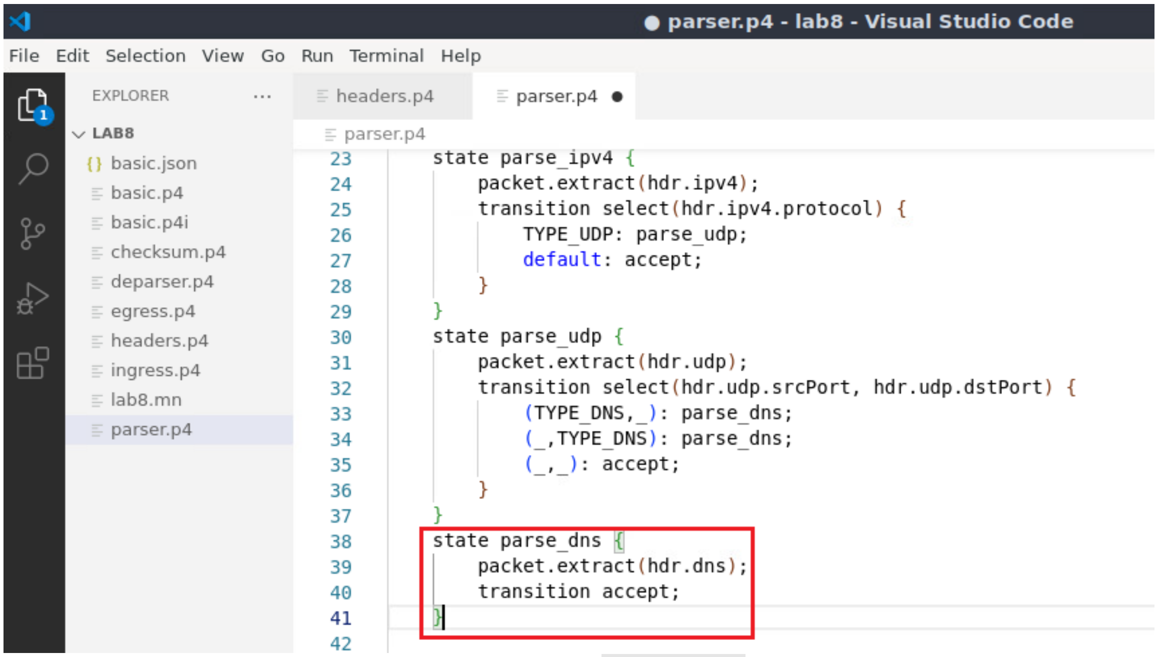


```
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            TYPE_UDP: parse_udp;
            default: accept;
        }
    }
    state parse_udp {
        packet.extract(hdr.udp);
        transition select(hdr.udp.srcPort, hdr.udp.dstPort) {
            (TYPE_DNS,_): parse_dns;
            (_,TYPE_DNS): parse_dns;
            (_,_): accept;
        }
    }
```

Figure 32. Adding the transition from UDP to DNS.

Step 2. Add the `parse_dns` state below the `parse_udp` state by typing the following code.

```
state parse_dns {
   packet.extract (hdr.dns);
   transition accept;
}
```



```
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            TYPE_UDP: parse_udp;
            default: accept;
        }
    }
    state parse_udp {
        packet.extract(hdr.udp);
        transition select(hdr.udp.srcPort, hdr.udp.dstPort) {
            (TYPE_DNS,_): parse_dns;
            (_,TYPE_DNS): parse_dns;
            (_,_): accept;
        }
    }
    state parse_dns {
        packet.extract(hdr.dns);
        transition accept;
    }
```

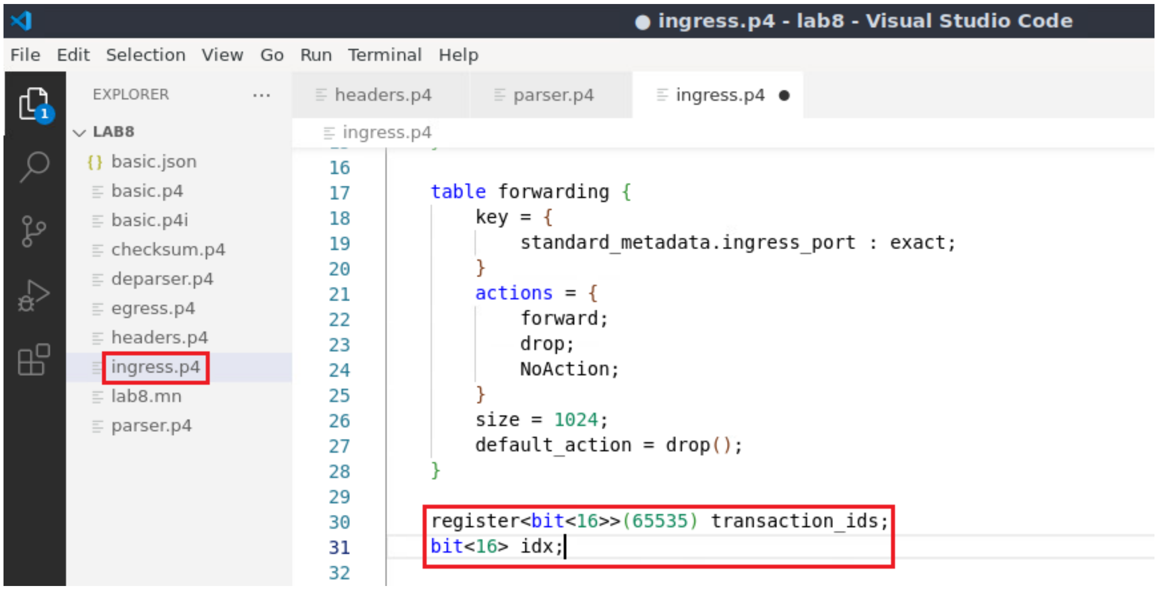
Figure 33. Adding the `parse_dns` state.

Step 3. Save the changes to the file by pressing `Ctrl + s`.

5.3 Modifying the ingress file

Step 1. In VScode, access the *parser.p4* file. In the *ingress.p4* file, define the register `transaction_ids` to store the DNS transaction IDs and a variable `idx` to store the hash of the 5-tuple.

```
register<bit<16>>(65535) transaction_ids;
bit<16> idx;
```



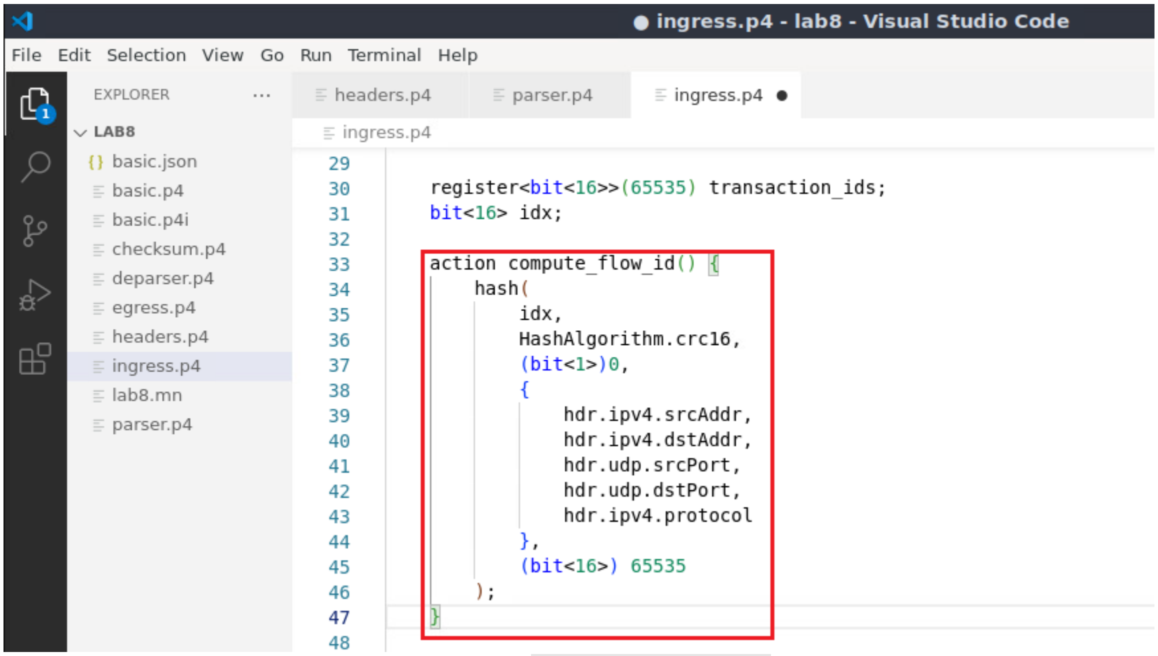
```

    table forwarding {
        key = {
            standard_metadata.ingress_port : exact;
        }
        actions = {
            forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    register<bit<16>>(65535) transaction_ids;
    bit<16> idx;

```

Figure 34. Defining register to store transaction Ids.

The code above defines a register named `transaction_ids`. The register contains 65535 cells. Each cell will be indexed by a flow ID and will store the transaction ID of that flow. The code also defines a 16-bit variable name `idx`. This variable will be used by the hashing actions.

Step 2. Define the action `compute_flow_id` by typing the following code.

```
action compute_flow_id(){
hash (
        idx,
        HashAlgorithm.crc16,
        (bit<1>)0,
        {
            hdr.ipv4.srcAddr,
            hdr.ipv4.dstAddr,
            hdr.udp.srcPort,
            hdr.udp.dstPort,
            hdr.ipv4.protocol
        },
        (bit<16>)65535
        );
}
```



```
    register<bit<16>>(65535) transaction_ids;
    bit<16> idx;

    action compute_flow_id() {
        hash(
            idx,
            HashAlgorithm.crc16,
            (bit<1>)0,
            {
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.udp.srcPort,
                hdr.udp.dstPort,
                hdr.ipv4.protocol
            },
            (bit<16>) 65535
        );
    }
```

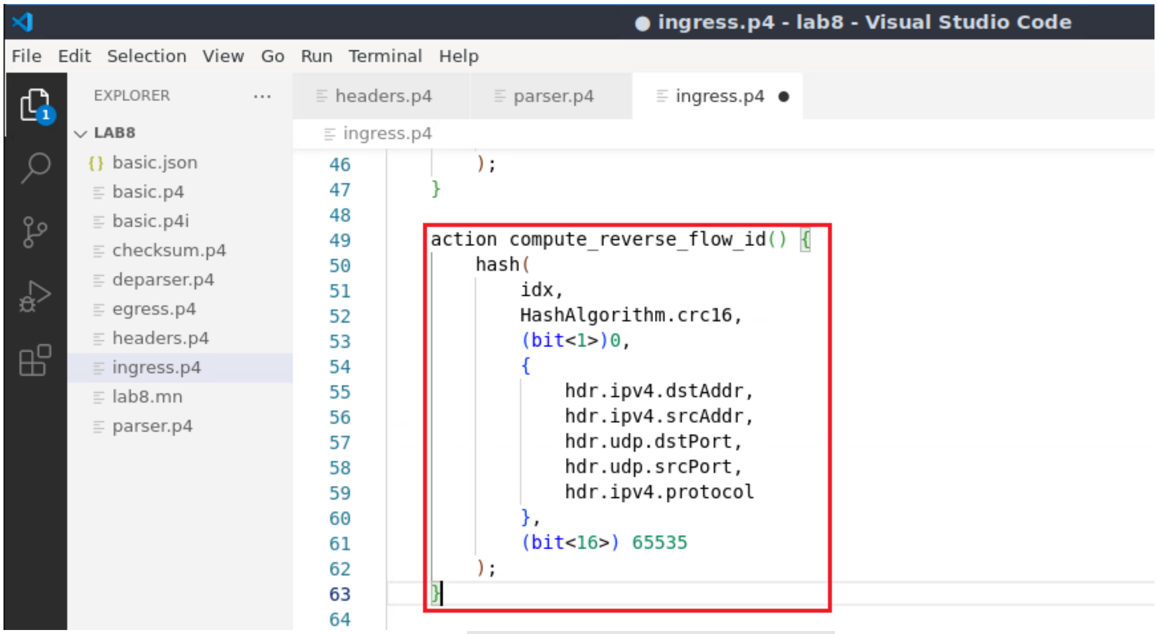
Figure 35. Defining `compute_flow_id` action.

The code in the figure above hashes flows based on their 5-tuple. The hash function produces a 16-bits output using the following parameters:

- `idx`: The variable used to store the output.
- `HashAlgorithm.crc16`: the hash algorithm.
- `(bit<1>)0`: the minimum (or base) value produced by the hash algorithm.
- `hdr.ipv4.srcAddr`, `hdr.ipv4.dstAddr`, `hdr.udp.srcPort`, `hdr.udp.dstPort`, and `hdr.ipv4.protocol`: the data to be hashed.
- `(bit<32>)65535`: the maximum value produced by the hash algorithm.

Step 3. Define the action `compute_reverse_flow_id` that hashes the 5-tuple of DNS packets.

```
action compute_reverse_flow_id(){
hash (
       idx,
       HashAlgorithm.crc16,
       (bit<1>)0,
       {
           hdr.ipv4.dstAddr,
           hdr.ipv4.srcAddr,
           hdr.udp.dstPort,
           hdr.udp.srcPort,
           hdr.ipv4.protocol
       },
       (bit<16>)65535
       );
}
```



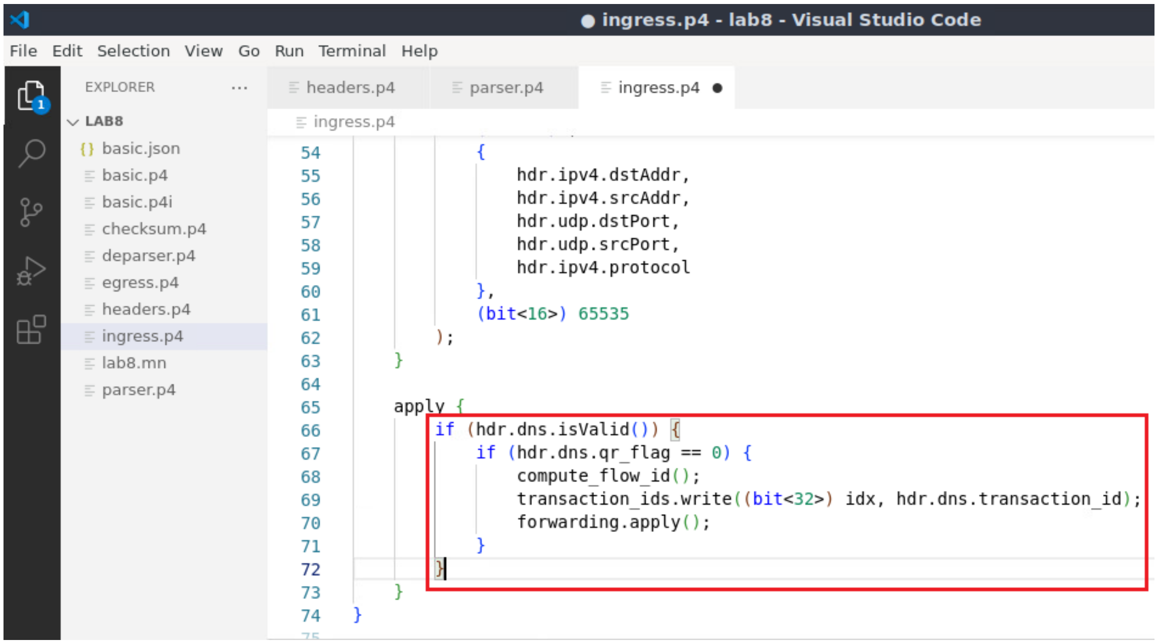
```
        );
    }

    action compute_reverse_flow_id() {
        hash(
            idx,
            HashAlgorithm.crc16,
            (bit<1>)0,
            {
                hdr.ipv4.dstAddr,
                hdr.ipv4.srcAddr,
                hdr.udp.dstPort,
                hdr.udp.srcPort,
                hdr.ipv4.protocol
            },
            (bit<16>) 65535
        );
    }
```

Figure 36. Defining `compute_reverse_flow_id` action.

Note that the order of hashing the source and destination IP addresses and ports is reversed in `compute_reverse_flow_id` compared to `compute_flow_id` so that the DNS requests and responses will be processed as a single flow and their hash will map to the same register cell.

Step 4. Override the *apply* block as follows.

```
apply {
    if (hdr.dns.isValid()){
         if (hdr.dns.qr_flag == 0) {
             compute_flow_id();
             transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
             forwarding.apply();
         }
    }
}
```



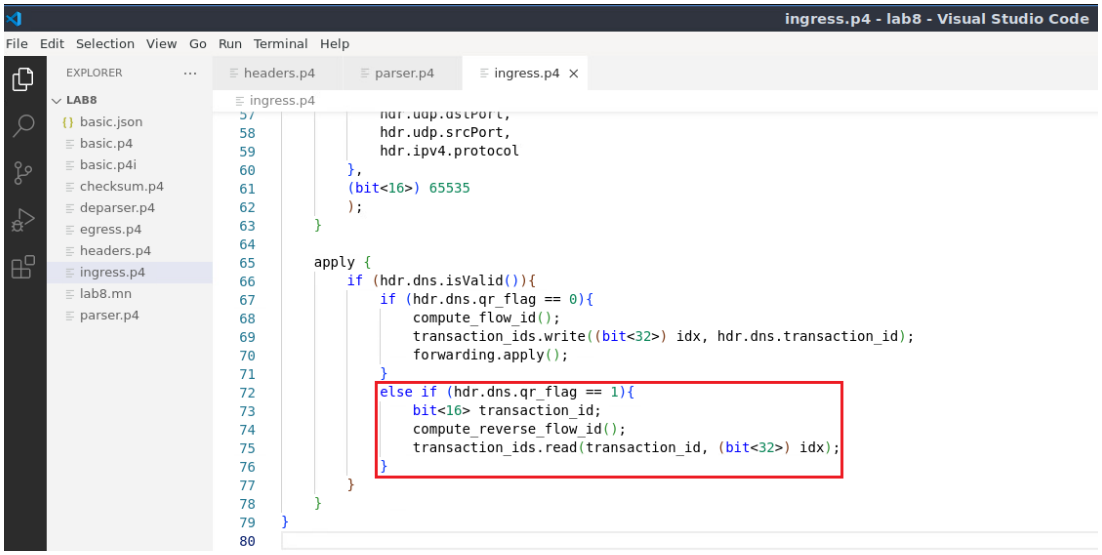
```
            {
                hdr.ipv4.dstAddr,
                hdr.ipv4.srcAddr,
                hdr.udp.dstPort,
                hdr.udp.srcPort,
                hdr.ipv4.protocol
            },
            (bit<16>) 65535
        );
    }

    apply {
        if (hdr.dns.isValid()) {
            if (hdr.dns.qr_flag == 0) {
                compute_flow_id();
                transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
                forwarding.apply();
            }
        }
    }
}
```

Figure 37. Implementing the apply block.

In the code above, `if(hdr.dns.isValid())` checks the validity of the DNS header. For DNS packets, `if(hdr.dns.qr_flag == 0)` checks if the current packet is a DNS request packet. If yes, the hash of the flow is calculated using `compute_flow_id` action. The transaction ID of the DNS packet is then stored inside the `transaction_ids` register. The hash of the flow is used to index the `transaction_ids` register.

Step 5. Add the following code to the *apply* block to calculate the hash of the DNS responses and retrieve the corresponding transaction id.

```
else if (hdr.dns.qr_flag == 1){
    bit<16> transaction_id;
    compute_reverse_flow_id();
    transaction_ids.read(transaction_id, (bit<32>) idx);
}
```



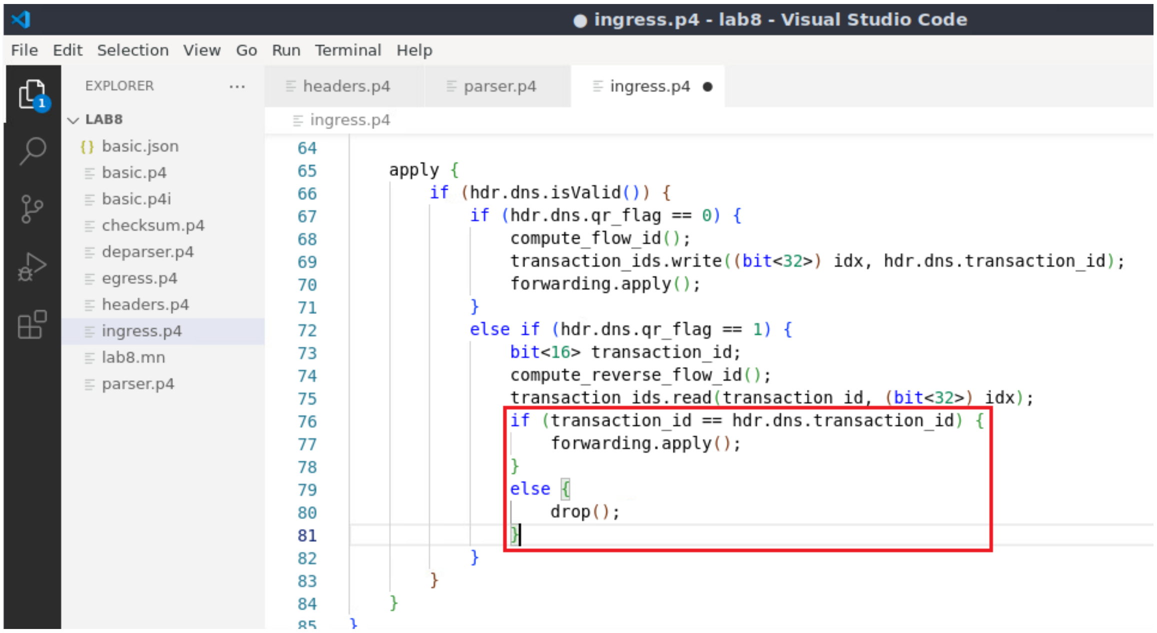
```
                hdr.udp.dstPort,
                hdr.udp.srcPort,
                hdr.ipv4.protocol
            },
            (bit<16>) 65535
        );
    }

    apply {
        if (hdr.dns.isValid()){
            if (hdr.dns.qr_flag == 0){
                compute_flow_id();
                transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
                forwarding.apply();
            }
            else if (hdr.dns.qr_flag == 1){
                bit<16> transaction_id;
                compute_reverse_flow_id();
                transaction_ids.read(transaction_id, (bit<32>) idx);
            }
        }
    }
}
```

Figure 38. Implementing the apply block.

In the code above, `else if (hdr.dns.qr_flag == 1)` checks if the DNS packet is a reply packet. The `compute_reverse_flow_id` action calculates the hash of DNS reply packet. The hash value is used to index the `transaction_ids` register and retrieve the corresponding transaction id. The retrieved ID is stored inside the `transaction_id` variable.

Step 6. Add the following code to forward the packet if the retrieved transaction ID (i.e., `transaction_id`) is the same as the transaction ID extracted from the current packet (i.e., `hdr.dns.transaction_id`). The packet will be dropped if the two ids do not match.

```
if (transaction_id == hdr.dns.transaction_id){
    forwarding.apply();
}
else {
    drop();
}
```

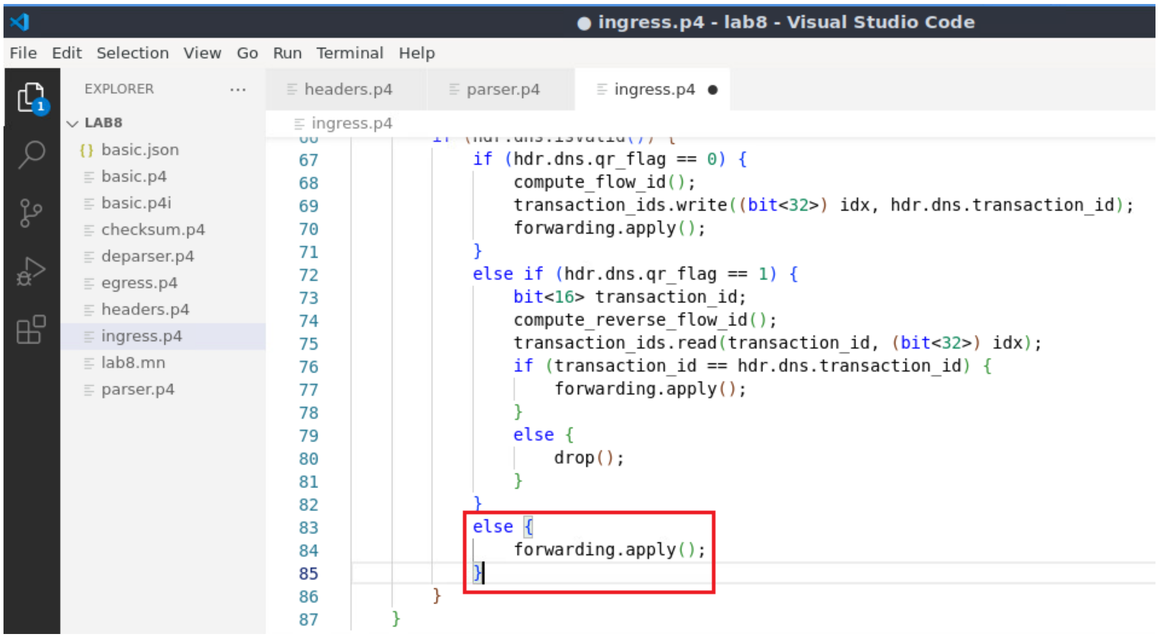


```

    apply {
        if (hdr.dns.isValid()) {
            if (hdr.dns.qr_flag == 0) {
                compute_flow_id();
                transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
                forwarding.apply();
            }
            else if (hdr.dns.qr_flag == 1) {
                bit<16> transaction_id;
                compute_reverse_flow_id();
                transaction_ids.read(transaction_id, (bit<32>) idx);
                if (transaction_id == hdr.dns.transaction_id) {
                    forwarding.apply();
                }
                else {
                    drop();
                }
            }
        }
    }
}
```

Figure 39. Implementing the apply block.

Step 7. Add the following code to forward non-DNS packets.

```
else {
    forwarding.apply();
}
```



```
            if (hdr.dns.isValid()) {
                if (hdr.dns.qr_flag == 0) {
                    compute_flow_id();
                    transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
                    forwarding.apply();
                }
                else if (hdr.dns.qr_flag == 1) {
                    bit<16> transaction_id;
                    compute_reverse_flow_id();
                    transaction_ids.read(transaction_id, (bit<32>) idx);
                    if (transaction_id == hdr.dns.transaction_id) {
                        forwarding.apply();
                    }
                    else {
                        drop();
                    }
                }
                else {
                    forwarding.apply();
                }
            }
        }
```

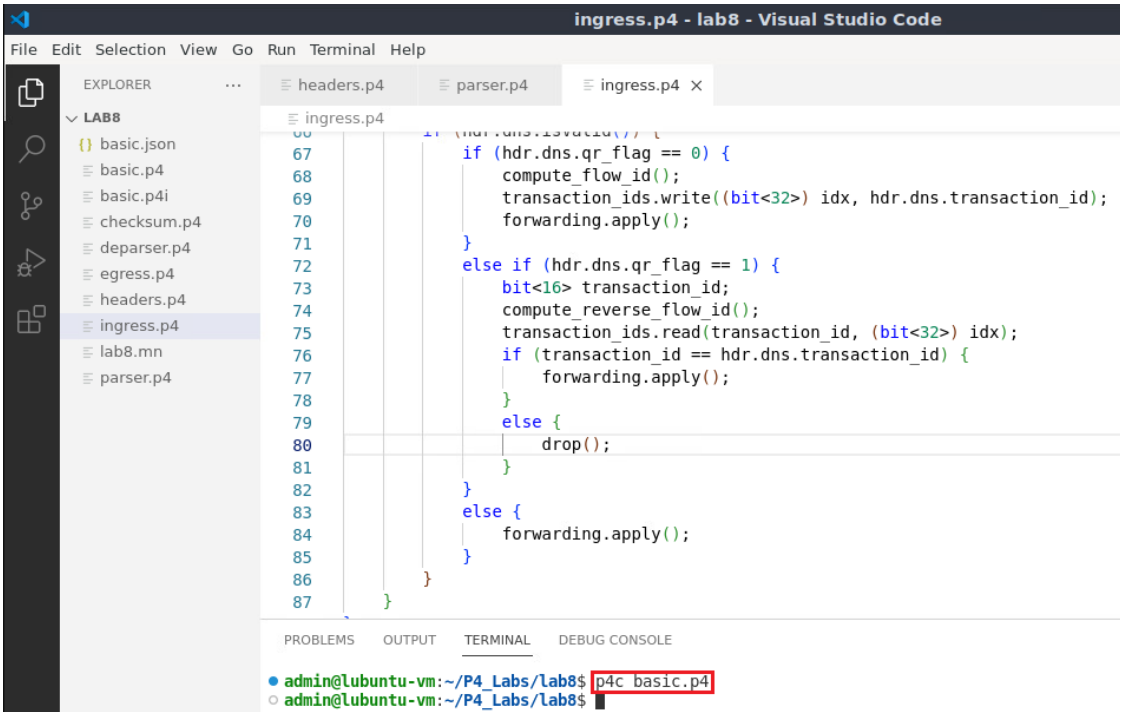
Figure 40. Implementing the apply block.

Step 8. Save the changes to the file by pressing `Ctrl + s`.

5.4 Loading the modified P4 program

Step 1. To compile the P4 program, issue the following command in the terminal panel in VS Code.

```
p4c basic.p4
```



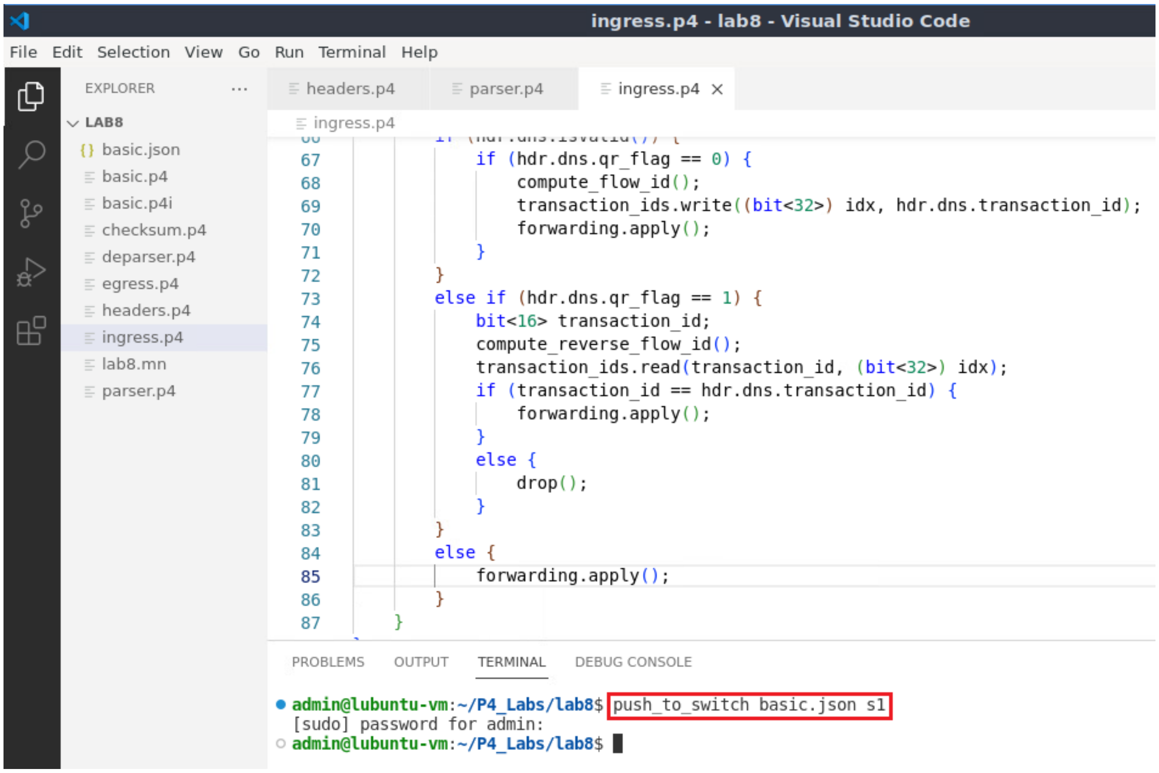
```
            if (hdr.dns.isValid()) {
                if (hdr.dns.qr_flag == 0) {
                    compute_flow_id();
                    transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
                    forwarding.apply();
                }
                else if (hdr.dns.qr_flag == 1) {
                    bit<16> transaction_id;
                    compute_reverse_flow_id();
                    transaction_ids.read(transaction_id, (bit<32>) idx);
                    if (transaction_id == hdr.dns.transaction_id) {
                        forwarding.apply();
                    }
                    else {
                        drop();
                    }
                }
                else {
                    forwarding.apply();
                }
            }
        }
```

```
admin@lubuntu-vm:~/P4_Labs/lab8$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab8$
```

Figure 41. Compiling the P4 program using the VS Code terminal.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



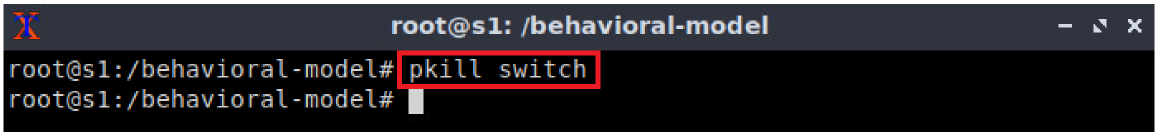
```
            if (hdr.dns.qr_flag == 0) {
                compute_flow_id();
                transaction_ids.write((bit<32>) idx, hdr.dns.transaction_id);
                forwarding.apply();
            }
        }
        else if (hdr.dns.qr_flag == 1) {
            bit<16> transaction_id;
            compute_reverse_flow_id();
            transaction_ids.read(transaction_id, (bit<32>) idx);
            if (transaction_id == hdr.dns.transaction_id) {
                forwarding.apply();
            }
            else {
                drop();
            }
        }
        else {
            forwarding.apply();
        }
    }
```

```
admin@lubuntu-vm:~/P4_Labs/lab8$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab8$
```

Figure 42. Downloading the compiled program to switch s1.

Step 3. Type the command below to kill the simple switch daemon, so that the new P4 program can be loaded.

```
pkill switch
```

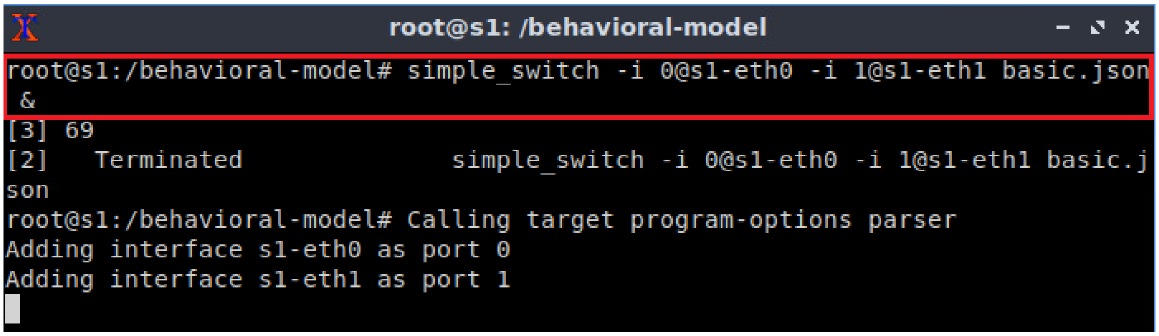


```
root@s1:/behavioral-model# pkill switch
root@s1:/behavioral-model#
```

Figure 43. Killing the simple switch daemon.

Step 4. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

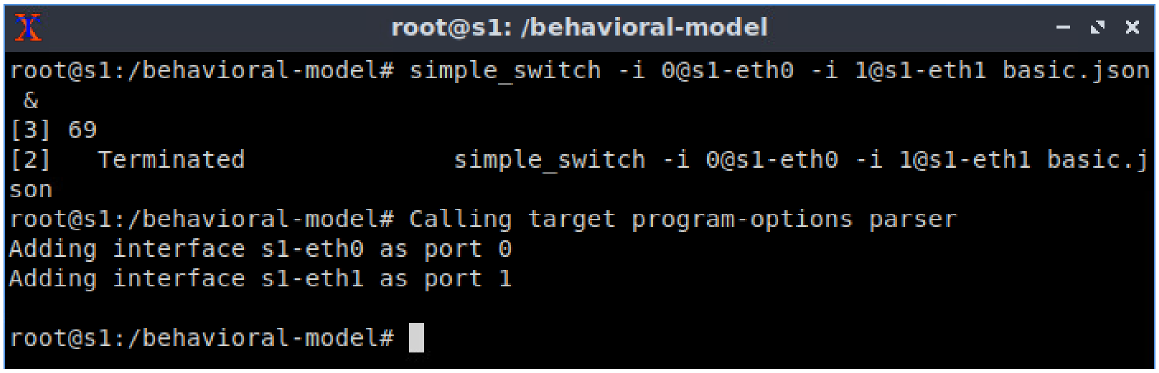
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[3] 69
[2]   Terminated              simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.j
son
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 44. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 5. In switch s1 terminal, press *Enter* to return the CLI.

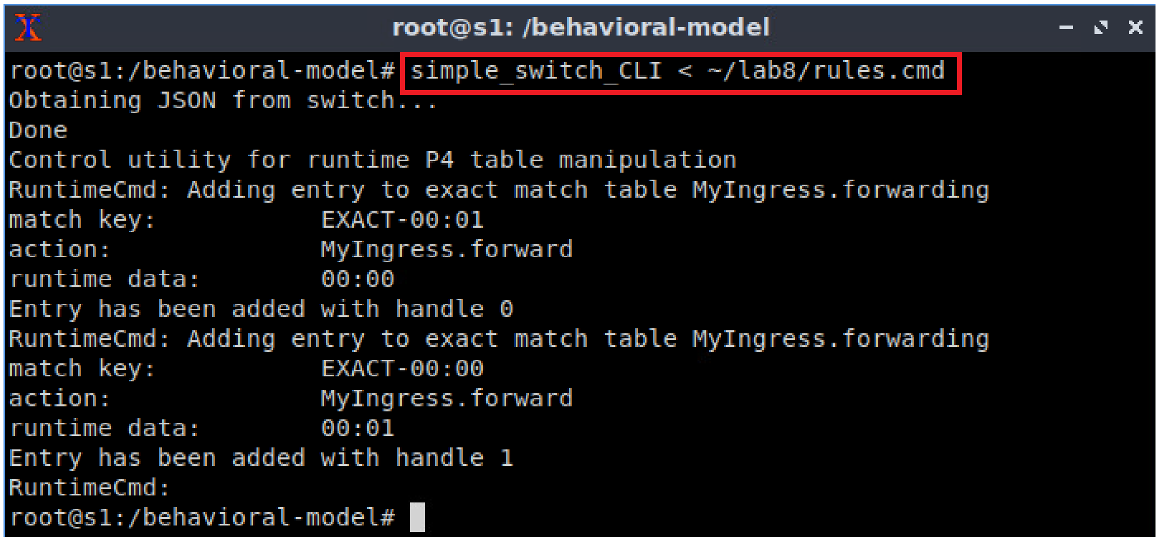


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[3] 69
[2]   Terminated              simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.j
son
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 45. Returning to switch s1 CLI.

Step 6. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab8/rules.cmd
```



```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab8/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model# 
```

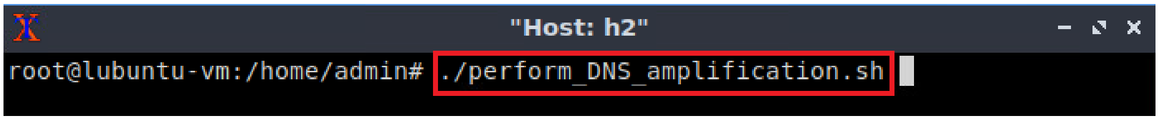
Figure 46. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

5.5 Performing the DNS amplification attack

Step 1. On h2 terminal, type the command below to perform DNS amplification attack.

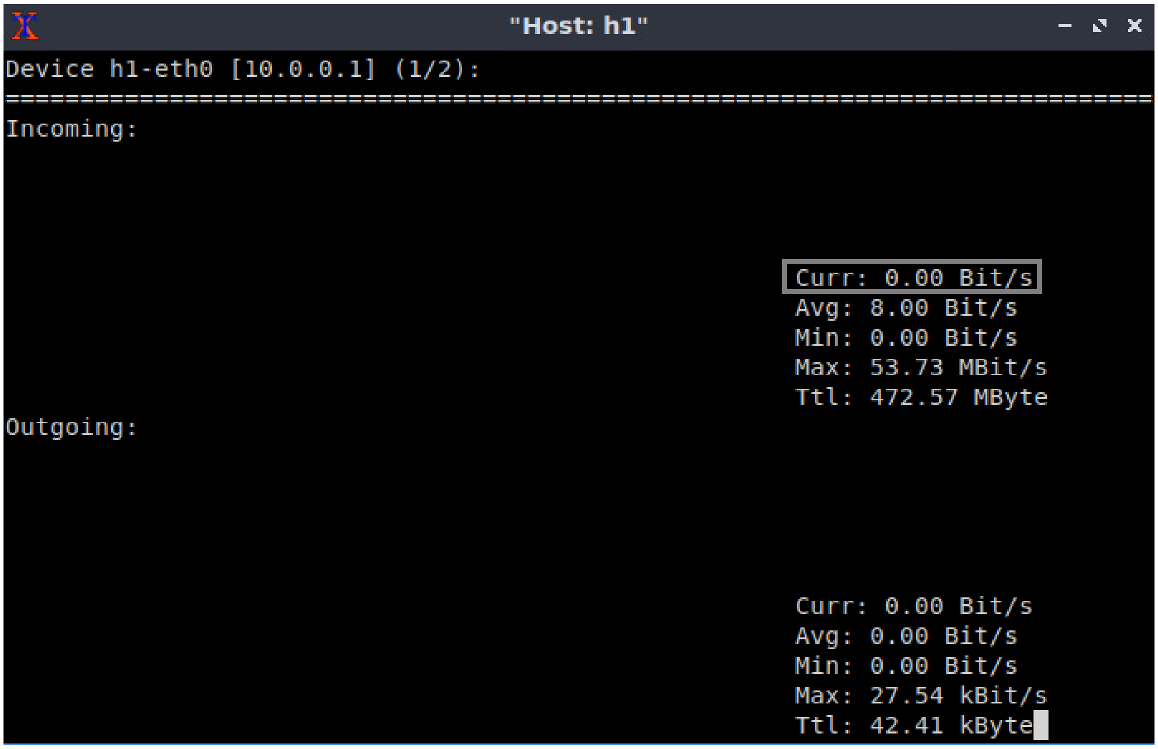
```
./perform_DNS_amplification.sh
```



```
root@lubuntu-vm:/home/admin# ./perform_DNS_amplification.sh
```

Figure 47. Performing the DNS amplification attack.

Step 2. Inspect the network usage at h1. Press `Ctrl + c` to exit *nload* after inspecting the network usage.



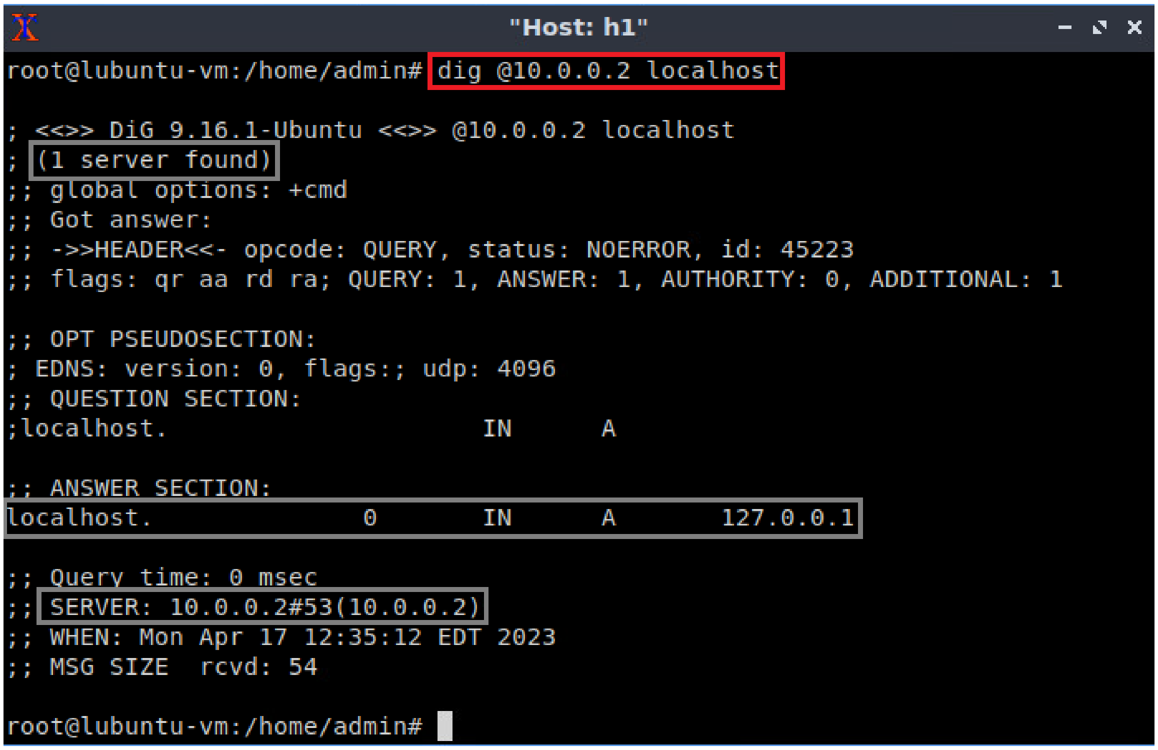
```
Device h1-eth0 [10.0.0.1] (1/2):
==============================================================================
Incoming:



                                                     Curr: 0.00 Bit/s
                                                     Avg: 8.00 Bit/s
                                                     Min: 0.00 Bit/s
                                                     Max: 53.73 MBit/s
                                                     Ttl: 472.57 MByte
Outgoing:



                                                     Curr: 0.00 Bit/s
                                                     Avg: 0.00 Bit/s
                                                     Min: 0.00 Bit/s
                                                     Max: 27.54 kBit/s
                                                     Ttl: 42.41 kByte
```

Figure 48. Inspecting resource usage at h1.

The figure above shows that there is no network usage at h1. The switch was successful in dropping all the packets.

Step 3. On h1 terminal, type the command below to validate that legitimate DNS queries will be forwarded by the switch.

```
dig @10.0.0.2 localhost
```



```
root@lubuntu-vm:/home/admin# dig @10.0.0.2 localhost

; <<>> DiG 9.16.1-Ubuntu <<>> @10.0.0.2 localhost
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45223
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;localhost.                     IN      A

;; ANSWER SECTION:
localhost.              0       IN      A       127.0.0.1

;; Query time: 0 msec
;; SERVER: 10.0.0.2#53(10.0.0.2)
;; WHEN: Mon Apr 17 12:35:12 EDT 2023
;; MSG SIZE  rcvd: 54

root@lubuntu-vm:/home/admin# 
```

Figure 49. Issuing DNS query.

The figure above shows that h1 received the DNS response.

This concludes lab 8. Stop the emulation and then exit out of MiniEdit.

References

1. Amazon, “What is DNS?” [Online]. Available: https://tinyurl.com/ynb9esn6
2. NOCTION, “DNS Amplification Attacks Detection with NetFlow or sFlow.” [Online]. Available: https://tinyurl.com/yh9v6nba
3. PURPLESEC, “How To Prevent A Domain Name Server (DNS) Amplification attack.” [Online]. Available: https://tinyurl.com/5evebess
4. Charles M. Kozierok, “The TCP/IP Guide.” [Online]. Available: https://tinyurl.com/83r4bc5m

UNIVERSITY OF
SOUTH CAROLINA

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 9: Identifying Heavy Hitters using Count-min Sketches (CMS)

Document Version: 04-18-2023

Contents

Overview

This lab introduces the concept of heavy hitters and demonstrates how to implement the Count-Min Sketch data structure in a P4 program to detect heavy hitters. Heavy hitters refer to network traffic flows with significantly higher data rates or packet counts than average, often dominating network resources and potentially causing congestion or service degradation. Count-Min Sketch is a probabilistic data structure that estimates the frequency of elements in a stream of data. The user will implement the Count-Min Sketch data structure using P4 registers to detect heavy hitters and then drop them.

Objectives

By the end of this lab, students should be able to:

1. Define heavy hitters.
2. Understand the workflow of the Count-Min Sketch data structure.
3. Leverage Count-Min Sketch to detect heavy flows in P4.

Lab settings

Table 1 contains the credentials of the virtual machine used for this lab.

Table 1**.** Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Implementing Count-Min Sketch data structure in P4.
4. Section 4: Testing the P4 program.

1 Introduction

Heavy hitter detection is an essential task in the analysis of large-scale data streams, aiming to identify items with a frequency exceeding a specified threshold. These items, known as heavy hitters, can reveal crucial insights in various applications, such as network traffic analysis, clickstream analysis, and natural language processing. Count-Min Sketch (CM Sketch) is a probabilistic data structure that provides an efficient solution for estimating item frequencies in high-dimensional and high-velocity data streams with bounded error[1].

CM Sketch utilizes hashing and a compact 2D array to store and track frequency information[2]. It allows for fast updates and queries while significantly reducing memory requirements compared to exact counting methods. As a result, it is particularly suitable for heavy hitter detection in situations where data streams are too large to fit in memory, and a small degree of error is acceptable.

To detect heavy hitters using CM Sketch, the data stream is processed incrementally, updating the sketch with each incoming packet[3]. When querying for potential heavy hitters, the sketch returns estimated frequencies, which can be compared to the predefined threshold to determine if an item qualifies as a heavy hitter.

While the nature of the CM Sketch introduces some estimation errors, it offers a tunable trade-off between accuracy and memory usage by adjusting its parameters[4]. This trade-off is crucial for applications where space efficiency is of paramount importance. Despite its inherent limitations, the CM Sketch remains a popular choice for heavy hitter detection due to its effectiveness, simplicity, and versatility in handling massive data streams.

Consider Figure 1. The CM Sketch data structure is constructed using *d* register arrays that contain *w* cells each. Thus, the data structure can be seen as a matrix of size $w * d$. The CM Sketch uses *d* pairwise-independent hash functions *hi, ..., hd* that are applied to the 5-tuple fields in the packet headers. The results of the hash functions correspond to the indices of the counts in the *d* register arrays; these counts are incremented by one. Calculating the minimum between these counts gives an approximation of the packet counts per flow; note that this is an approximation and not the exact count because collisions might occur, which leads to overestimating the counts.

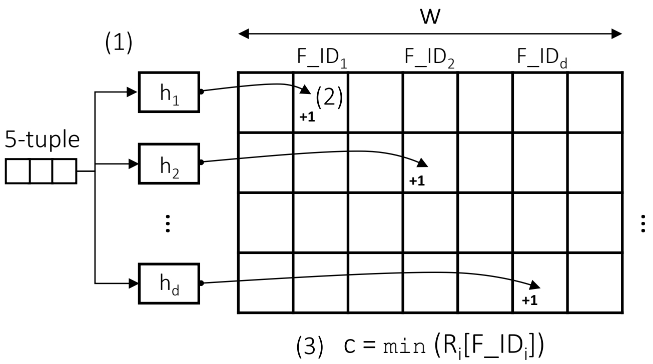


Figure 1. Workflow of CM Sketch.

1.1 Lab scenario

In this lab, the P4 switch will blacklist heavy flows. The P4 program will estimate the number of packets per flow, where a flow is characterized by its 5-tuple (source IP, destination IP, source port, destination port, and protocol). The CM Sketch data structure will track the number of packets for each flow. When the number of packets exceeds a predefined threshold, the data plane will classify the flow as a heavy hitter and blacklist it. All subsequent packets of a blacklisted flow are dropped.

Consider Figure 2. The topology consists of an HTTP server, an iPerf3 server, an HTTP client, and an iPerf3 client. The HTTP client performs GET requests from the HTTP server. The requests will be successful as the number of packets per request will be less than the heavy hitter detection threshold. The iPerf3 client and the iPerf3 server will be transferring a large file. The number of packets of the iPerf3 flow will exceed the heavy detection threshold, causing the flow to be blacklisted.

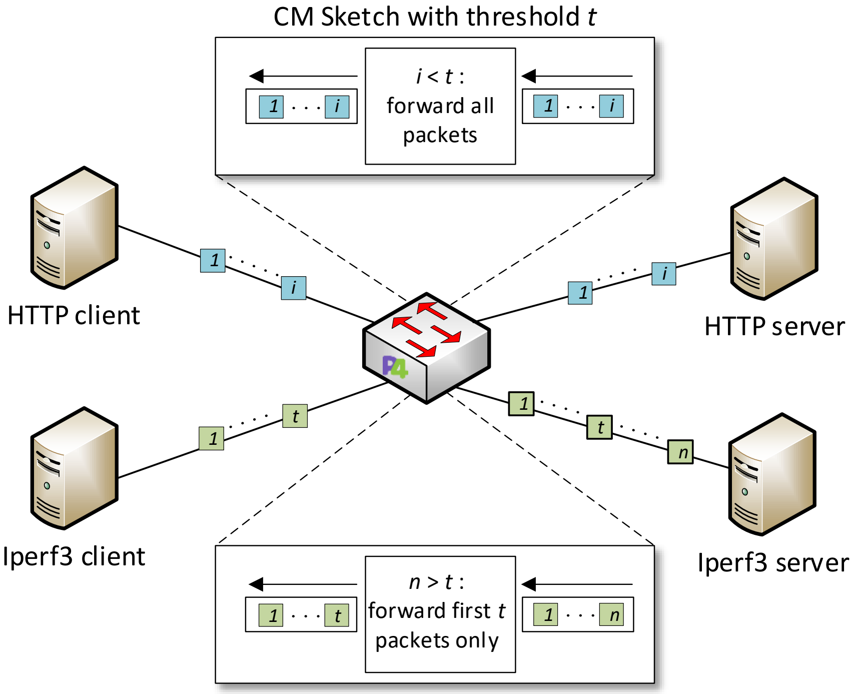


Figure 2. Lab scenario.

2 Lab topology

Let’s get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

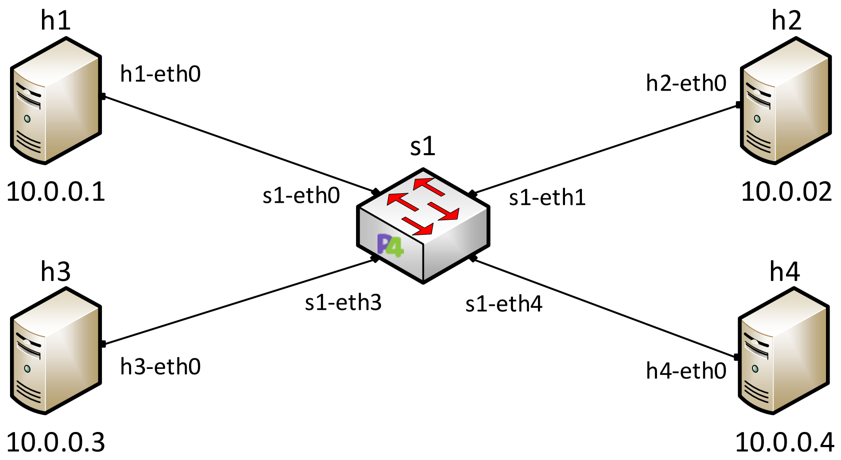


Figure 3. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

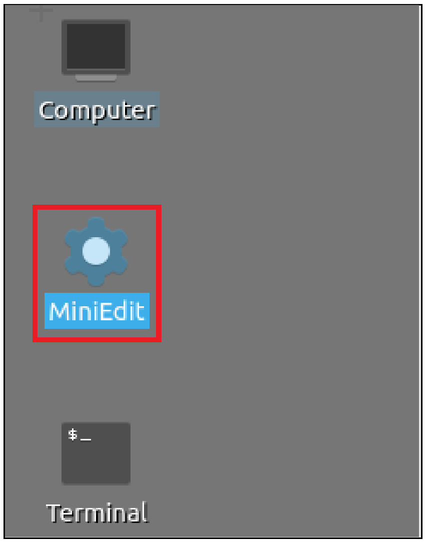


Figure 4. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab9*, select the file *lab9.mn,* and click on *Open*.

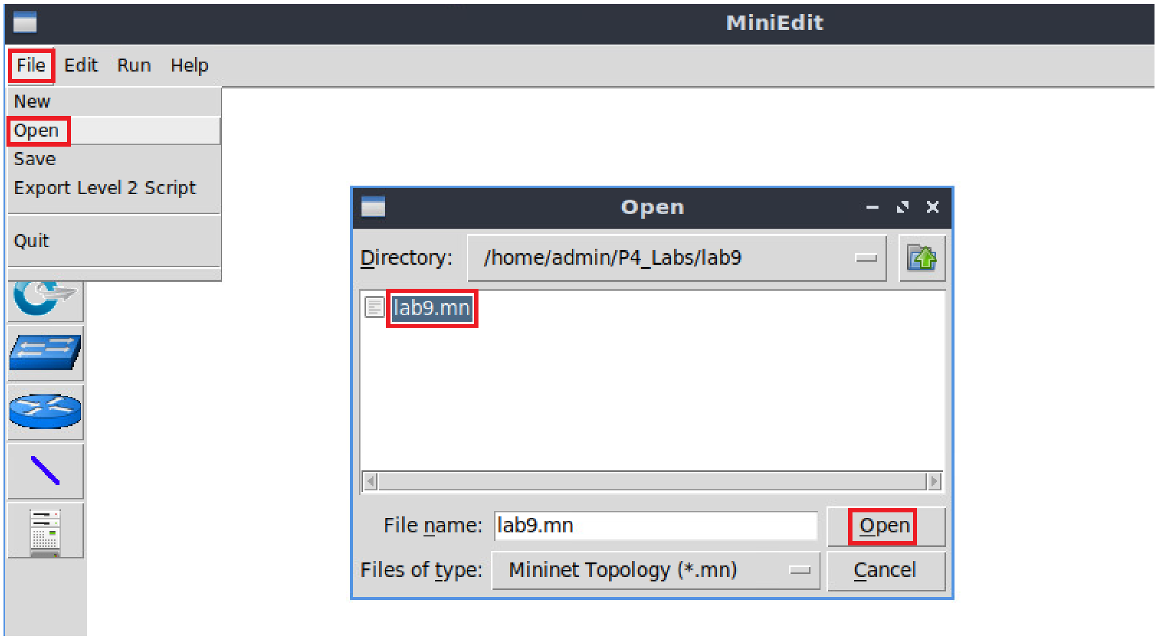


Figure 5. Opening a topology in MiniEdit.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

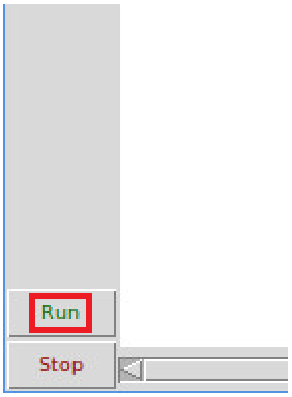


Figure 6. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

Step 1. Hold the right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

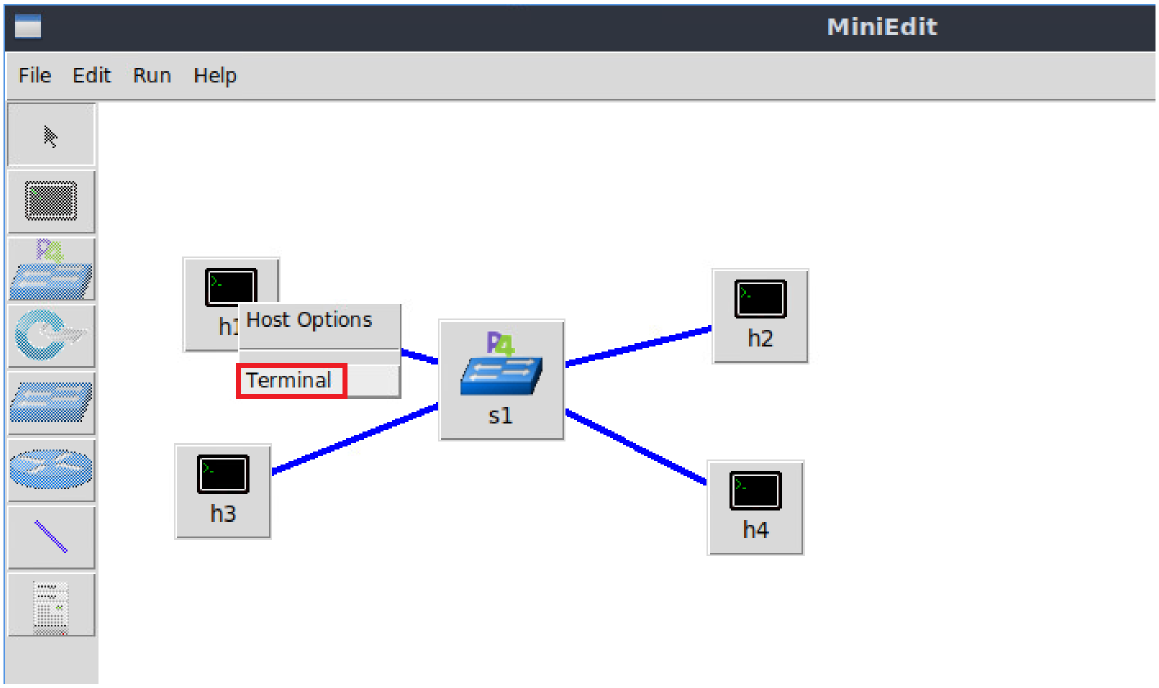
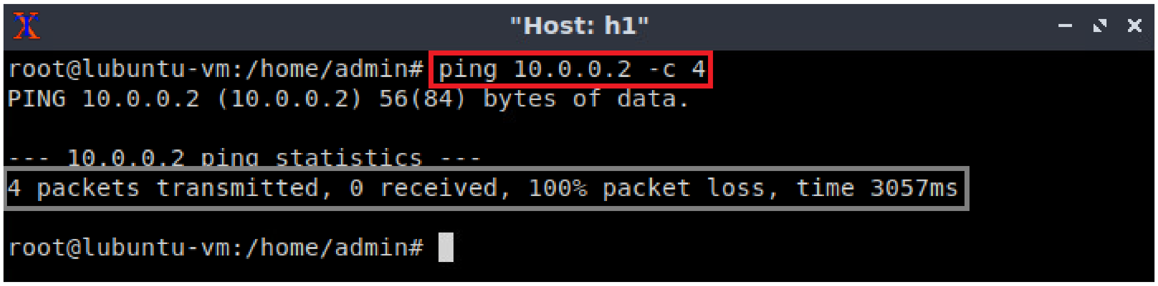


Figure 7. Opening a terminal on host h1.

Step 2. Test the connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```



"Host: h1"

```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3057ms

root@lubuntu-vm:/home/admin# 
```

Figure 8. Performing a connectivity test between host h1 and host h2.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded into the switch. Note that there will be no connection between any two hosts in the topology before loading the P4 program.

3 Implement Count-Min Sketch data structure in P4

In this section, the user will implement the CM Sketch data structure. The data structure will utilize three different hash functions.

3.1 Loading the programming environment

Step 1. Launch a Linux terminal by double-clicking on the Linux terminal icon located on the desktop.

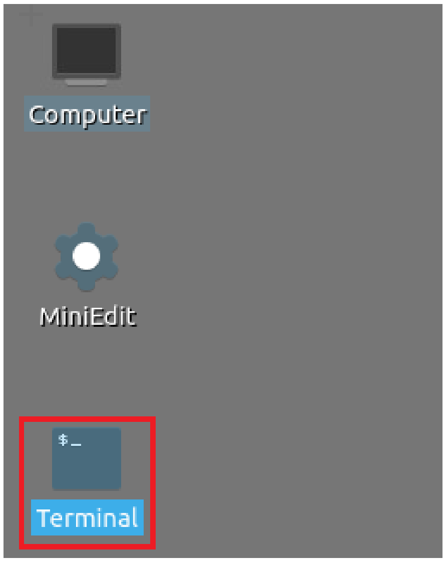
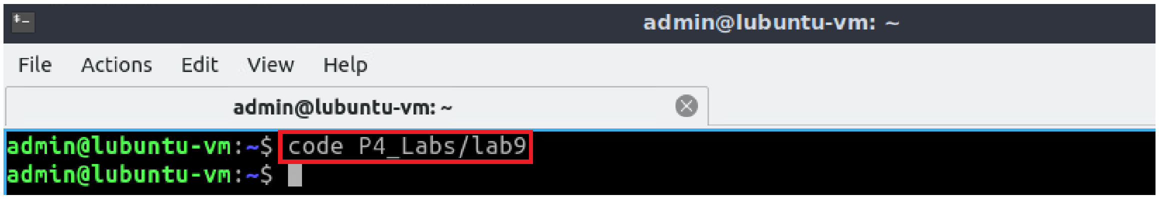


Figure 9. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the VS Code and opens the directory where the P4 program for this lab is located.

```
code P4_Labs/lab9
```



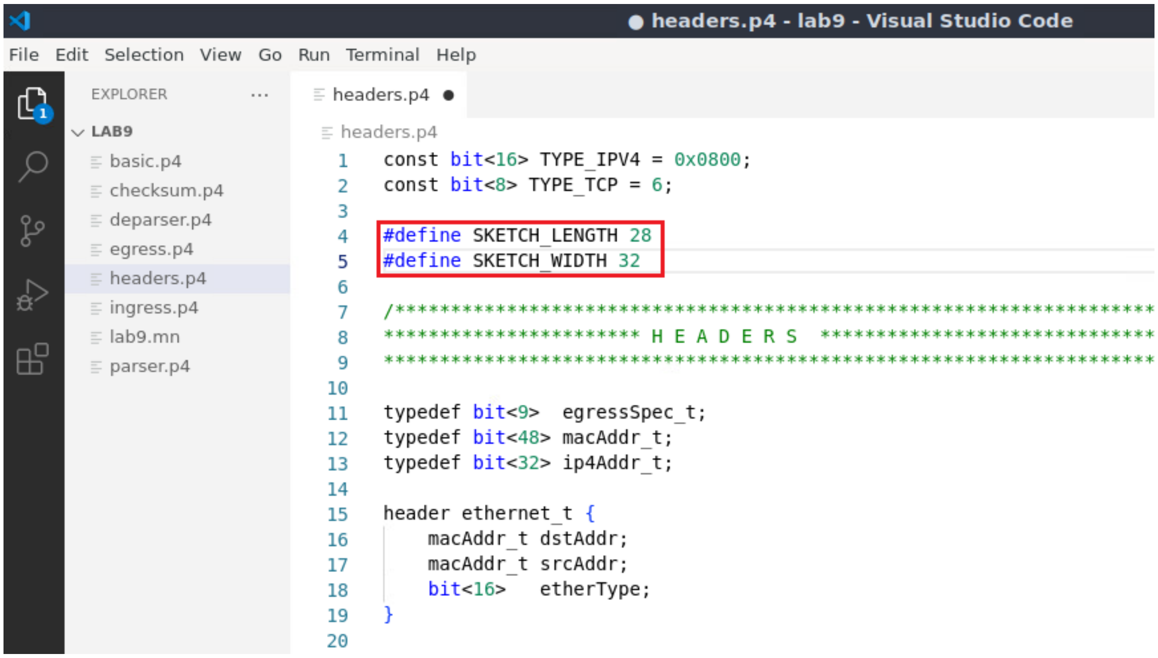
```
admin@lubuntu-vm:~$ code P4_Labs/lab9
admin@lubuntu-vm:~$ 
```

Figure 10. Launching the editor and opening the lab9 directory.

3.2 Modify the header file

Step 1. In the *header.p4* file, add the following two definitions.

```
#define SKETCH_LENGTH 28
#define SKETCH_WIDTH 32
```



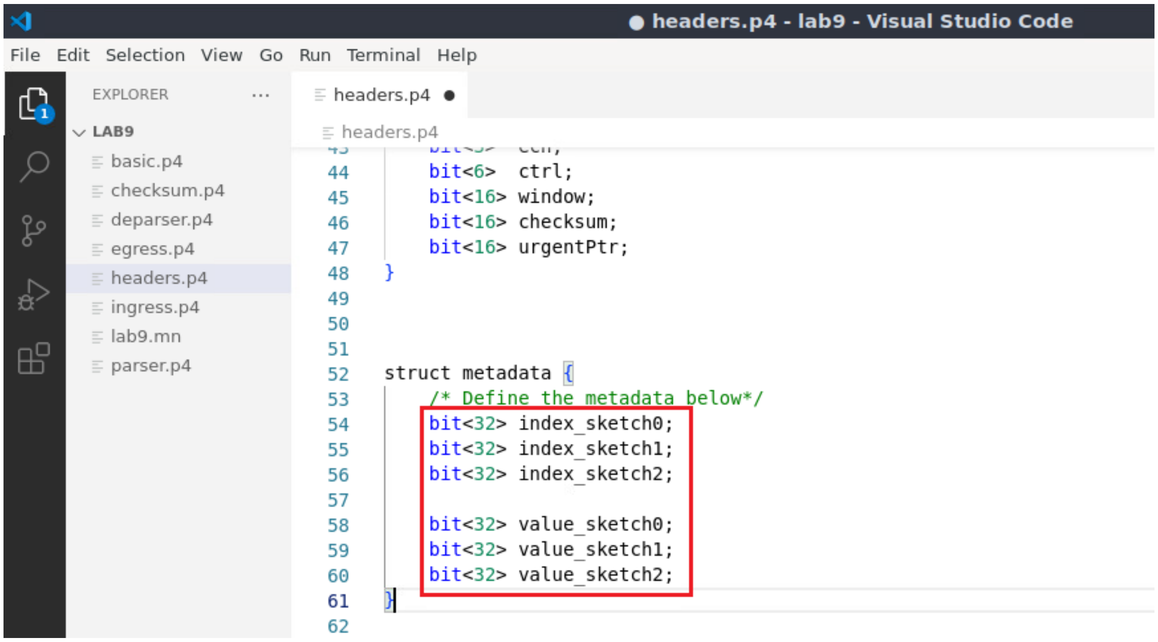
```
const bit<16> TYPE_IPV4 = 0x0800;
const bit<8> TYPE_TCP = 6;

#define SKETCH_LENGTH 28
#define SKETCH_WIDTH 32

/*************************************************************************
*********************** 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;
}
```

Figure 11. Defining the length and width of the sketches.

In the code above, `SKETCH_LENGTH` represents the number of cells in a sketch and `SKETCH_WIDTH` represents the size of counters inside a sketch.

Step 2. Define the metadata by adding the following code inside the metadata struct.

```
bit<32> index_sketch0;
bit<32> index_sketch1;
bit<32> index_sketch2;

bit<32> value_sketch0;
bit<32> value_sketch1;
bit<32> value_sketch2;
```



```
    bit<6>  ctrl;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}



struct metadata {
    /* Define the metadata below*/
    bit<32> index_sketch0;
    bit<32> index_sketch1;
    bit<32> index_sketch2;

    bit<32> value_sketch0;
    bit<32> value_sketch1;
    bit<32> value_sketch2;
}
```

Figure 12. Defining the metadata.

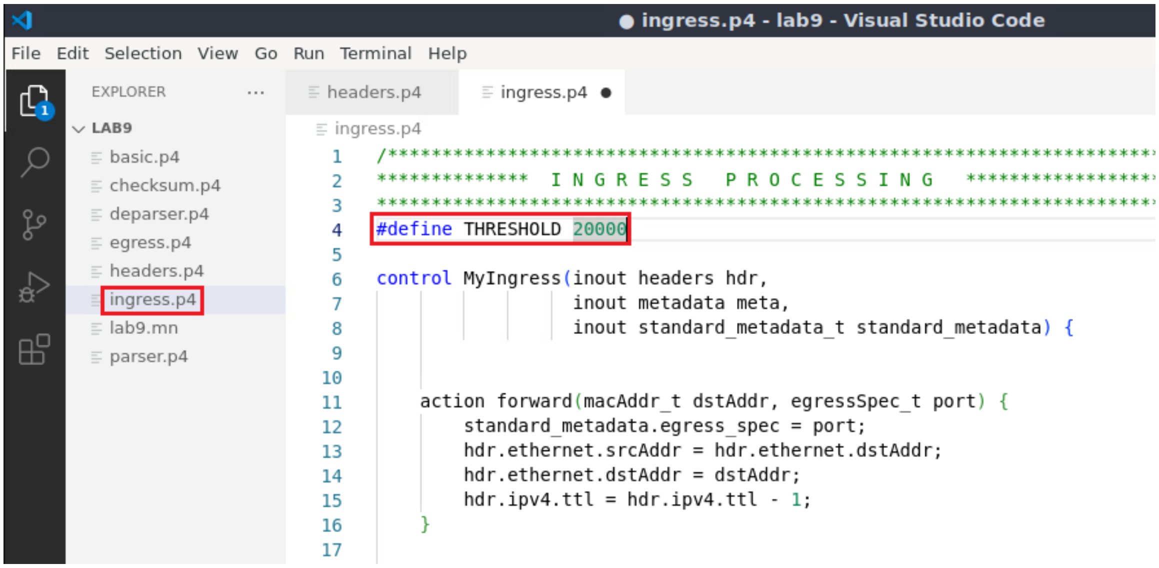
The code above defines an index variable and a value variable for each of the three sketches (sketch0, sketch1, and sketch2) that will build the CM Sketch.

Step 3. Save the changes to the file by pressing `Ctrl + s`.

3.3 Modify the ingress file

Step 1. Use VScode Explorer to access the *ingress.p4* file. In the *ingress.p4* file, add the definition below. `THRESHOLD` defines the maximum number of packets a flow can have before being considered a heavy flow. The threshold is set to 20000 packets.

```
#define THRESHOLD 20000
```

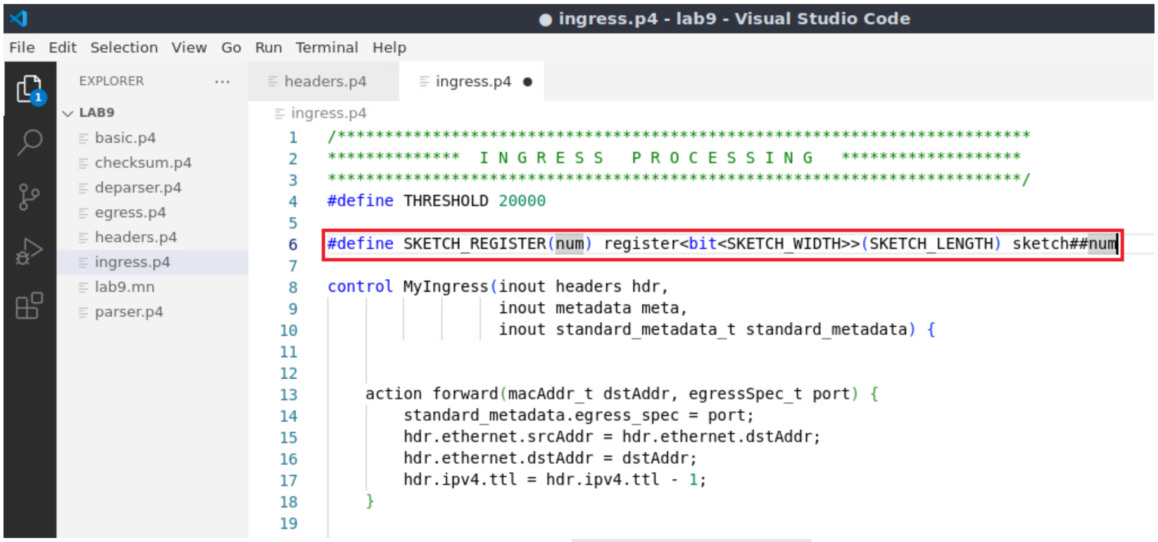


```
/************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *****************
*************************************************************************
#define THRESHOLD 20000

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {


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

```

Figure 13. Defining the detection threshold.

Step 2. In the *ingress.p4* file, add the following code.

```
#define SKETCH_REGISTER(num) register<bit<SKETCH_WIDTH>>(SKETCH_LENGTH)
sketch##num
```



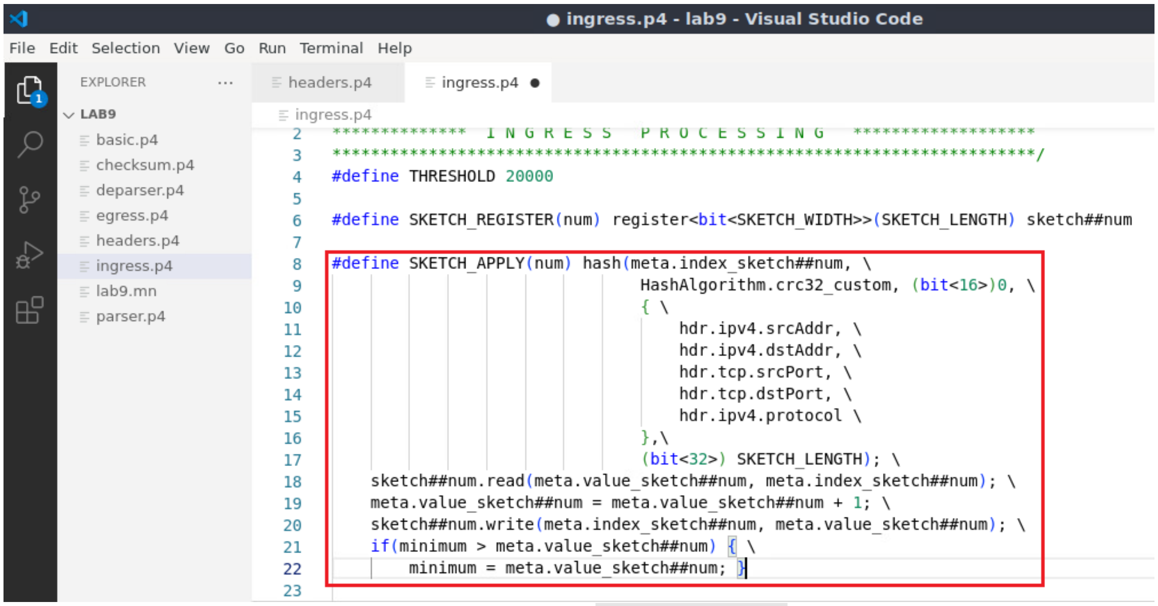
```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/
#define THRESHOLD 20000

#define SKETCH_REGISTER(num) register<bit<SKETCH_WIDTH>>(SKETCH_LENGTH) sketch##num

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {


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

```

Figure 14. Defining `SKETCH_REGISTER`.

In the definition above, the `##` symbols indicate that the name of the register will depend on the `num` provided for `SKETCH_REGISTER`. When the user initiates a `SKETCH_REGISTER` and provides a number `num` as an argument (i.e., `SKETCH_REGISTER(num)`), a register with name `sketchnum` will be initiated. For example, if the user defines SKETCH_REGISTER(0) then the following register will be created: `register<bit<32>>(28) sketch0`. The register will have `SKETCH_LENGTH` cells (i.e., 28 cells) where each cell stores `SKETCH_WIDTH` bits (i.e., 32 bits).

Step 3. In the *ingress.p4* file, add the following code.

```
#define SKETCH_APPLY(num) hash(meta.index_sketch##num, \
                                HashAlgorithm.crc32_custom, (bit<16>)0, \
                                    { \
                                        hdr.ipv4.srcAddr, \
                                        hdr.ipv4.dstAddr, \
                                        hdr.tcp.srcPort, \
                                        hdr.tcp.dstPort, \
                                        hdr.ipv4.protocol \
                                    },\
                                    (bit<32>)SKETCH_LENGTH); \
    sketch##num.read(meta.value_sketch##num, meta.index_sketch##num); \
    meta.value_sketch##num = meta.value_sketch##num + 1; \
    sketch##num.write(meta.index_sketch##num, meta.value_sketch##num); \
    if(minimum > meta.value_sketch##num) { \
        minimum = meta.value_sketch##num; }
```



```
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/
#define THRESHOLD 20000

#define SKETCH_REGISTER(num) register<bit<SKETCH_WIDTH>>(SKETCH_LENGTH) sketch##num

#define SKETCH_APPLY(num) hash(meta.index_sketch##num, \
                                HashAlgorithm.crc32_custom, (bit<16>)0, \
                                { \
                                    hdr.ipv4.srcAddr, \
                                    hdr.ipv4.dstAddr, \
                                    hdr.tcp.srcPort, \
                                    hdr.tcp.dstPort, \
                                    hdr.ipv4.protocol \
                                },\
                                (bit<32>) SKETCH_LENGTH); \
    sketch##num.read(meta.value_sketch##num, meta.index_sketch##num); \
    meta.value_sketch##num = meta.value_sketch##num + 1; \
    sketch##num.write(meta.index_sketch##num, meta.value_sketch##num); \
    if(minimum > meta.value_sketch##num) { \
        minimum = meta.value_sketch##num; }
```

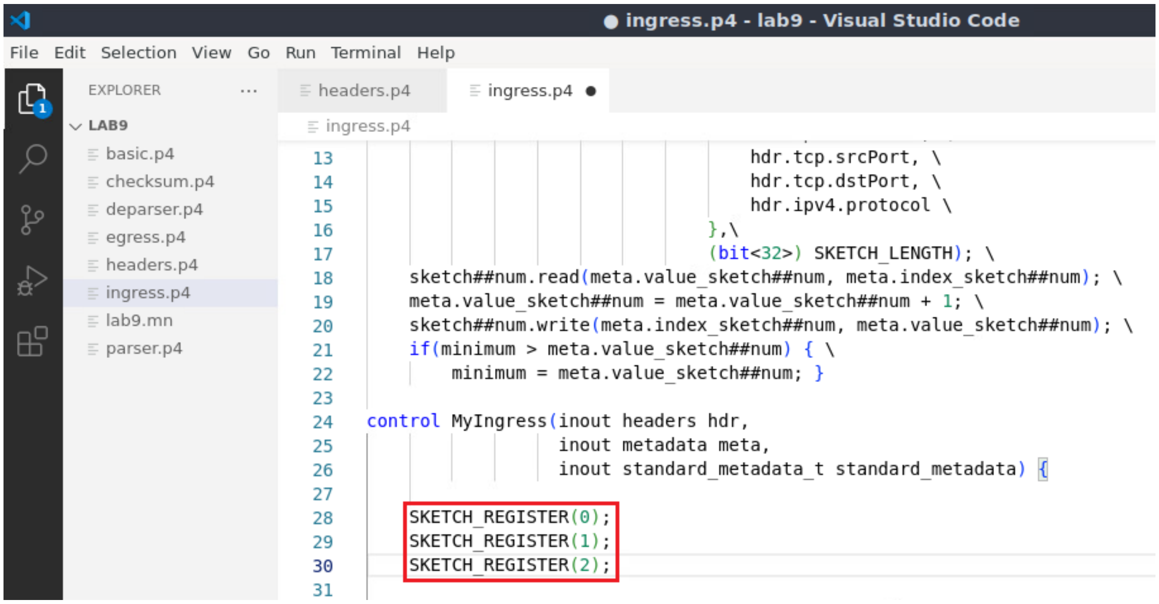
Figure 15. Defining SKETCH_APPLY.

The code above defines SKETCH_APPLY function that takes the number of the sketch as input and performs the actions to be described next on that sketch. For simplicity, assume that SKETCH_APPLY(0) is called. The hash of the 5-tuple (source IP, destination IP, source port, destination port, protocol) is stored inside meta.index_sketch0 using HashAlgorithm.crc32_custom hashing algorithm. The hashing algorithm uses different offsets for each sketch to assure that the hash functions of different sketches produce different hash values for the same 5-tuple. The offsets are populated by the control plane.

After calculating the index, the number of packets stored in sketch0 register at that index is retrieved and stored inside meta.value_sketch0. meta.value_sketch0 is then incremented by one to account for the current packet. Next, the updated number of packets is stored inside the sketch0 register at meta.index_sketch0. The updated value (i.e., meta.value_sketch0) is compared to the variable minimum. If the value is smaller than minimum, minimum is updated to be meta.value_sketch0.

Step 4. In the *MyIngress* control block, add the following code to initiate three sketch registers.

```
SKETCH_REGISTER(0);
SKETCH_REGISTER(1);
SKETCH_REGISTER(2);
```

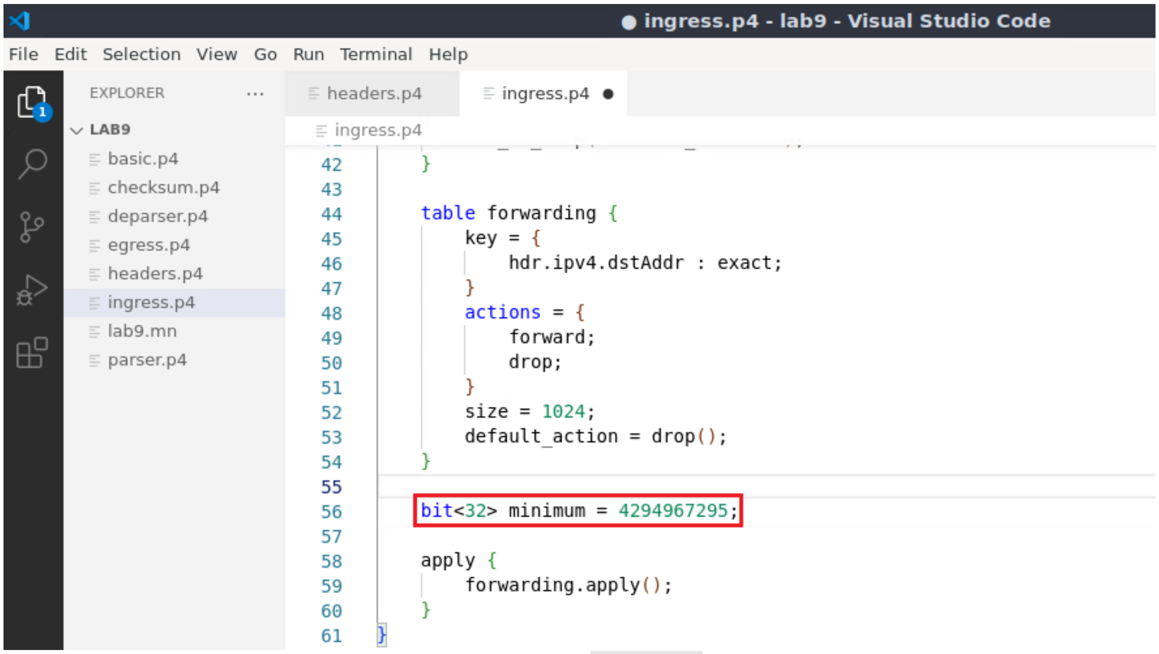


```
                                    hdr.tcp.srcPort, \
                                    hdr.tcp.dstPort, \
                                    hdr.ipv4.protocol \
                                },\
                                (bit<32>) SKETCH_LENGTH); \
    sketch##num.read(meta.value_sketch##num, meta.index_sketch##num); \
    meta.value_sketch##num = meta.value_sketch##num + 1; \
    sketch##num.write(meta.index_sketch##num, meta.value_sketch##num); \
    if(minimum > meta.value_sketch##num) { \
        minimum = meta.value_sketch##num; }

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    SKETCH_REGISTER(0);
    SKETCH_REGISTER(1);
    SKETCH_REGISTER(2);

```

Figure 16. Initiating three sketch registers.

Step 5. In the *MyIngress* control block, add the following code to declare a variable `minimum`. The value of `minimum` is set to a large number so that the variable will be larger than the `meta.value_sketchnum` and be override after calling `SKETCH_APPLY(num)`.

```
bit<32> minimum = 4294967295;
```



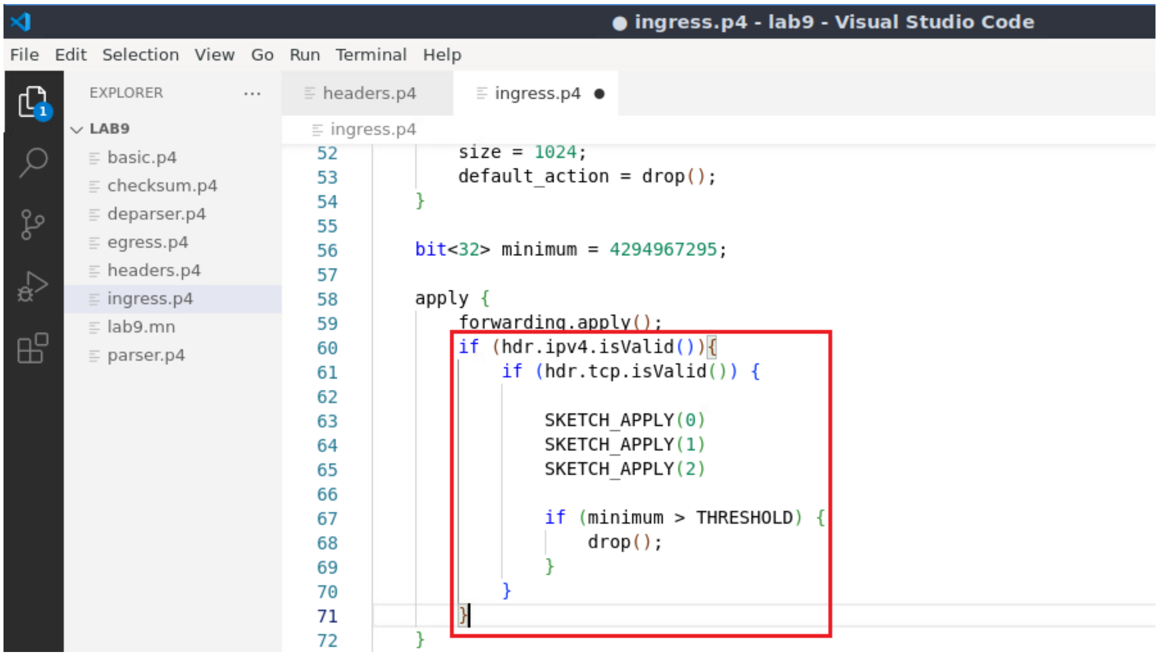
```
    }

    table forwarding {
        key = {
            hdr.ipv4.dstAddr : exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    bit<32> minimum = 4294967295;

    apply {
        forwarding.apply();
    }
}
```

Figure 17. Declaring `minimum` variable.

Step 6. Add the following code inside the *apply* block.

```
if(hdr.ipv4.isValid()) {
    if(hdr.tcp.isValid()) {

       SKETCH_APPLY(0)
       SKETCH_APPLY(1)
```

```
        SKETCH_APPLY(2)

        if(minimum > THRESHOLD) {
            drop();
        }
    }
}
```



```
        size = 1024;
        default_action = drop();
    }

    bit<32> minimum = 4294967295;

    apply {
        forwarding.apply();
        if (hdr.ipv4.isValid()){
            if (hdr.tcp.isValid()) {

                SKETCH_APPLY(0)
                SKETCH_APPLY(1)
                SKETCH_APPLY(2)

                if (minimum > THRESHOLD) {
                    drop();
                }
            }
        }
    }
```

Figure 18. Modifying the apply block.

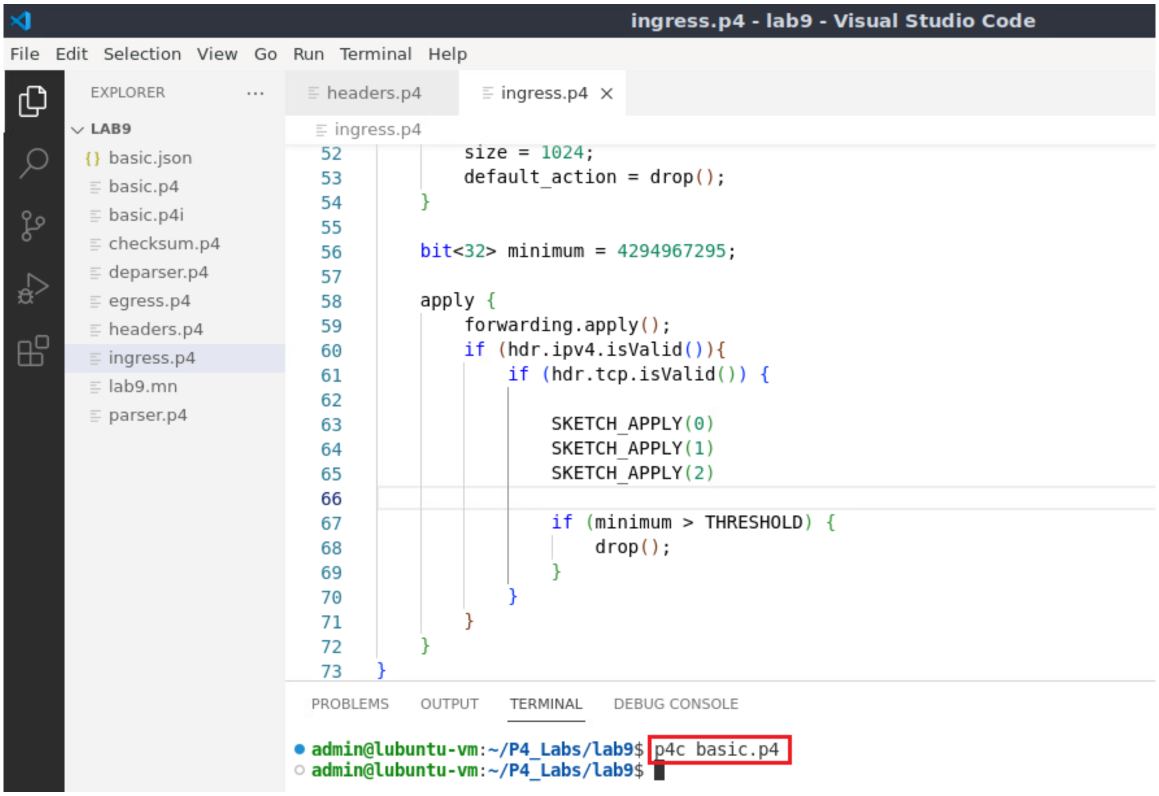
In the code above, `if(hdr.ipv4.isValid())` checks if the packet contains the IPv4 header. `if(hdr.tcp.isValid())` checks if the packet contains the TCP header. For a TCP packet, `SKETCH_APPLY` is called for the three sketch registers. If the minimum variable is larger than `THRESHOLD` (i.e., the flow has at least 20000 packets), the packet will be dropped.

Step 7. Save the changes to the file by pressing `Ctrl + s`.

3.4 Loading the program and configuring the switch

Step 1. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```



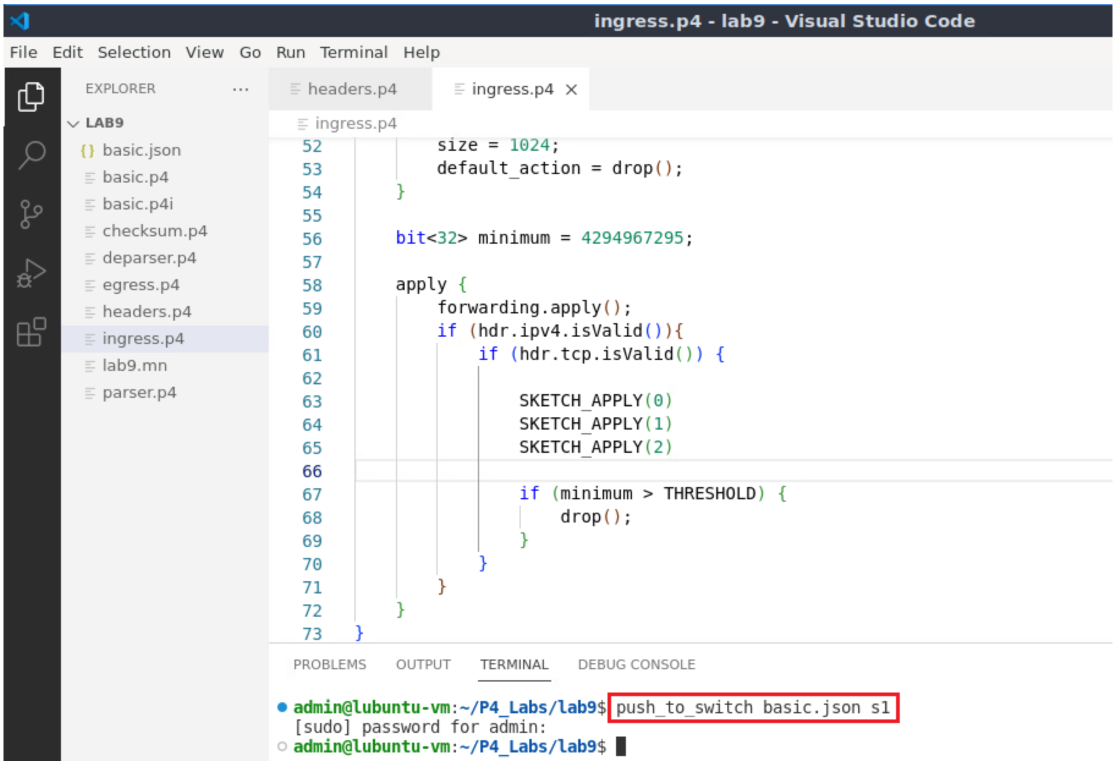
```
        size = 1024;
        default_action = drop();
    }

    bit<32> minimum = 4294967295;

    apply {
        forwarding.apply();
        if (hdr.ipv4.isValid()){
            if (hdr.tcp.isValid()) {

                SKETCH_APPLY(0)
                SKETCH_APPLY(1)
                SKETCH_APPLY(2)

                if (minimum > THRESHOLD) {
                    drop();
                }
            }
        }
    }
}
```

```
admin@lubuntu-vm:~/P4_Labs/lab9$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab9$ 
```

Figure 19. Compiling the P4 program using the VS Code terminal.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```

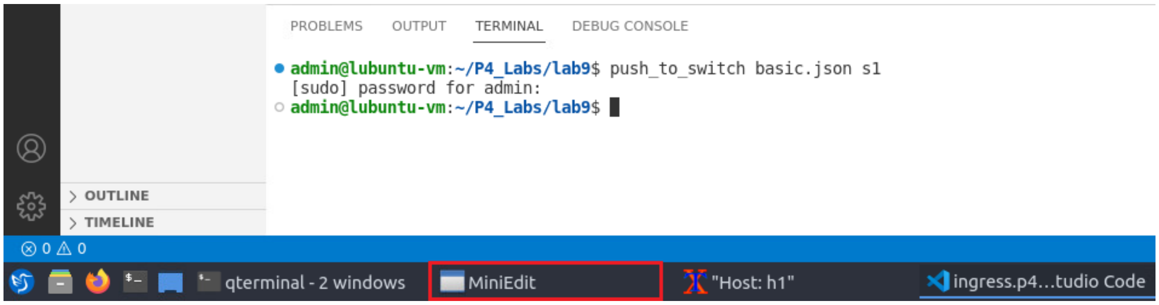


```
        size = 1024;
        default_action = drop();
    }

    bit<32> minimum = 4294967295;

    apply {
        forwarding.apply();
        if (hdr.ipv4.isValid()){
            if (hdr.tcp.isValid()) {

                SKETCH_APPLY(0)
                SKETCH_APPLY(1)
                SKETCH_APPLY(2)

                if (minimum > THRESHOLD) {
                    drop();
                }
            }
        }
    }
}
```

```
admin@lubuntu-vm:~/P4_Labs/lab9$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab9$ 
```

Figure 20. Downloading the compiled program to switch s1.

Step 3. Click on the MinEdit tab in the start bar to maximize the window.



```
admin@lubuntu-vm:~/P4_Labs/lab9$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab9$
```

Figure 21. Maximizing the MiniEdit window.

Step 4. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

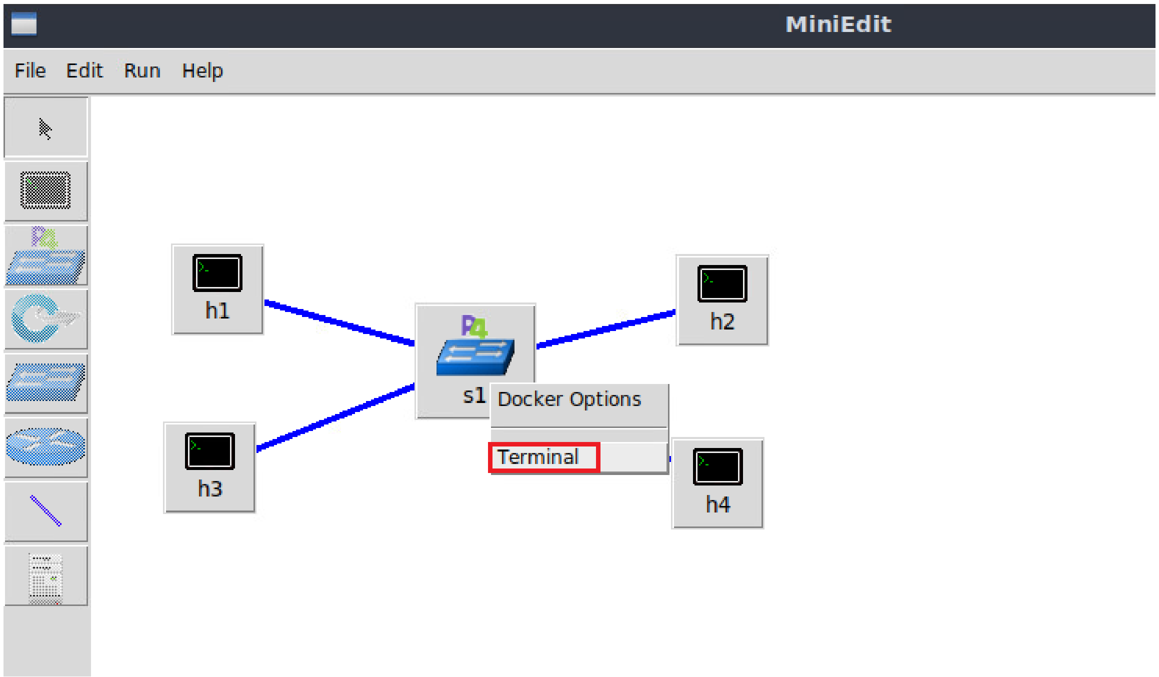
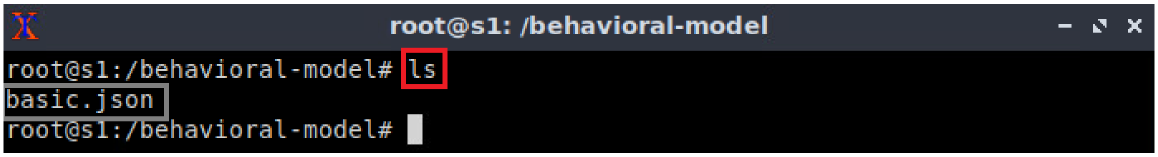


Figure 22. Starting the terminal on switch s1.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch's terminal.

Step 5. Issue the following command to list the files in the current directory.

```
ls
```



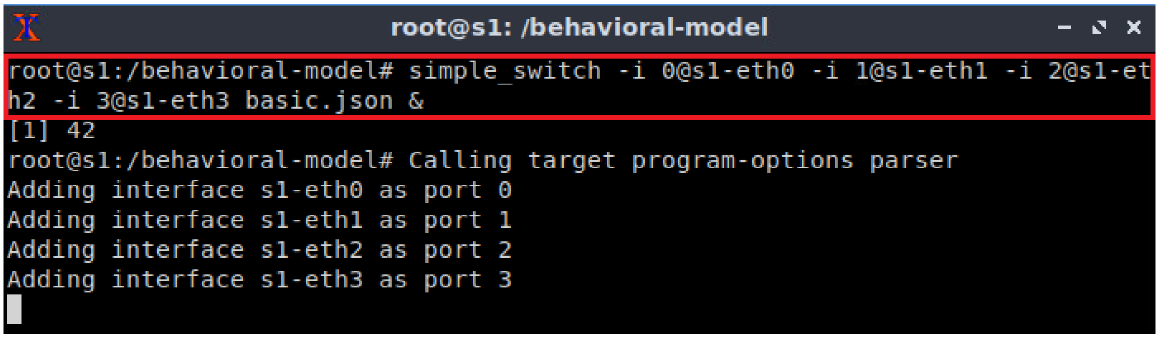
```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model#
```

Figure 23. Displaying the contents of the current directory in the switch s1.

We can see that the switch contains the *basic.json* file that was downloaded after compiling the P4 program.

Step 6. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

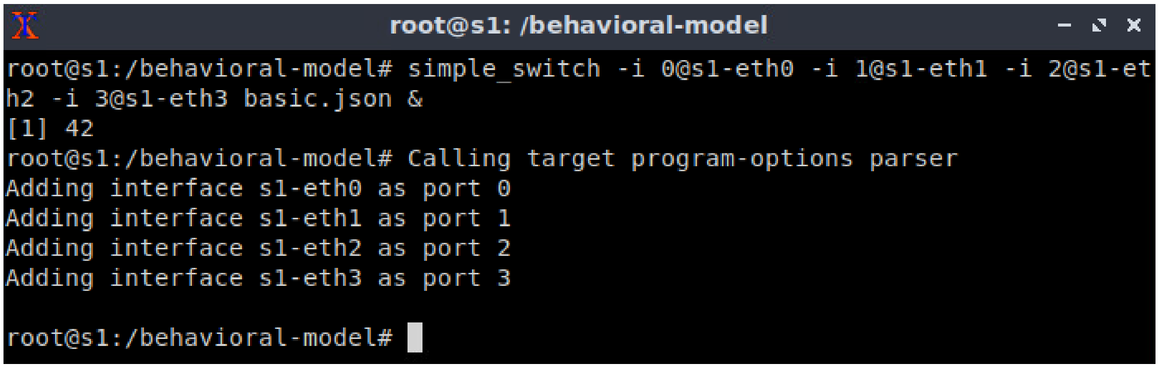
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 -i 3@s1-eth3 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 -i 3@s1-eth3 basic.json &
[1] 42
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
Adding interface s1-eth3 as port 3
```

Figure 24. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 7. In switch s1 terminal, press *Enter* to return the CLI.

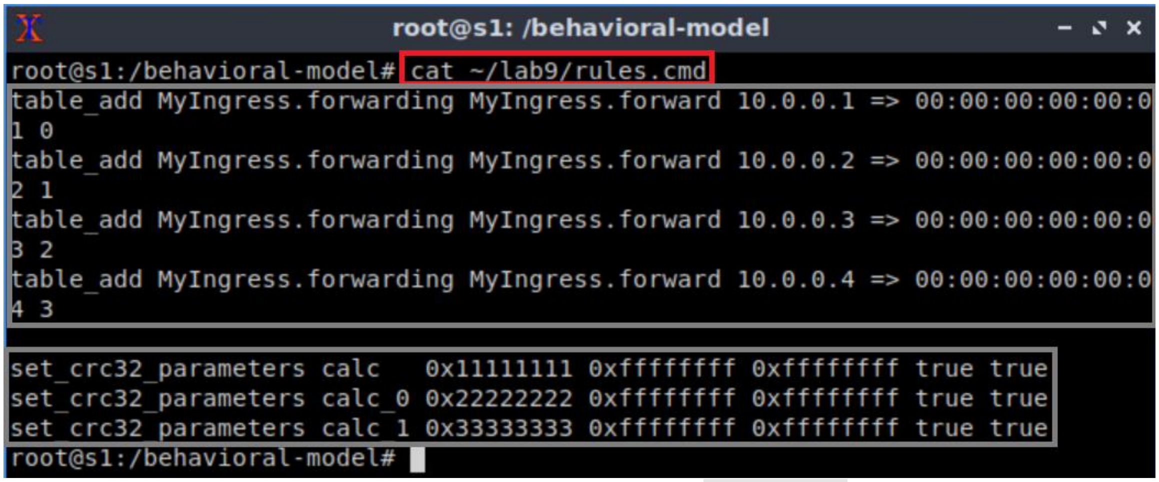


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 -i 3@s1-eth3 basic.json &
[1] 42
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
Adding interface s1-eth3 as port 3

root@s1:/behavioral-model#
```

Figure 25. Returning to switch s1 CLI.

Step 8. Issue the following command to inspect the rules to be populated by the control plane. `cat` command concatenates files and prints on the standard output.

```
cat ~/lab9/rules.cmd
```



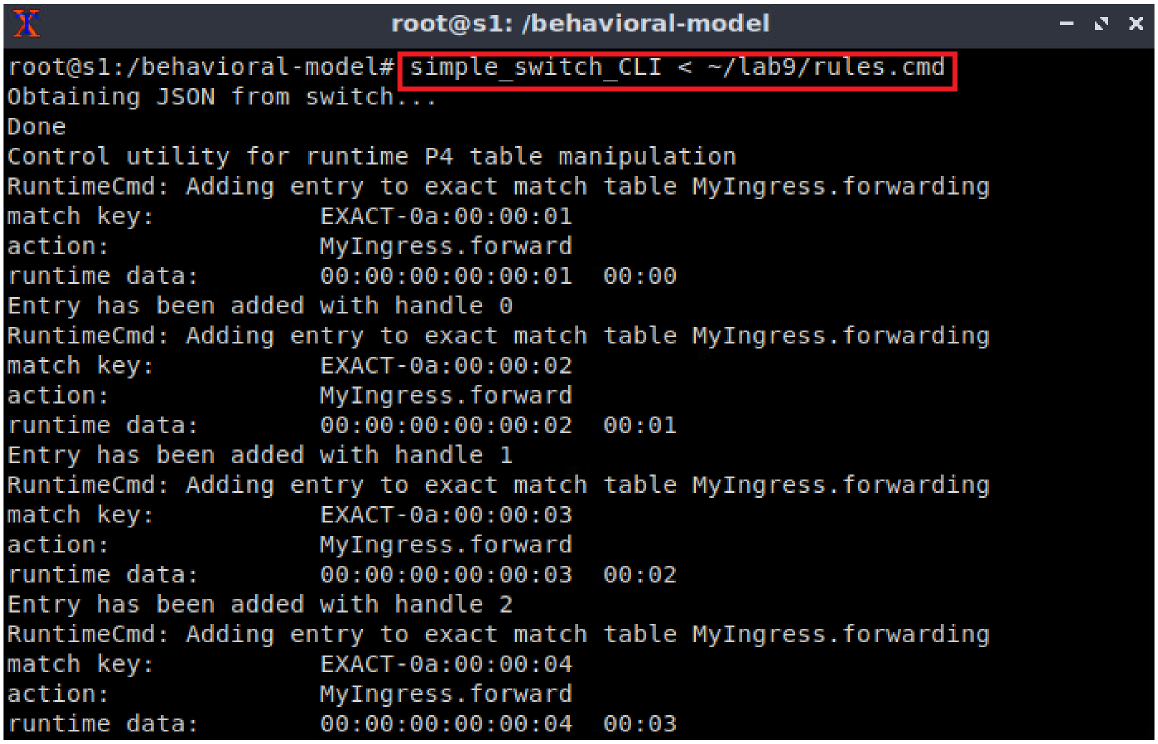
```
root@s1:/behavioral-model# cat ~/lab9/rules.cmd
table_add MyIngress.forwarding MyIngress.forward 10.0.0.1 => 00:00:00:00:00:0
1 0
table_add MyIngress.forwarding MyIngress.forward 10.0.0.2 => 00:00:00:00:00:0
2 1
table_add MyIngress.forwarding MyIngress.forward 10.0.0.3 => 00:00:00:00:00:0
3 2
table_add MyIngress.forwarding MyIngress.forward 10.0.0.4 => 00:00:00:00:00:0
4 3

set_crc32_parameters calc   0x11111111 0xffffffff 0xffffffff true true
set_crc32_parameters calc_0 0x22222222 0xffffffff 0xffffffff true true
set_crc32_parameters calc_1 0x33333333 0xffffffff 0xffffffff true true
root@s1:/behavioral-model#
```

Figure 26. Inspecting the contents of `rules.cmd` file.

The figure above displays the forwarding rules (first four rules in the file). The last three rules define three different seeds for the three hash functions of the sketches (sketch0, sketch1, and sketch2). By having different seeds, the three hash functions will output three different hash values for the same input. Note that the hash functions use the same hashing algorithm and that it is necessary to have different seeds for the functions to output different hash values.

Step 9. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab9/rules.cmd
```



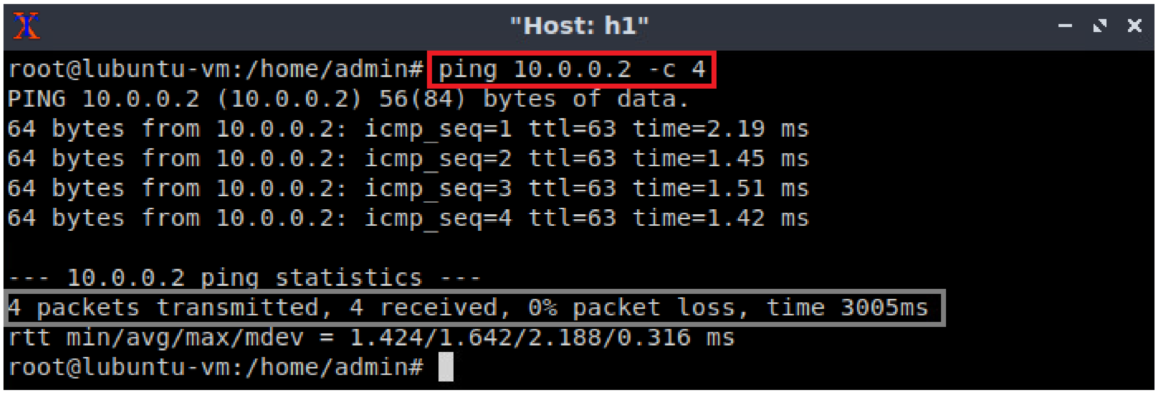
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab9/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:02
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:03
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 2
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:04
action:              MyIngress.forward
runtime data:        00:00:00:00:00:04   00:03
```

Figure 27. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

Step 10. Go back to host h1 terminal to test the connectivity between host h1 and host h2 by issuing the following command.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=63 time=2.19 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=63 time=1.45 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=63 time=1.51 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=63 time=1.42 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.424/1.642/2.188/0.316 ms
root@lubuntu-vm:/home/admin#
```

Figure 28. Performing a connectivity test between host h1 and host h2.

The figure above shows that there is connectivity between the two hosts.

4 Testing the P4 program

Step 1. Hold the right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

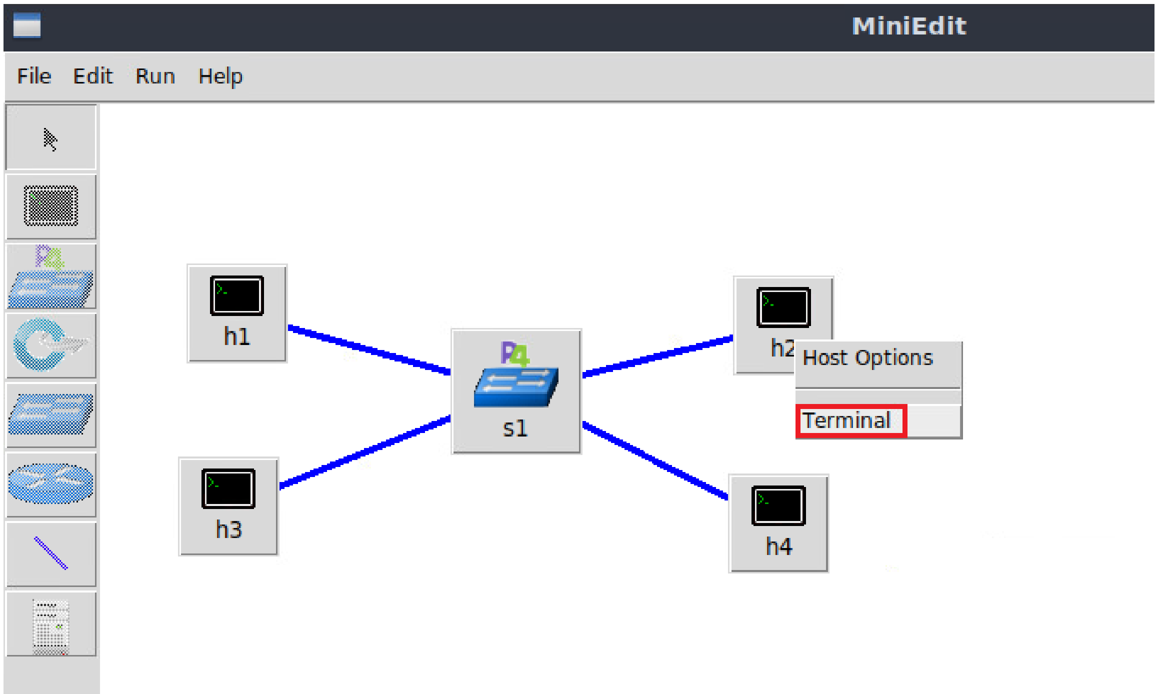
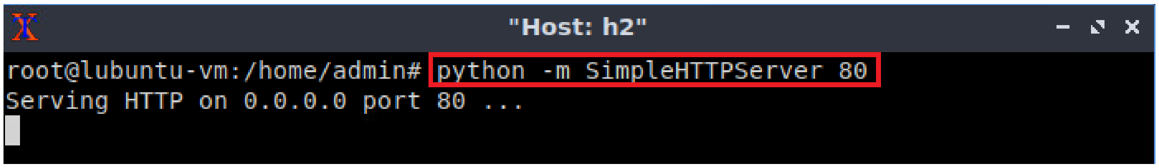


Figure 29. Opening a terminal on host h2.

Step 2. On h2 terminal, type the command below to start an HTTP server using Python. `-m` is used to run a module as a script, allowing the execution of Python module directly from the command line. `SimpleHTTPServer` is a Python 2 module that provides a basic HTTP server capable of serving static files from the current directory. The server will be listening on port `80` for incoming packets.

```
python -m SimpleHTTPServer 80
```



```
root@lubuntu-vm:/home/admin# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Figure 30. Starting HTTP server on h2.

Step 3. Hold the right-click on host h4 and select *Terminal*. This opens the terminal of host h4 and allows the execution of commands on that host.

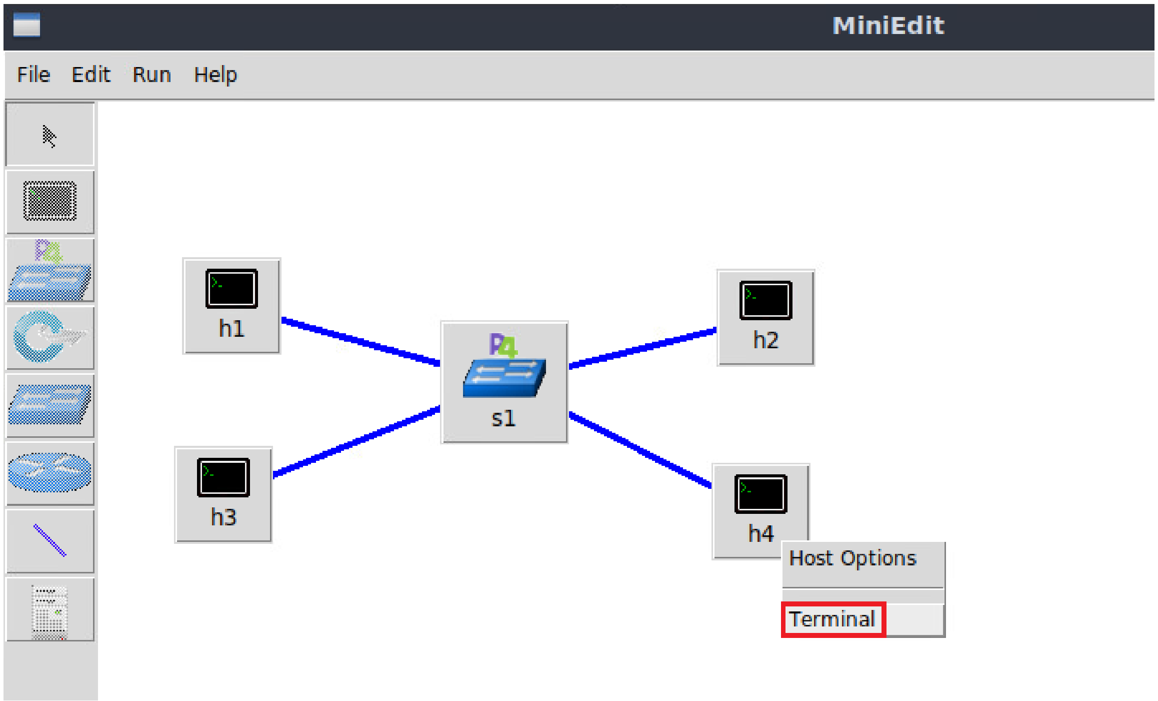
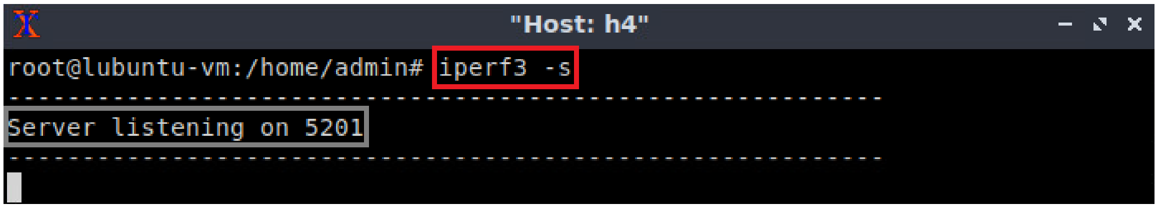


Figure 31. Opening a terminal on host h4.

Step 4. On h4 terminal, type the command below to start `iperf3` as a server. `iperf3` is a tool for performing network throughput measurements. `-s` option runs `iperf3` in server mode.

```
iperf3 -s
```



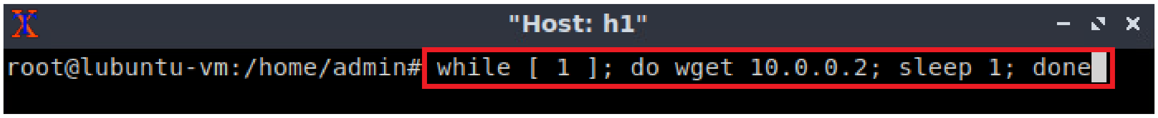
```
root@lubuntu-vm:/home/admin# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 32. Starting iperf3 server on h4.

The figure above shows that the server is listening on port 5201.

Step 5. On h1 terminal, type the command below to continuously issue HTTP requests. The script utilized `wget` to perform HTTP Get request every 1 second. `wget` is a utility for non-interactive download of files from the Web. `10.0.0.2` is the IP address of the HTTP server. `sleep 1` command causes the calling thread to sleep for 1 second.

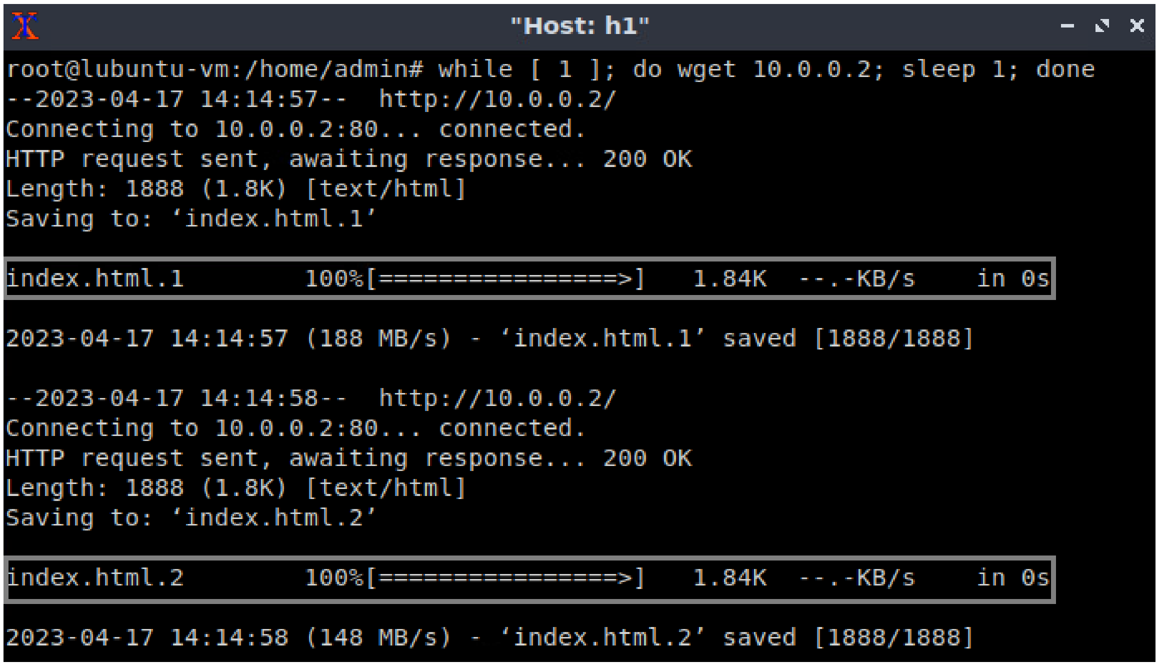
```
while [ 1 ]; do wget 10.0.0.2; sleep 1; done
```



```
root@lubuntu-vm:/home/admin# while [ 1 ]; do wget 10.0.0.2; sleep 1; done
```

Figure 33. Issuing repetitive HTTP GET requests from h1.

Step 6. Inspect h1 terminal.



```
root@lubuntu-vm:/home/admin# while [ 1 ]; do wget 10.0.0.2; sleep 1; done
--2023-04-17 14:14:57--  http://10.0.0.2/
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1888 (1.8K) [text/html]
Saving to: ‘index.html.1’

index.html.1        100%[================>]   1.84K  --.-KB/s    in 0s

2023-04-17 14:14:57 (188 MB/s) - ‘index.html.1’ saved [1888/1888]

--2023-04-17 14:14:58--  http://10.0.0.2/
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1888 (1.8K) [text/html]
Saving to: ‘index.html.2’

index.html.2        100%[================>]   1.84K  --.-KB/s    in 0s

2023-04-17 14:14:58 (148 MB/s) - ‘index.html.2’ saved [1888/1888]
```

Figure 34. Inspecting h1 terminal.

The figure above shows that the client successfully downloaded the file *index.html* twice from the HTTP server.

Step 7. Hold the right-click on host h3 and select *Terminal*. This opens the terminal of host h3 and allows the execution of commands on that host.

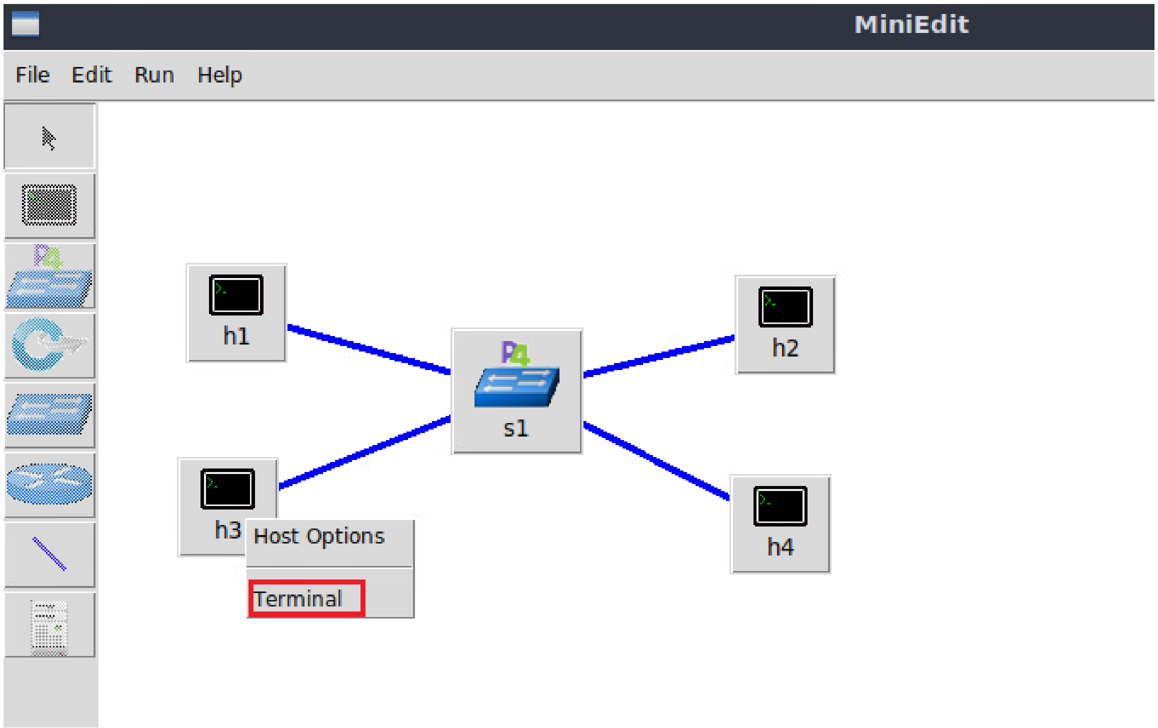
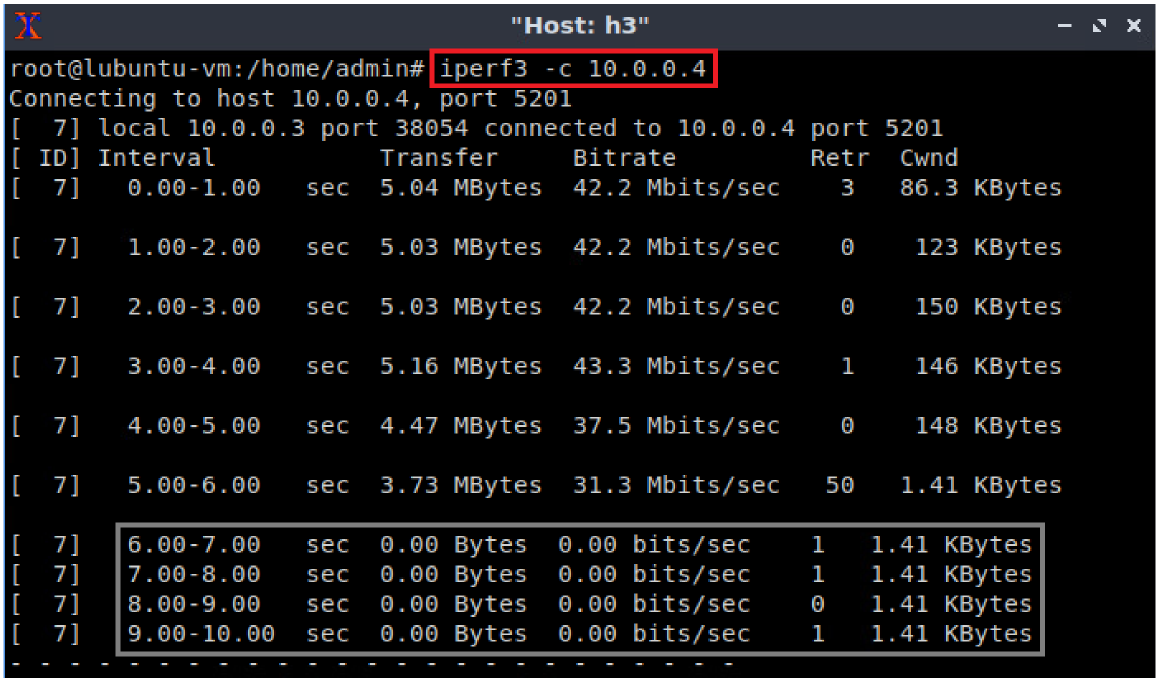


Figure 35. Opening a terminal on host h3.

Step 8. On h3 terminal, type the command below to start `iperf3` as a client. `-c` option runs iperf3 in client mode. `10.0.0.4` is the IP address of the iperf3 server.

```
iperf3 -c 10.0.0.4
```

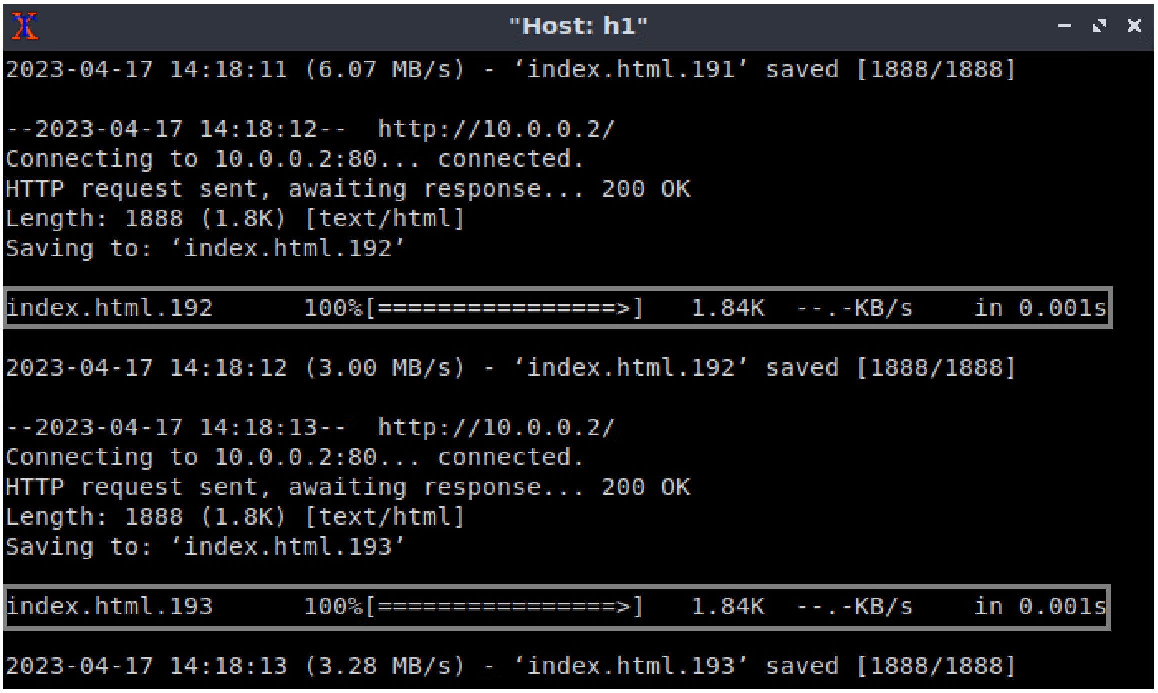


```
root@lubuntu-vm:/home/admin# iperf3 -c 10.0.0.4
Connecting to host 10.0.0.4, port 5201
[  7] local 10.0.0.3 port 38054 connected to 10.0.0.4 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  7]   0.00-1.00   sec  5.04 MBytes  42.2 Mbits/sec    3   86.3 KBytes

[  7]   1.00-2.00   sec  5.03 MBytes  42.2 Mbits/sec    0    123 KBytes

[  7]   2.00-3.00   sec  5.03 MBytes  42.2 Mbits/sec    0    150 KBytes

[  7]   3.00-4.00   sec  5.16 MBytes  43.3 Mbits/sec    1    146 KBytes

[  7]   4.00-5.00   sec  4.47 MBytes  37.5 Mbits/sec    0    148 KBytes

[  7]   5.00-6.00   sec  3.73 MBytes  31.3 Mbits/sec   50   1.41 KBytes

[  7]   6.00-7.00   sec  0.00 Bytes  0.00 bits/sec    1   1.41 KBytes
[  7]   7.00-8.00   sec  0.00 Bytes  0.00 bits/sec    1   1.41 KBytes
[  7]   8.00-9.00   sec  0.00 Bytes  0.00 bits/sec    0   1.41 KBytes
[  7]   9.00-10.00  sec  0.00 Bytes  0.00 bits/sec    1   1.41 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 36. Starting iperf3 test between h3 and h4.

The figure above shows that the bitrate dropped to zero at second 6. This occurs because the number of packets transferred by the test exceeded the defined threshold and the switch blacklisted the flow.

Step 9. Inspect h1 terminal.



```
2023-04-17 14:18:11 (6.07 MB/s) - ‘index.html.191’ saved [1888/1888]

--2023-04-17 14:18:12--  http://10.0.0.2/
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1888 (1.8K) [text/html]
Saving to: ‘index.html.192’

index.html.192      100%[================>]   1.84K  --.-KB/s    in 0.001s

2023-04-17 14:18:12 (3.00 MB/s) - ‘index.html.192’ saved [1888/1888]

--2023-04-17 14:18:13--  http://10.0.0.2/
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1888 (1.8K) [text/html]
Saving to: ‘index.html.193’

index.html.193      100%[================>]   1.84K  --.-KB/s    in 0.001s

2023-04-17 14:18:13 (3.28 MB/s) - ‘index.html.193’ saved [1888/1888]
```

Figure 37. Inspecting h1 terminal.

The figure above shows that all the GET requests are successful. Note that the switch did not drop the flows because the number of packets of an HTTP GET request is smaller than the heavy hitter detection threshold.

This concludes lab 9. Stop the emulation and then exit out of MiniEdit.

References

1. Itamar Haber, “Count-Min Sketch: The Art and Science of Estimating Stuff.” [Online]. Available: https://tinyurl.com/9f6ynpm2
2. Yu, Minlan, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch." NSDI. Vol. 13. 2013.
3. Brandon Fain, “Count Min-Sketch: The Heavy Hitters Problem.” [Online]. Available: https://tinyurl.com/mtswjdmf
4. Cormode, Graham. "Count-Min Sketch." (2009): 511-516.

UNIVERSITY OF
SOUTH CAROLINA

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 10: Limiting the Impact of SYN Flood by Probabilistically Dropping Packets

Document Version: 04-20-2023

Contents

Overview

This lab introduces the TCP SYN flood attack and provides the steps to implement a P4 program that mitigates the attack. In TCP SYN flood, the attacker initiates many TCP connections without completing the handshake process. To mitigate this attack, the user will define a policy to drop a percentage of packets when the number of received SYN packets per second exceeds a predefined threshold. The user will utilize P4 registers to store the dropping percentage, so that the dropping percentage can be modified at runtime.

Objectives

By the end of this lab, students should be able to:

1. Define the TCP SYN flood attack.
2. Understand the workflow of the TCP SYN flood attack.
3. Perform a TCP SYN flood attack.
4. Write a P4 program that mitigates the TCP SYN flood attack.

Lab settings

Table 1 contains the credentials of the virtual machine used for this lab.

Table 1. Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Developing a P4 program to mitigate SYN flood attack.
4. Section 4: Testing the P4 code against SYN flood attack.

1 Introduction

Volumetric DoS attacks flood the target machine with traffic, depriving legitimate users from downloading the target’s resources[1]. Today, most attackers launch Distributed DoS

(DDoS) to amplify the attack's volume. In DDoS, an attacker instructs hundreds or thousands of machines to flood a target server with requests. DoS attacks typically spoof the source IP address of the packets to hide the identity of the attacker.

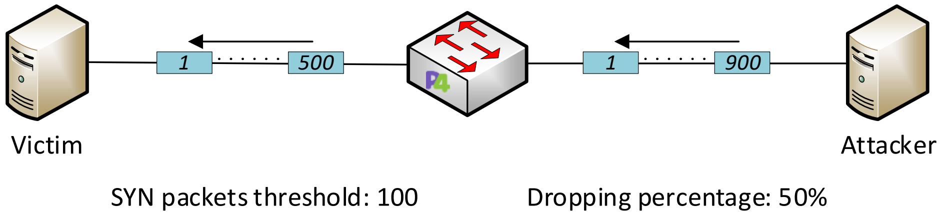
DoS attacks can be performed at various levels of the protocol stack. For instance, an attacker can launch a DoS attack by leveraging a transport layer protocol such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Examples of transport layer flood attacks include TCP SYN flood[2].

TCP SYN flood is an attack that initiates many TCP connections without completing the TCP handshake[4]. The TCP handshake process includes three steps: 1) the client sends a TCP SYN packet to the server; the server reserves resources that will be dedicated to the TCP connection; 2) the server responds with a TCP SYN-ACK packet; and 3) the client sends a TCP ACK packet. In SYN flood attack, the attacker does not complete the handshake (it stops at step 2), leaving the server in a waiting state. When many SYN requests are sent, all the resources of the server will be allocated, which prevents legitimate users from accessing the server.

1.1 Lab scenario

In this lab, a P4 programmable switch will mitigate the SYN flood attack by performing random packet drop when the number of received SYN packets exceeds a predefined threshold. The programmable switch detects SYN flood by monitoring the number of SYN packets. During an attack, the number of SYN packets will be larger than some threshold. The switch counts and compares the number of SYN packets per second against that threshold. If the number is larger, the switch considers that a SYN flood attack is being performed. Consequently, the switch starts dropping subsequent SYN packets based on a dropping percentage specified from the control plane. The switch exits from the mitigation phase when the count of SYN packets per second drops below the threshold.

Consider Figure 1. Assume that the SYN packets threshold is 100, the dropping percentage is 50 (i.e., 50 packets out of 100 will be dropped), and the attacker is sending 900 packets per second to the victim. Because the number of the sent packets is larger than the threshold, the switch will drop 50% of the packets above the threshold (i.e., out of the 900 packets, 100 will be normally forwarded, and the 50% dropping rate will be applied to the remaining 800 packets per second). Thus, the switch will forward 500 packets per second only.



SYN packets threshold: 100 Dropping percentage: 50%

Figure 1. Lab scenario.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

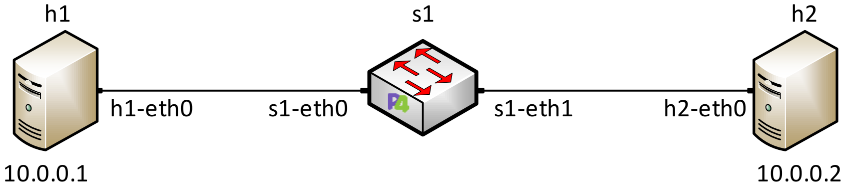


Figure 2. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

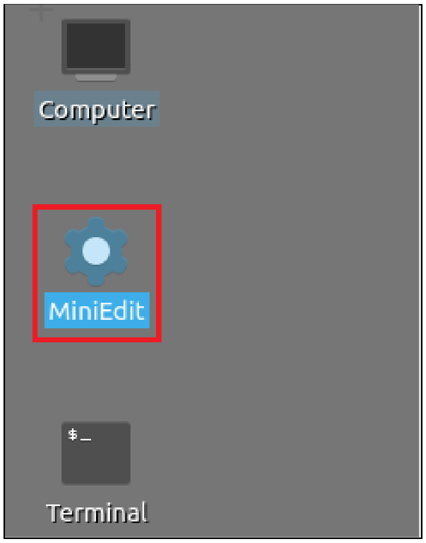


Figure 3. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab10*, select the file *lab10.mn,* and click on *Open*.

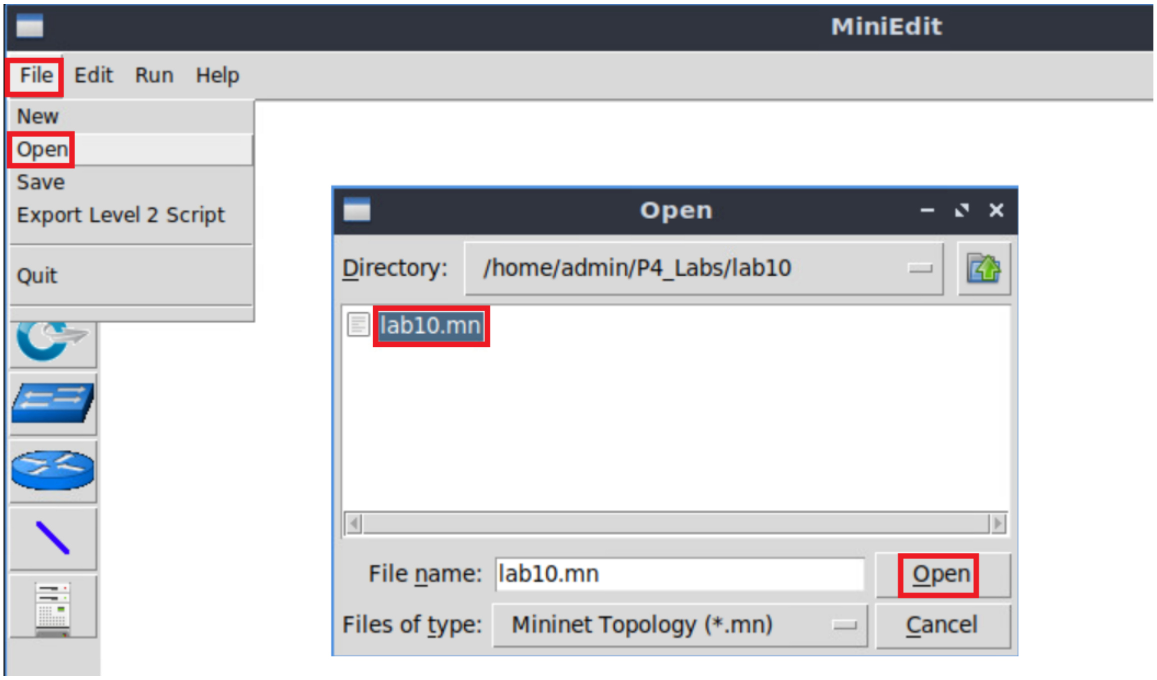


Figure 4. Opening a topology in MiniEdit.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

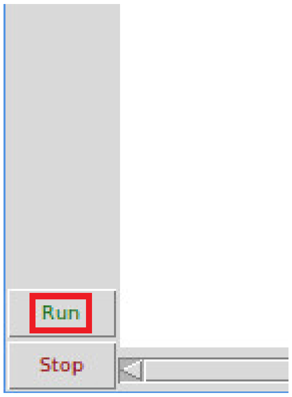


Figure 5. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

Step 1. Hold the right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

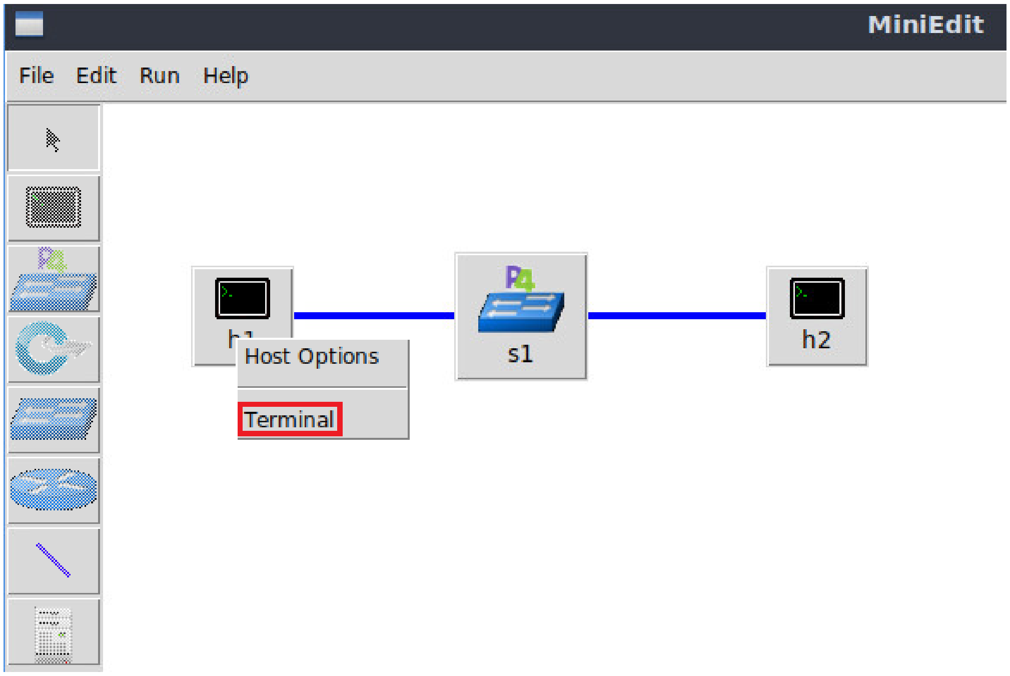
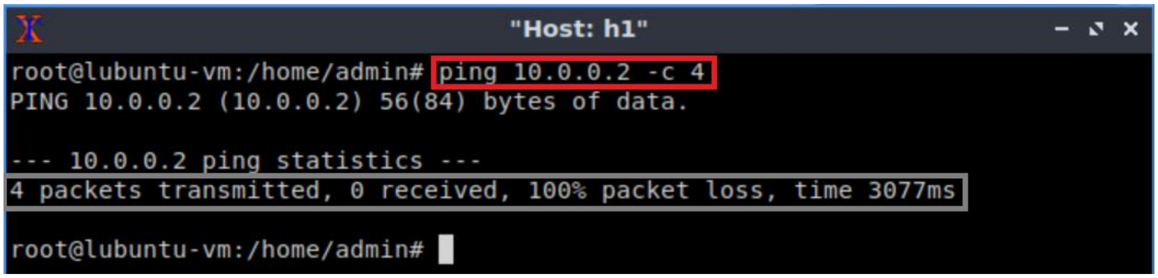


Figure 6. Opening a terminal on host h1.

Step 2. Test the connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3077ms

root@lubuntu-vm:/home/admin#
```

Figure 7. Performing a connectivity test between host h1 and host h2.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded into the switch.

3 Developing a P4 program to mitigate SYN flood attack

In this section, a basic P4 program will be modified to mitigate SYN flood attack. To do this, stateful registers will be used to track the number of received SYN packets per second. If the number of packets exceeds a predefined threshold, the switch will start dropping packets. The percentage to be dropped is stored in a stateful register and can be configured from the control plane.

3.1 Loading the environment

Step 1. Launch a Linux terminal by double-clicking on the Linux terminal icon located on the desktop. Alternatively, you can click on the icon in the taskbar located in lower left-hand side.

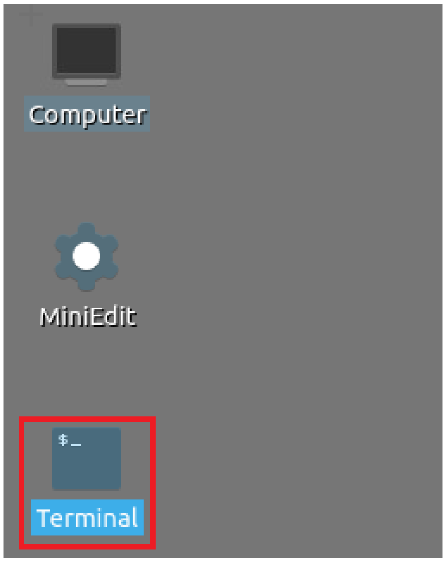
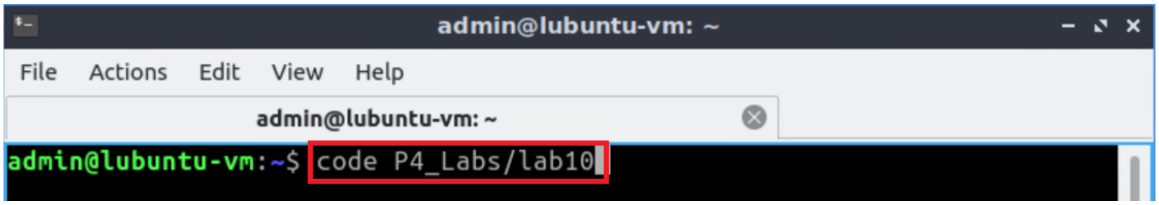


Figure 8. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the VS Code and opens the directory where the P4 program for this lab is located.

```
code P4_Labs/lab10
```



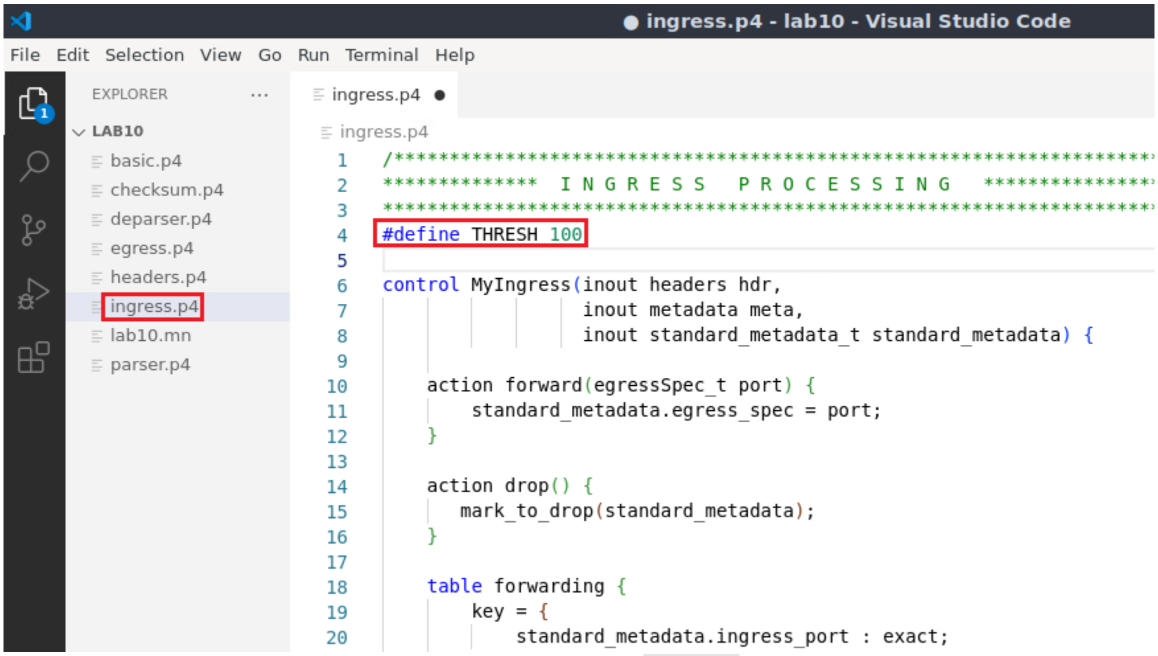
```
admin@lubuntu-vm:~$ code P4_Labs/lab10
```

Figure 9. Launching the editor and opening the *lab10* directory.

3.2 Modifying the ingress file

Step 1. In the *ingress.p4* file, define the variable `THRESH`. `THRESH` represents the threshold of received SYN packets per second after which the switch will start dropping packets. The threshold is set to 100 packets.

```
#define THRESH 100
```

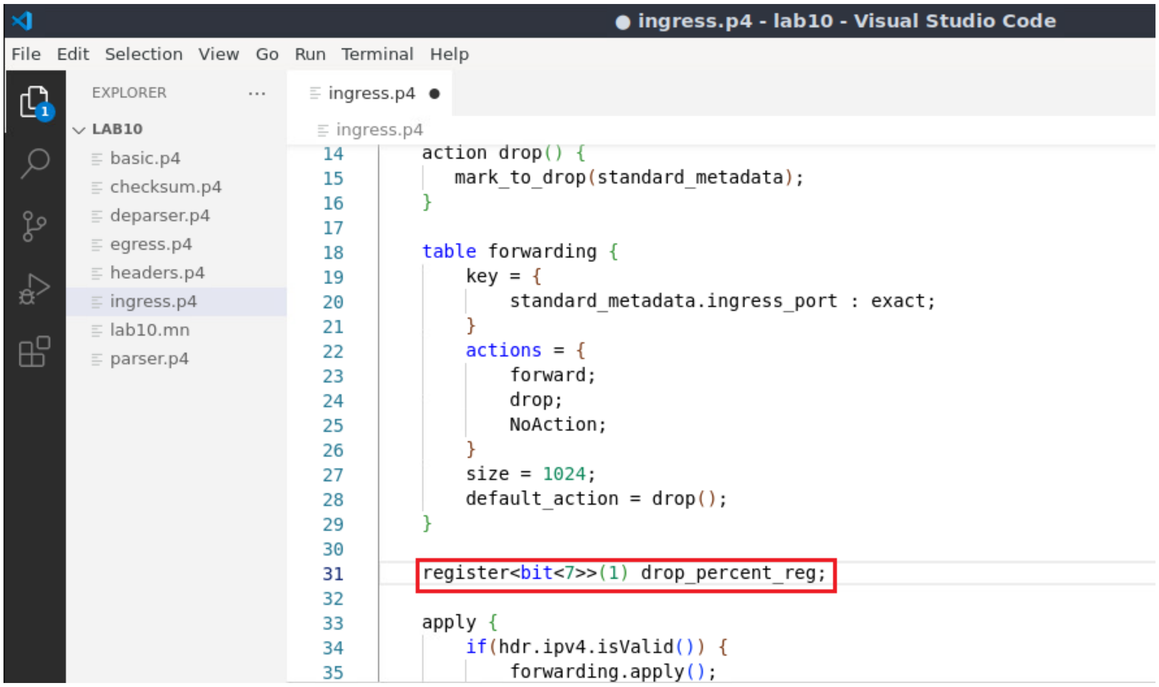


```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************
#define THRESH 100

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action forward(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }

    action drop() {
        mark_to_drop(standard_metadata);
    }

    table forwarding {
        key = {
            standard_metadata.ingress_port : exact;
```

Figure 10. Defining THRESH.

Step 2. Define the register drop_percent_reg to maintain the percentage of packets to be dropped after the number of received SYN packets increases above the threshold.

```
register<bit<7>>(1) drop_percent_reg;
```



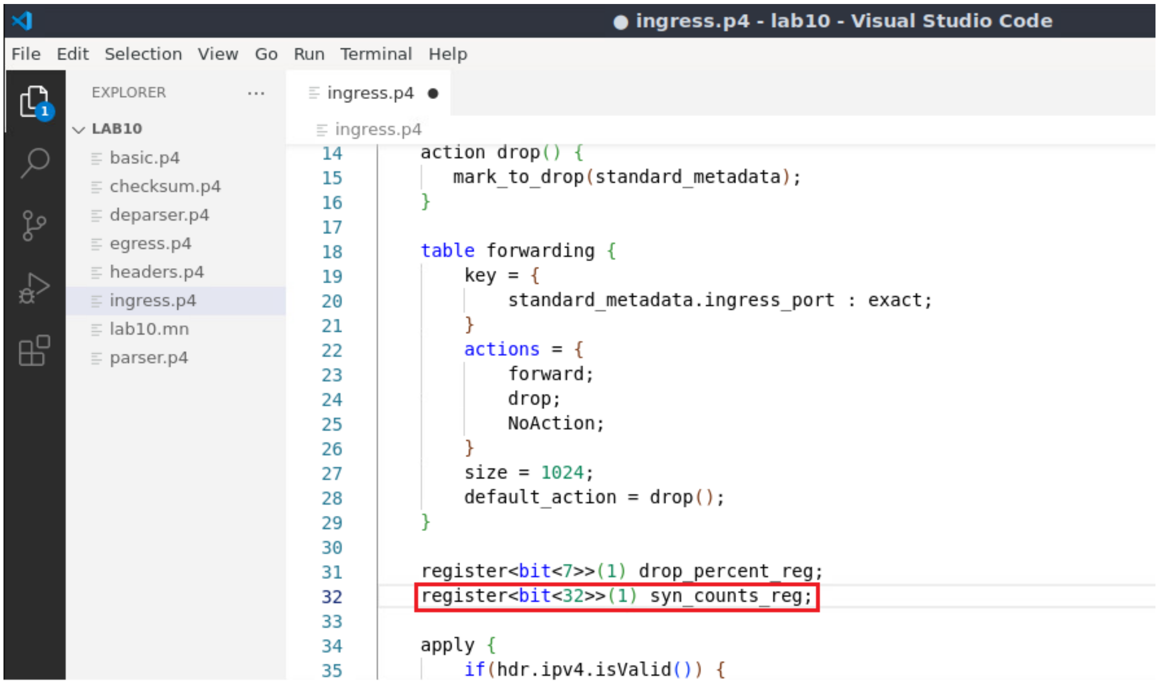
```
    action drop() {
        mark_to_drop(standard_metadata);
    }

    table forwarding {
        key = {
            standard_metadata.ingress_port : exact;
        }
        actions = {
            forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    register<bit<7>>(1) drop_percent_reg;

    apply {
        if(hdr.ipv4.isValid()) {
            forwarding.apply();
```

Figure 11. Defining register to store the packet drop percentage.

The code above defines a register named drop_percent_reg. The register contains a single cell. The cell stores the percentage of packets to be dropped. This register can be configured from the control plane at runtime to specify the dropping percentage. Because the maximum possible dropping percentage is 100, and 7 is the minimum number of bits needed to represent 100 (since $2^7 = 128$), the size of the register cell is set to 7 bits.

Step 3. Define the register `syn_counts_reg` to maintain the count of the received SYN packets.

```
register<bit<32>>(1) syn_counts_reg;
```



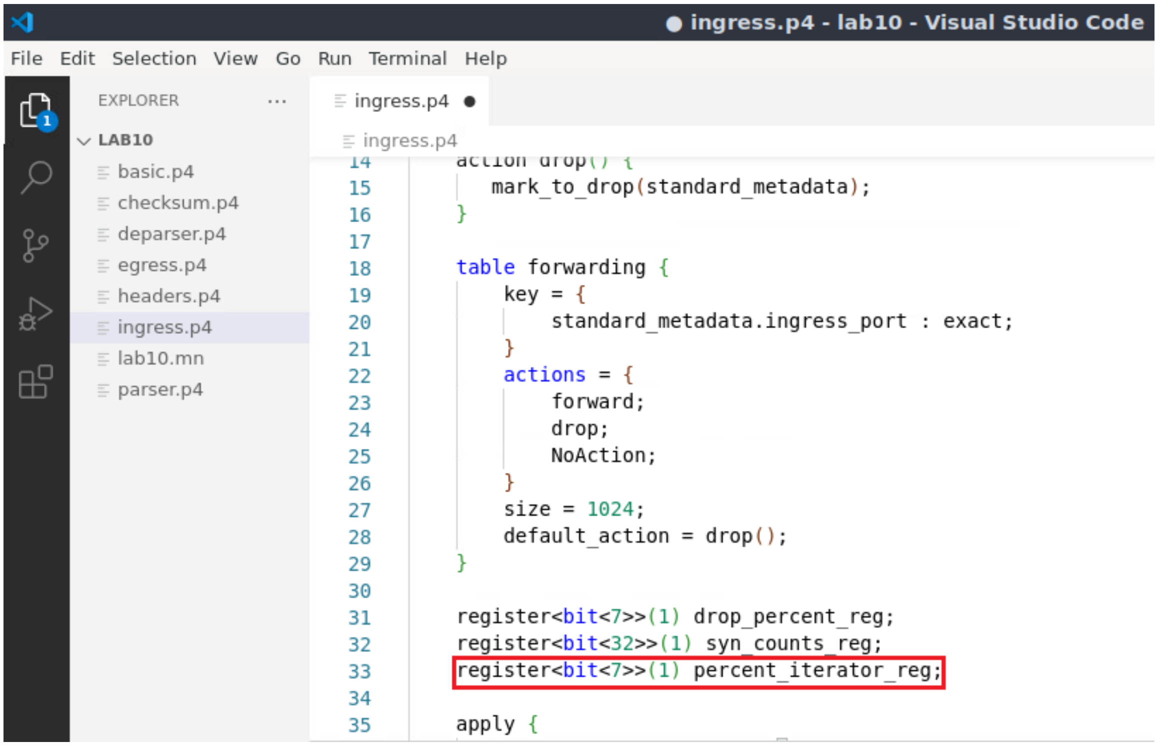
```
action drop() {
    mark_to_drop(standard_metadata);
}

table forwarding {
    key = {
        standard_metadata.ingress_port : exact;
    }
    actions = {
        forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
}

register<bit<7>>(1) drop_percent_reg;
register<bit<32>>(1) syn_counts_reg;

apply {
    if(hdr.ipv4.isValid()) {
```

Figure 12. Defining register to store the number of received SYN packets.

The code above defines a register named `syn_counts_reg`. The register contains a single cell. The cell stores the number of received SYN packets. Later, we will be resetting the value of this register to zero; thus, this cell will contain the number of received SYN packets per second.

Step 4. Define the register `percent_iterator_reg` to maintain the packet count iterator.

```
register<bit<7>>(1) percent_iterator_reg;
```



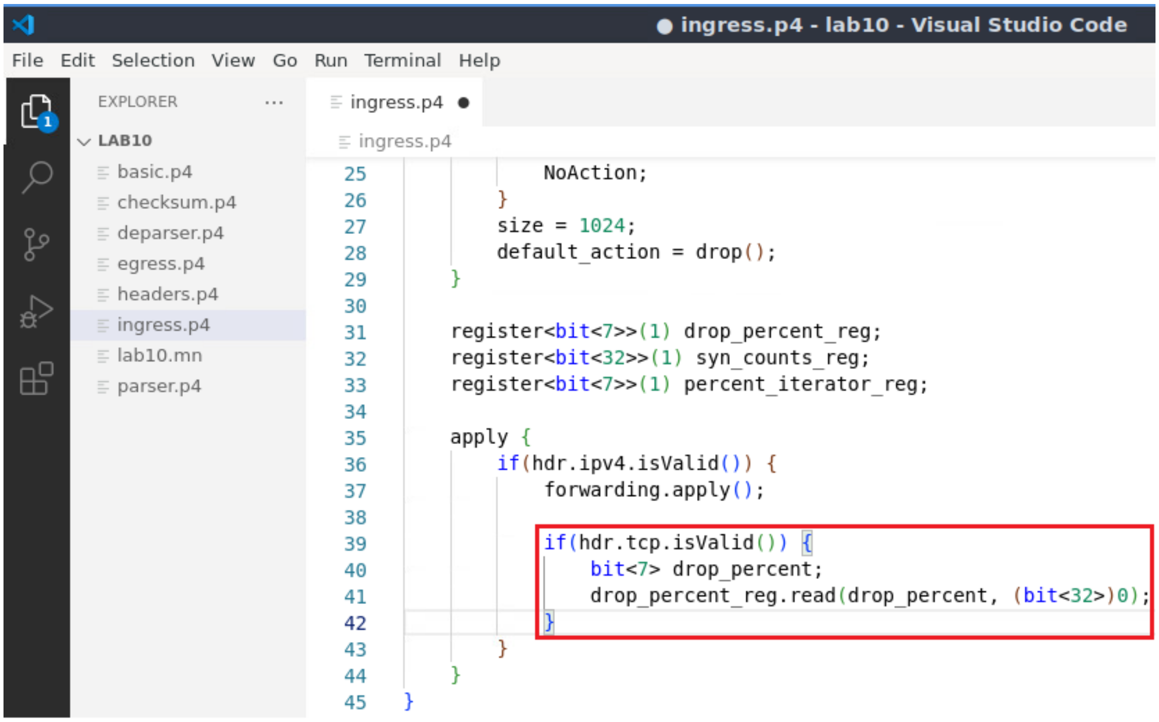
```
action drop() {
    mark_to_drop(standard_metadata);
}

table forwarding {
    key = {
        standard_metadata.ingress_port : exact;
    }
    actions = {
        forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
}

register<bit<7>>(1) drop_percent_reg;
register<bit<32>>(1) syn_counts_reg;
register<bit<7>>(1) percent_iterator_reg;

apply {
```

Figure 13. Defining register to store the number dropped SYN packets.

The code above defines a register named `percent_iterator_reg`. The register contains a single cell. This cell is used to track how many packets to drop and to allow out of 100.

Step 5. Add the following code to the *apply* block to retrieve the dropping percentage from the register.

```
if(hdr.tcp.isValid()){
    bit<7> drop_percent;
    drop_percent_reg.read(drop_percent, (bit<32>)0);
}
```



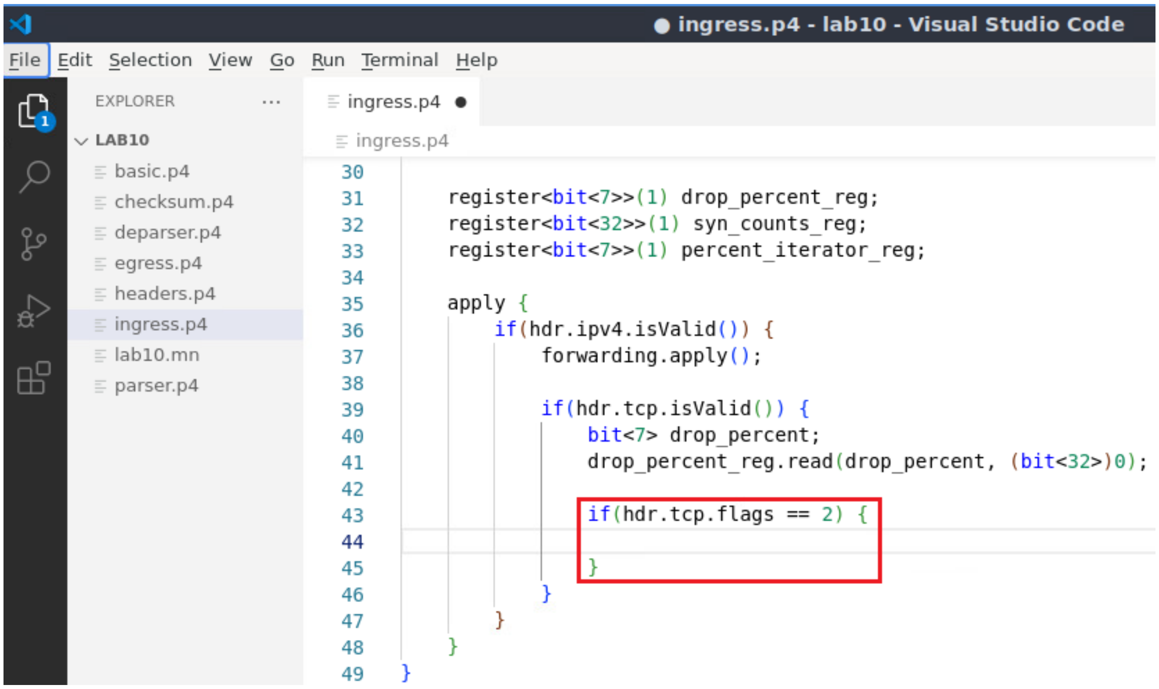
```
                NoAction;
            }
            size = 1024;
            default_action = drop();
        }

        register<bit<7>>(1) drop_percent_reg;
        register<bit<32>>(1) syn_counts_reg;
        register<bit<7>>(1) percent_iterator_reg;

        apply {
            if(hdr.ipv4.isValid()) {
                forwarding.apply();

                if(hdr.tcp.isValid()) {
                    bit<7> drop_percent;
                    drop_percent_reg.read(drop_percent, (bit<32>)0);
                }
            }
        }
    }
```

Figure 14. Retrieving the dropping percentage from the register.

In the code above, `if(hdr.tcp.isValid())` checks if the packet is a TCP packet. For TCP packets, the dropping percentage is retrieved from `drop_percentage_reg` and stored in the `drop_percent` variable. Note that the dropping percentage will be specified from the control plane.

Step 6. Add the following code to check if the incoming packet is a SYN packet.

```
if(hdr.tcp.flags == 2) {
}
```

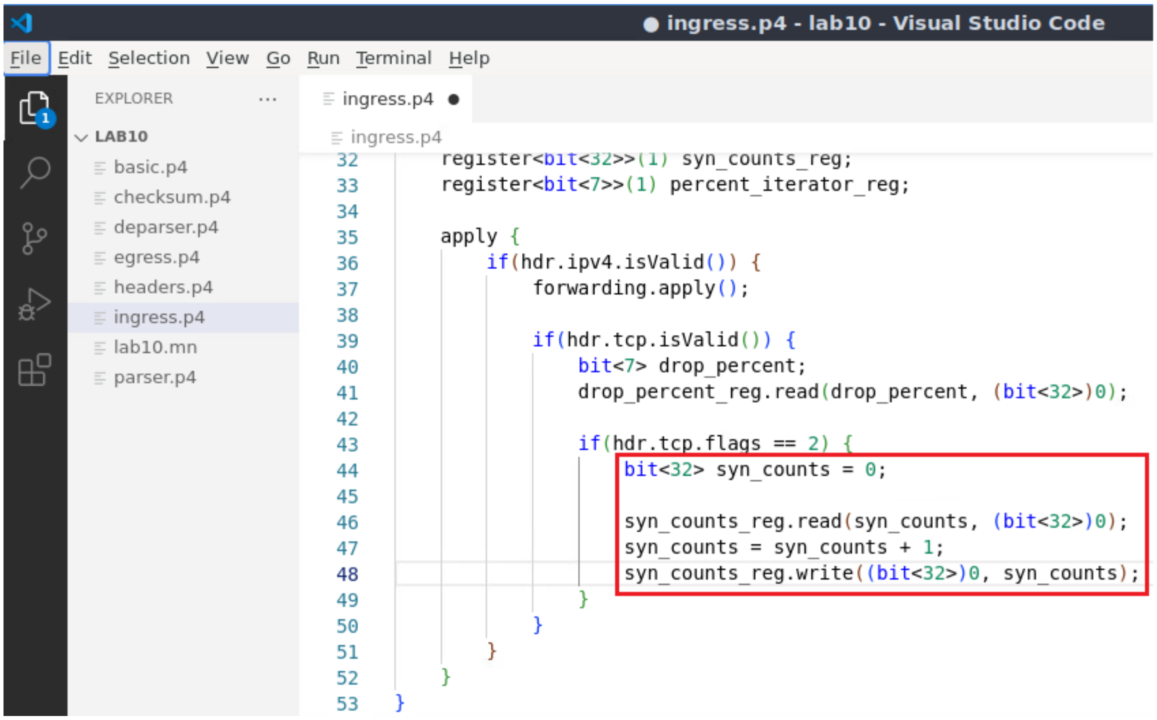


```
    register<bit<7>>(1) drop_percent_reg;
    register<bit<32>>(1) syn_counts_reg;
    register<bit<7>>(1) percent_iterator_reg;

    apply {
        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()) {
                bit<7> drop_percent;
                drop_percent_reg.read(drop_percent, (bit<32>)0);

                if(hdr.tcp.flags == 2) {

                }
            }
        }
    }
}
```

Figure 15. Checking the type of TCP packets.

Step 7. Add the following code to increment the count of SYN packets.

```
bit<32> syn_counts = 0;

syn_counts_reg.read(syn_counts, (bit<32>)0);
syn_counts = syn_counts +1;
syn_counts_reg.write((bit<32>)0, syn_counts);
```



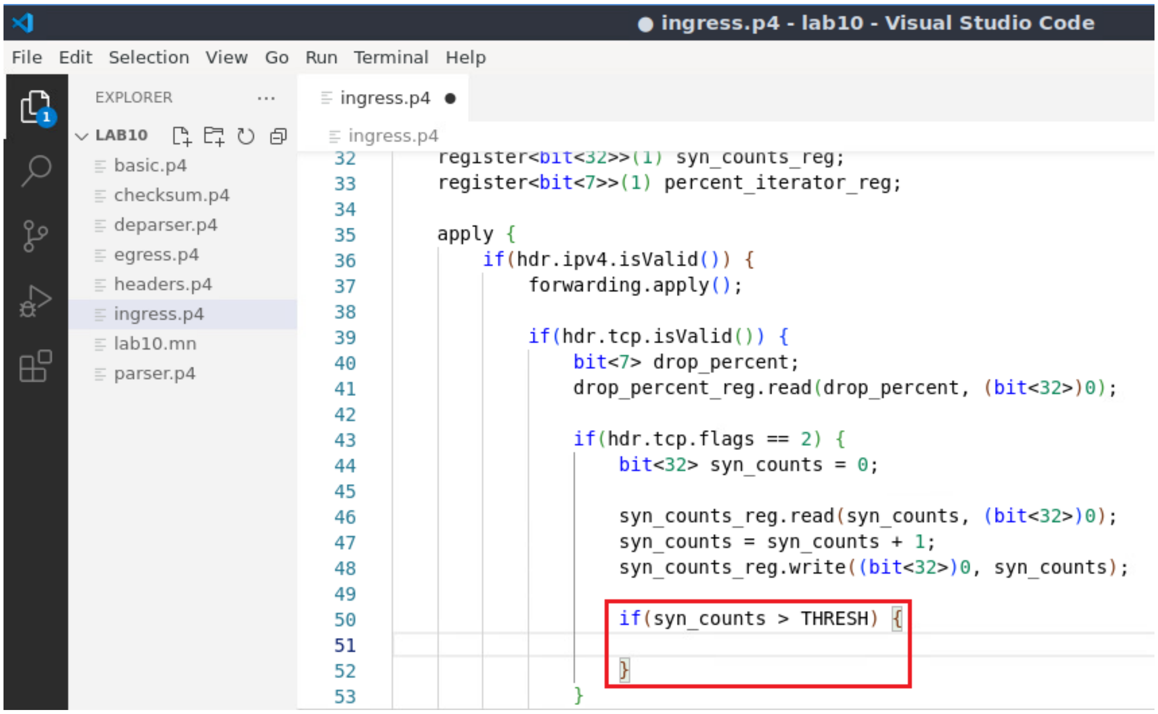
```
    register<bit<32>>(1) syn_counts_reg;
    register<bit<7>>(1) percent_iterator_reg;

    apply {
        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()) {
                bit<7> drop_percent;
                drop_percent_reg.read(drop_percent, (bit<32>)0);

                if(hdr.tcp.flags == 2) {
                    bit<32> syn_counts = 0;

                    syn_counts_reg.read(syn_counts, (bit<32>)0);
                    syn_counts = syn_counts + 1;
                    syn_counts_reg.write((bit<32>)0, syn_counts);
                }
            }
        }
    }
}
```

Figure 16. Incrementing the count of SYN packets.

In the code above, the count of SYN packets is retrieved from the `syn_counts_reg` and stored in the `syn_counts` variable. The variable is incremented by one to account for the current packet. After that, the updated `syn_counts` variable is stored in the `syn_count_reg` register.

Step 8. Add the following code to check if the number of SYN packets exceeded `THRESH`.

```
if(syn_counts > THRESH){

}
```

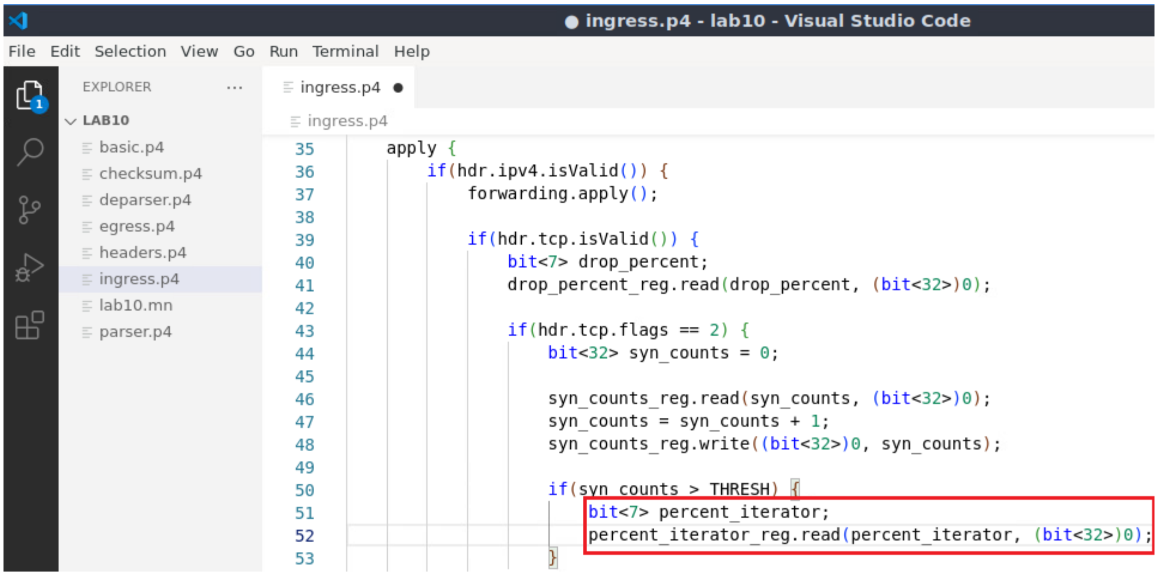


```
    register<bit<32>>(1) syn_counts_reg;
    register<bit<7>>(1) percent_iterator_reg;

    apply {
        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()) {
                bit<7> drop_percent;
                drop_percent_reg.read(drop_percent, (bit<32>)0);

                if(hdr.tcp.flags == 2) {
                    bit<32> syn_counts = 0;

                    syn_counts_reg.read(syn_counts, (bit<32>)0);
                    syn_counts = syn_counts + 1;
                    syn_counts_reg.write((bit<32>)0, syn_counts);

                    if(syn_counts > THRESH) {

                    }
                }
```

Figure 17. Checking the number of SYN packets against the threshold.

Step 9. Add the following code to retrieve the iterator from the `percent_iterator_reg`.

```
bit<7> percent_iterator;
percent_iterator_reg.read(percent_iterator, (bit<32>)0);
```



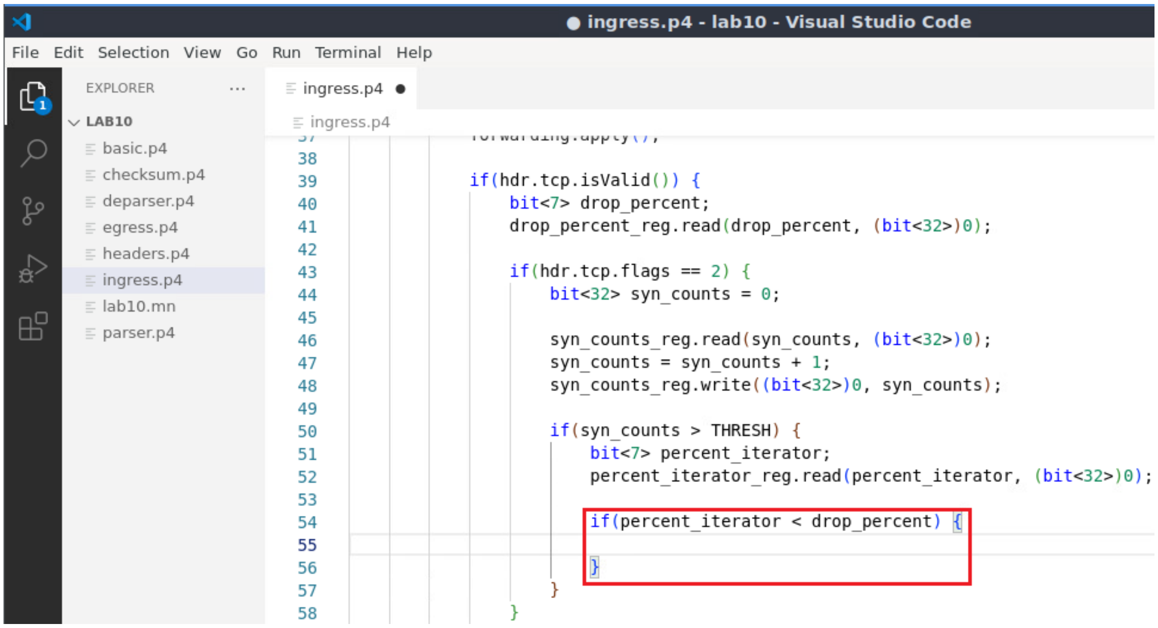
```
apply {
    if(hdr.ipv4.isValid()) {
        forwarding.apply();

        if(hdr.tcp.isValid()) {
            bit<7> drop_percent;
            drop_percent_reg.read(drop_percent, (bit<32>)0);

            if(hdr.tcp.flags == 2) {
                bit<32> syn_counts = 0;

                syn_counts_reg.read(syn_counts, (bit<32>)0);
                syn_counts = syn_counts + 1;
                syn_counts_reg.write((bit<32>)0, syn_counts);

                if(syn_counts > THRESH) {
                    bit<7> percent_iterator;
                    percent_iterator_reg.read(percent_iterator, (bit<32>)0);
                }
```

Figure 18. Retrieving the number of dropped packets from the register.

In the code above, the iterator is retrieved from `percent_iterator_reg` and stored inside the `percent_iterator` variable.

Step 10. Add the following code to check if the iterator is less than the dropping percentage.

```
if(percent_iterator <  drop_percent){

}
```

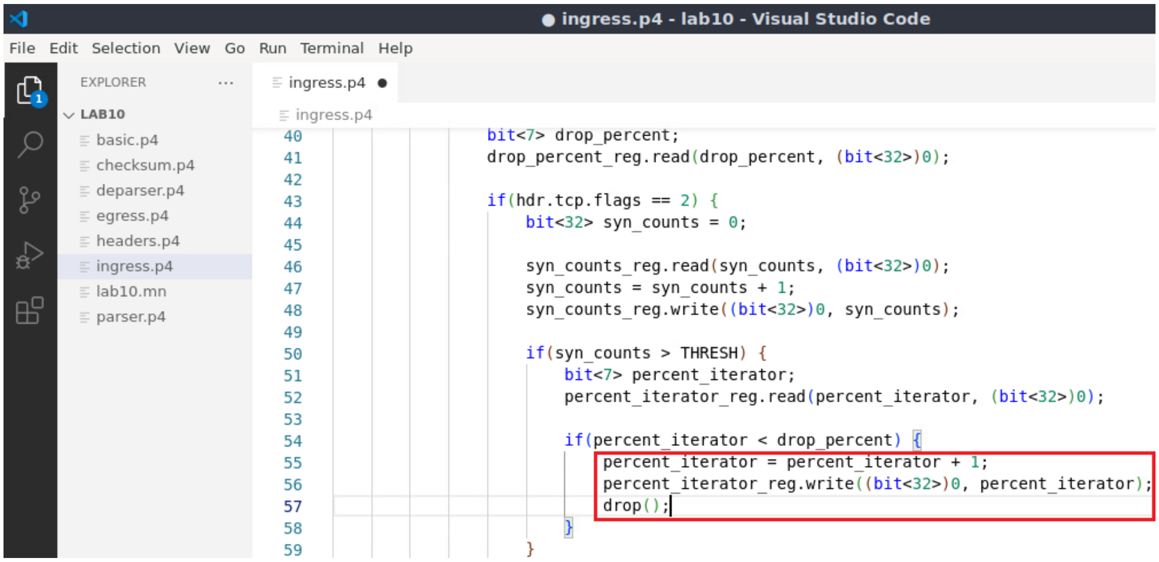


```

        if(hdr.tcp.isValid()) {
            bit<7> drop_percent;
            drop_percent_reg.read(drop_percent, (bit<32>)0);

            if(hdr.tcp.flags == 2) {
                bit<32> syn_counts = 0;

                syn_counts_reg.read(syn_counts, (bit<32>)0);
                syn_counts = syn_counts + 1;
                syn_counts_reg.write((bit<32>)0, syn_counts);

                if(syn_counts > THRESH) {
                    bit<7> percent_iterator;
                    percent_iterator_reg.read(percent_iterator, (bit<32>)0);

                    if(percent_iterator < drop_percent) {

                    }
                }
            }
```

Figure 19. Checking if the number of dropped packets is less than the dropping percentage.

Step 11. Add the following code to drop the packet and increment the iterator if the `percent_iterator` is less than the `drop_percent`.

```
percent_iterator = percent_iterator + 1;
percent_iterator_reg.write((bit<32>)0, percent_iterator);
drop();
```



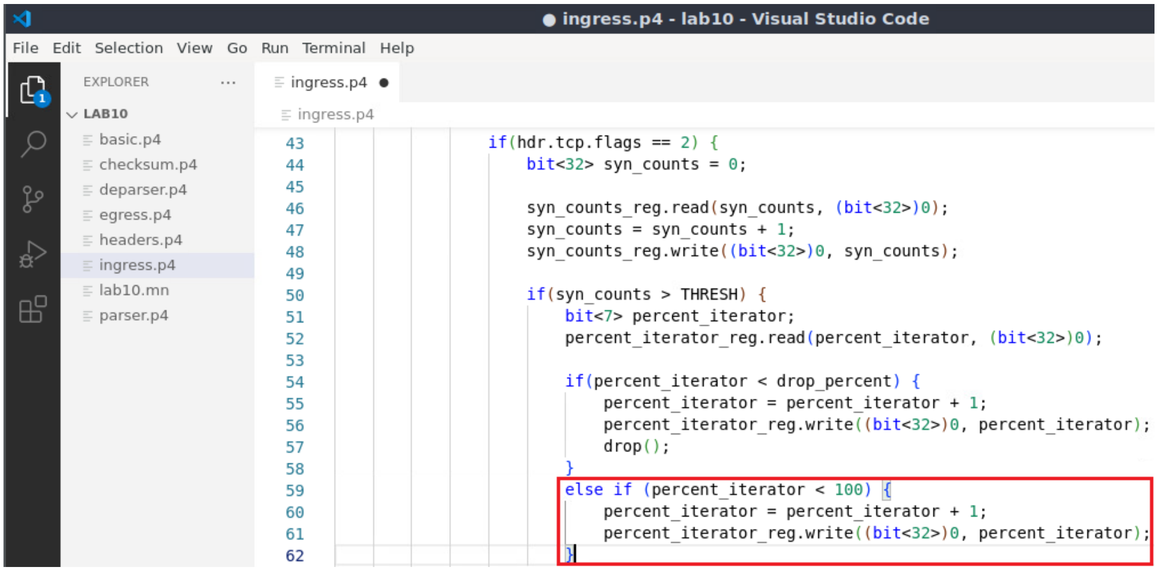
```
                bit<7> drop_percent;
                drop_percent_reg.read(drop_percent, (bit<32>)0);

                if(hdr.tcp.flags == 2) {
                    bit<32> syn_counts = 0;

                    syn_counts_reg.read(syn_counts, (bit<32>)0);
                    syn_counts = syn_counts + 1;
                    syn_counts_reg.write((bit<32>)0, syn_counts);

                    if(syn_counts > THRESH) {
                        bit<7> percent_iterator;
                        percent_iterator_reg.read(percent_iterator, (bit<32>)0);

                        if(percent_iterator < drop_percent) {
                            percent_iterator = percent_iterator + 1;
                            percent_iterator_reg.write((bit<32>)0, percent_iterator);
                            drop();
                        }
                    }
```

Figure 20. Dropping SYN packets.

In the code above, the `percent_iterator` variable is incremented by one and stored in the `percent_iterator_reg` register. After that, the packet is dropped.

Step 12. Add the following code to increment the count of dropped packets by one without dropping the packet if the number of dropped packet is less than 100.

```
else if (percent_iterator < 100) {
    percent_iterator = percent_iterator + 1;
    percent_iterator_reg.write((bit<32>)0, percent_iterator);
}
```



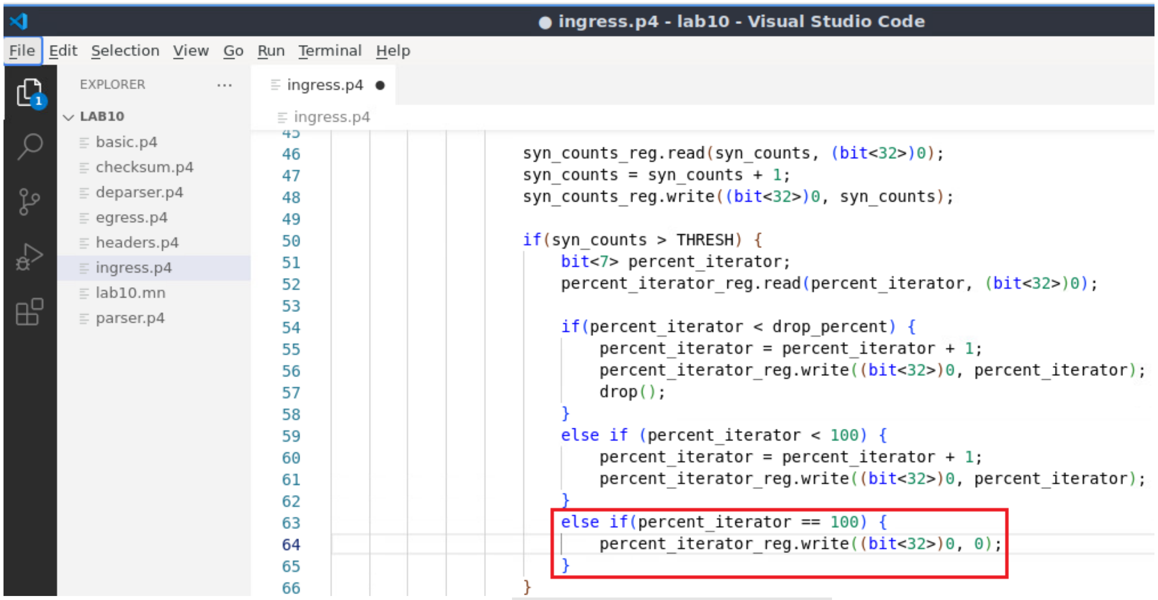
```
                if(hdr.tcp.flags == 2) {
                    bit<32> syn_counts = 0;

                    syn_counts_reg.read(syn_counts, (bit<32>)0);
                    syn_counts = syn_counts + 1;
                    syn_counts_reg.write((bit<32>)0, syn_counts);

                    if(syn_counts > THRESH) {
                        bit<7> percent_iterator;
                        percent_iterator_reg.read(percent_iterator, (bit<32>)0);

                        if(percent_iterator < drop_percent) {
                            percent_iterator = percent_iterator + 1;
                            percent_iterator_reg.write((bit<32>)0, percent_iterator);
                            drop();
                        }
                        else if (percent_iterator < 100) {
                            percent_iterator = percent_iterator + 1;
                            percent_iterator_reg.write((bit<32>)0, percent_iterator);
                        }
```

Figure 21. Incrementing the iterator.

From each 100 packets, we are dropping the first `drop_percent` packets (e.g., the first 50 packets if the `drop_percent` is 50%). The remaining packets (i.e., 100 – `drop_percent`) are forwarded.

Step 13. Add the following code to reset `percent_iterator_reg` register when `percent_iterator` reaches 100.

```
else if (percent_iterator == 100) {
   percent_iterator_reg.write((bit<32>)0, 0);
}
```



```
syn_counts_reg.read(syn_counts, (bit<32>)0);
syn_counts = syn_counts + 1;
syn_counts_reg.write((bit<32>)0, syn_counts);

if(syn_counts > THRESH) {
    bit<7> percent_iterator;
    percent_iterator_reg.read(percent_iterator, (bit<32>)0);

    if(percent_iterator < drop_percent) {
        percent_iterator = percent_iterator + 1;
        percent_iterator_reg.write((bit<32>)0, percent_iterator);
        drop();
    }
    else if (percent_iterator < 100) {
        percent_iterator = percent_iterator + 1;
        percent_iterator_reg.write((bit<32>)0, percent_iterator);
    }
    else if(percent_iterator == 100) {
        percent_iterator_reg.write((bit<32>)0, 0);
    }
}
```

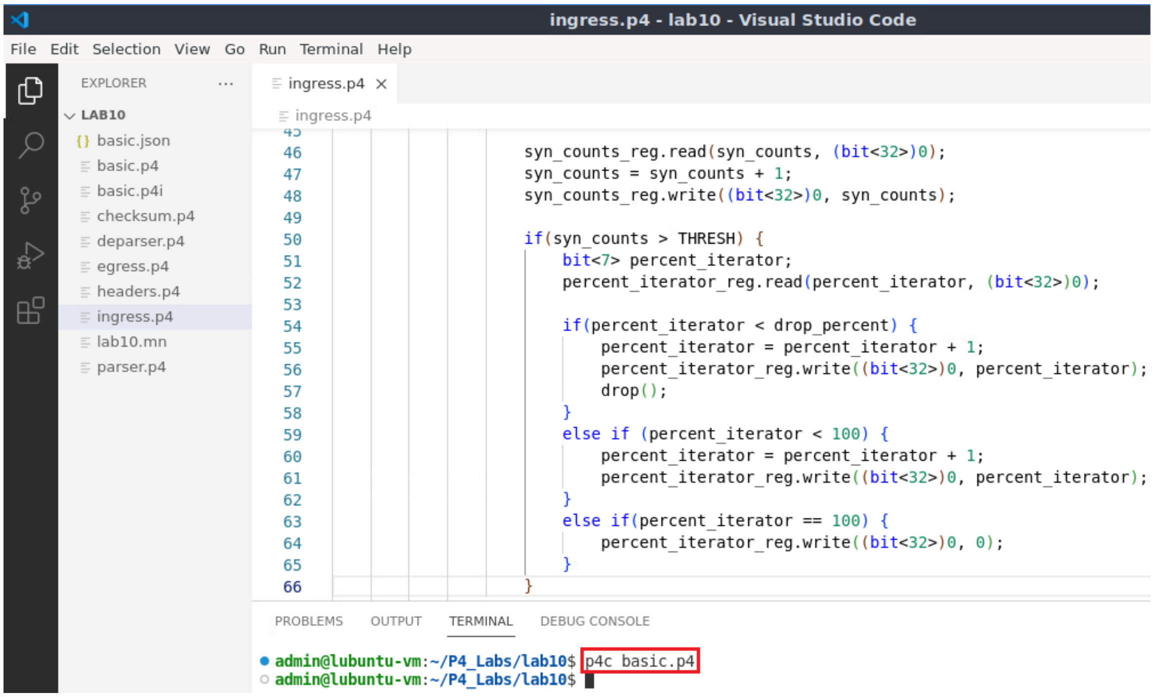
Figure 22. Resetting the `percent_iterator_reg` register.

Step 14. Save the changes to the file by pressing `Ctrl + s`.

3.3 Loading the P4 program

Step 1. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```


ingress.p4 - lab10 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB10
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab10.mn
parser.p4
ingress.p4
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

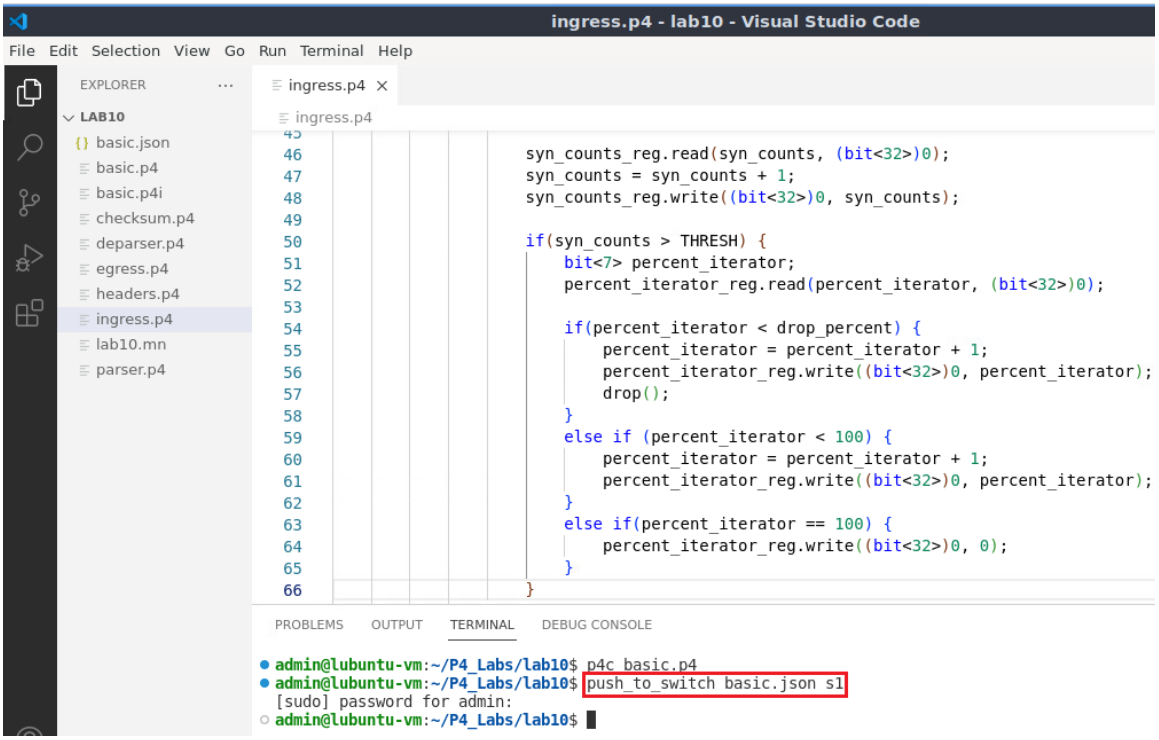
```
syn_counts_reg.read(syn_counts, (bit<32>)0);
syn_counts = syn_counts + 1;
syn_counts_reg.write((bit<32>)0, syn_counts);

if(syn_counts > THRESH) {
    bit<7> percent_iterator;
    percent_iterator_reg.read(percent_iterator, (bit<32>)0);

    if(percent_iterator < drop_percent) {
        percent_iterator = percent_iterator + 1;
        percent_iterator_reg.write((bit<32>)0, percent_iterator);
        drop();
    }
    else if (percent_iterator < 100) {
        percent_iterator = percent_iterator + 1;
        percent_iterator_reg.write((bit<32>)0, percent_iterator);
    }
    else if(percent_iterator == 100) {
        percent_iterator_reg.write((bit<32>)0, 0);
    }
}
```

```
admin@lubuntu-vm:~/P4_Labs/lab10$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab10$
```

Figure 23. Compiling the P4 program using the VS Code terminal.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```

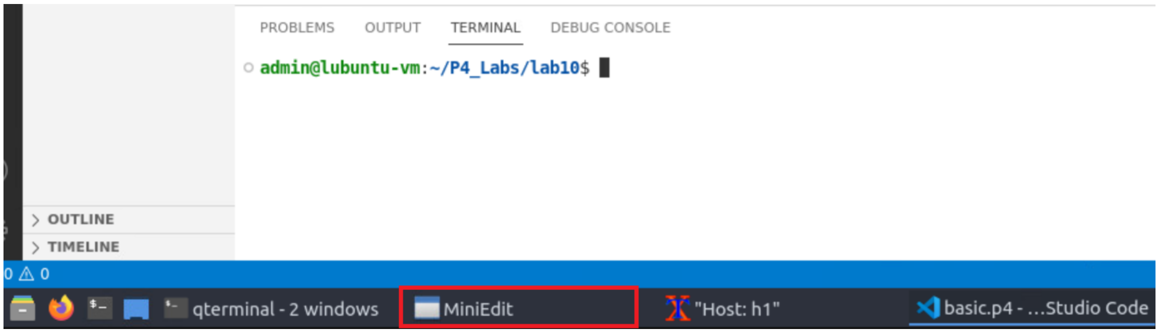

ingress.p4 - lab10 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB10
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab10.mn
parser.p4
ingress.p4
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
syn_counts_reg.read(syn_counts, (bit<32>)0);
syn_counts = syn_counts + 1;
syn_counts_reg.write((bit<32>)0, syn_counts);

if(syn_counts > THRESH) {
    bit<7> percent_iterator;
    percent_iterator_reg.read(percent_iterator, (bit<32>)0);

    if(percent_iterator < drop_percent) {
        percent_iterator = percent_iterator + 1;
        percent_iterator_reg.write((bit<32>)0, percent_iterator);
        drop();
    }
    else if (percent_iterator < 100) {
        percent_iterator = percent_iterator + 1;
        percent_iterator_reg.write((bit<32>)0, percent_iterator);
    }
    else if(percent_iterator == 100) {
        percent_iterator_reg.write((bit<32>)0, 0);
    }
}
```

```
admin@lubuntu-vm:~/P4_Labs/lab10$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab10$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab10$
```

Figure 24. Downloading the compiled program to switch s1.

Step 3. Click on the MinEdit tab in the start bar to maximize the window.



```
admin@lubuntu-vm:~/P4_Labs/lab10$
```

Figure 25. Maximizing the MiniEdit window.

Step 4. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

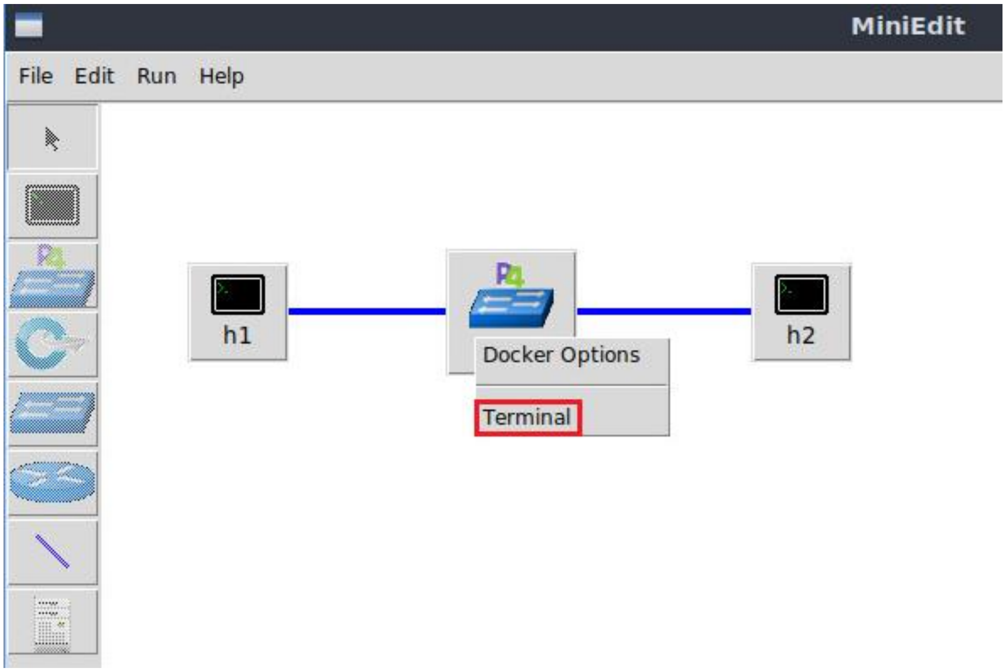
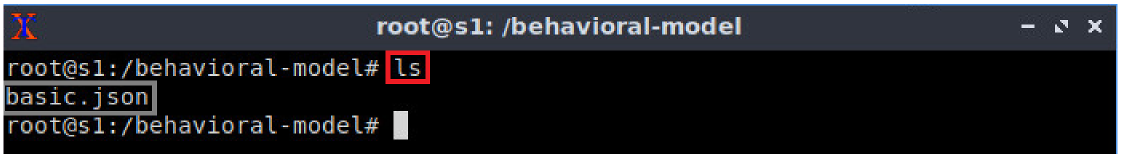


Figure 26. Starting the terminal on switch s1.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch's terminal.

Step 5. Issue the following command to list the files in the current directory.

```
ls
```



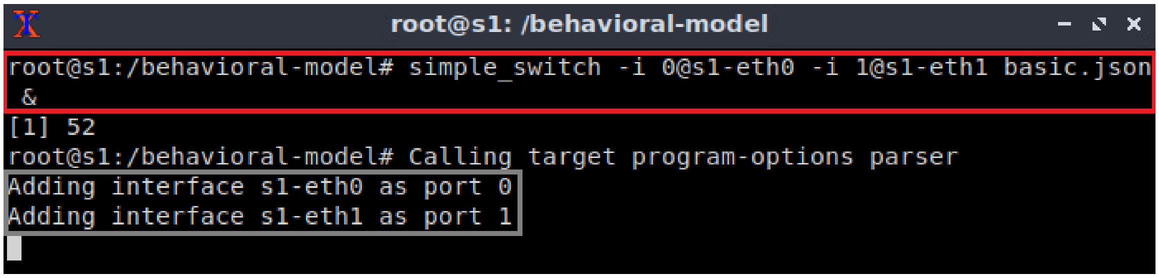
```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model#
```

Figure 27. Displaying the contents of the current directory in the switch s1.

We can see that the switch contains the *basic.json* file that was downloaded after compiling the P4 program.

Step 6. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 52
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
```

Figure 28. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

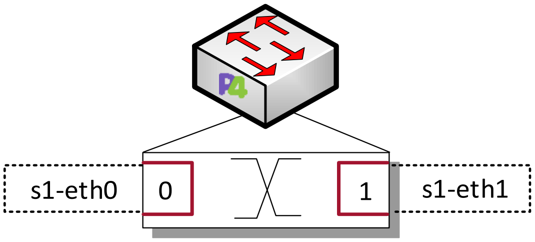
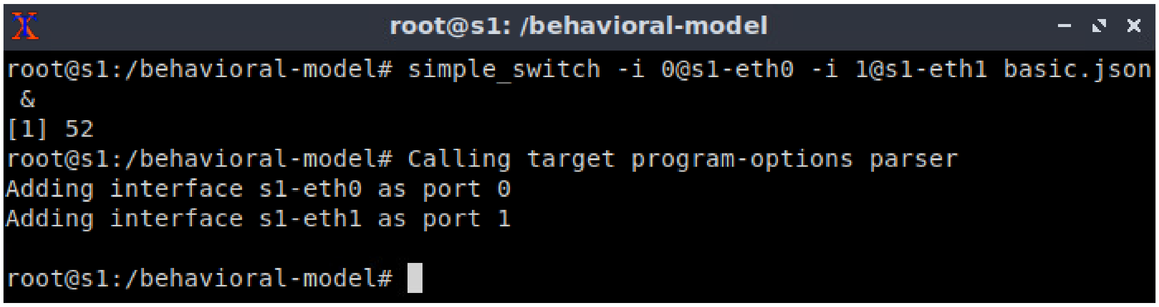


Figure 29. Ports 0 and 1 are mapped to the interfaces *s1-eth0* and *s1-eth1* of switch s1.

Step 7. In switch s1 terminal, press *Enter* to return the CLI.

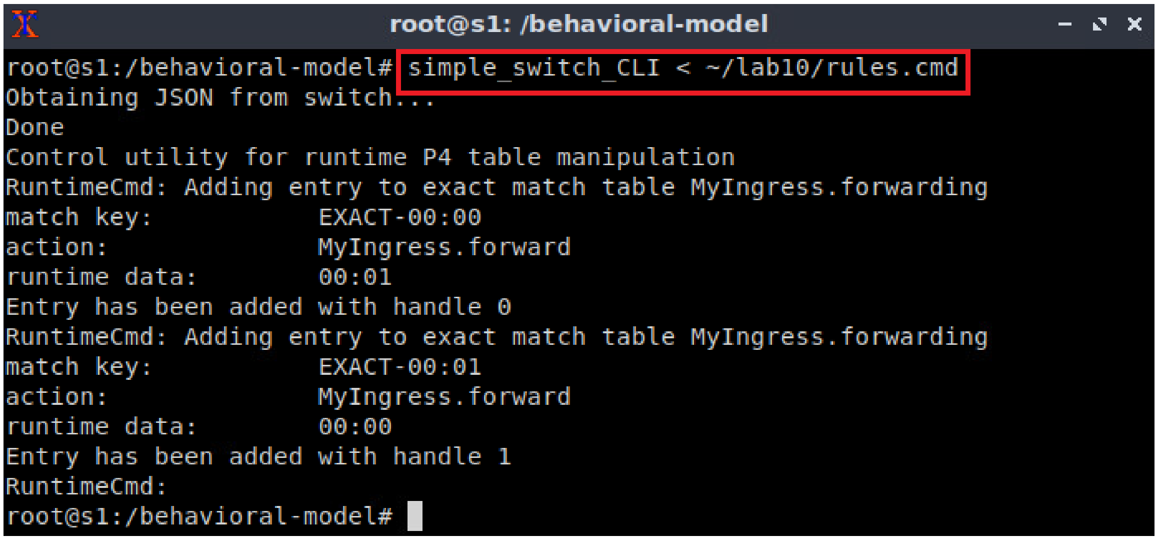


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 basic.json
 &
[1] 52
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1

root@s1:/behavioral-model# 
```

Figure 30. Returning to switch s1 CLI.

Step 8. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab10/rules.cmd
```



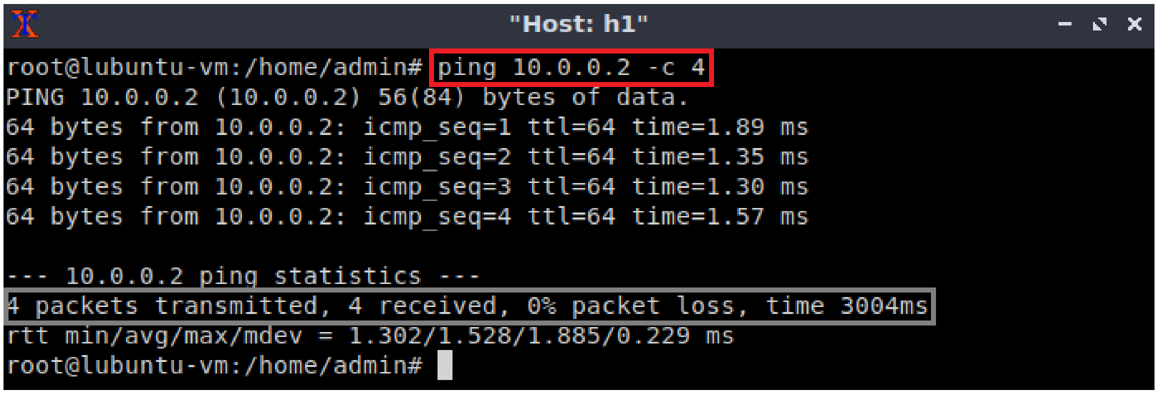
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab10/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:00
action:              MyIngress.forward
runtime data:        00:01
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-00:01
action:              MyIngress.forward
runtime data:        00:00
Entry has been added with handle 1
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 31. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

Step 9. Go back to host h1 terminal to test the connectivity between host h1 and host h2 by issuing the following command.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.89 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.35 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.30 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=1.57 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.302/1.528/1.885/0.229 ms
root@lubuntu-vm:/home/admin#
```

Figure 32. Performing a connectivity test between host h1 and host h2.

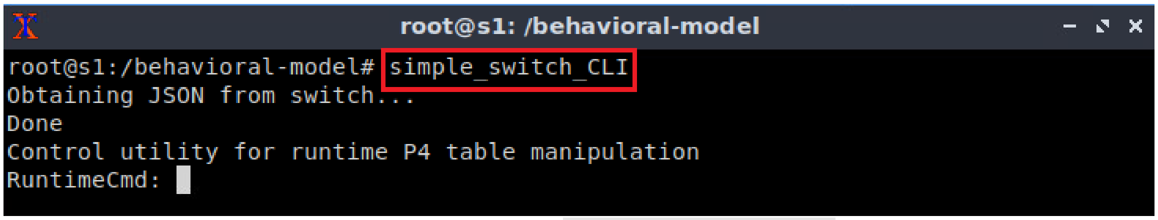
The figure above shows that there is connectivity between the two hosts.

4 Testing the P4 code against SYN flood attack

4.1 Configuring the mitigation parameters

Step 1. In switch s1 terminal, access the `simple_switch_CLI` by typing the command below.

```
simple_switch_CLI
```

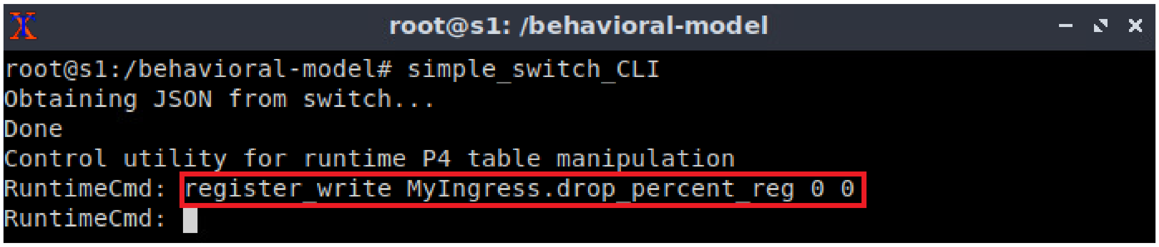


```
root@s1:/behavioral-model# simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd:
```

Figure 33. Accessing `simple_switch_CLI`.

Step 2. Configure the dropping rate to be 0% by typing the command below.

```
register_write MyIngress.drop_percent_reg 0 0
```



```
root@s1:/behavioral-model# simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: register_write MyIngress.drop_percent_reg 0 0
RuntimeCmd:
```

Figure 34. Configuring the dropping rate.

By setting the dropping percentage to 0%, we are disabling the mitigation phase.

Note that the register value in P4 is 0 by default[5]. In the previous step, the value was set to 0 to explicitly show the user that the dropping rate is 0% and that the mitigation phase of the program is disabled.

Step 3. Start a second terminal on s1 by right-clicking on the P4 switch icon in MiniEdit and select *Terminal*.

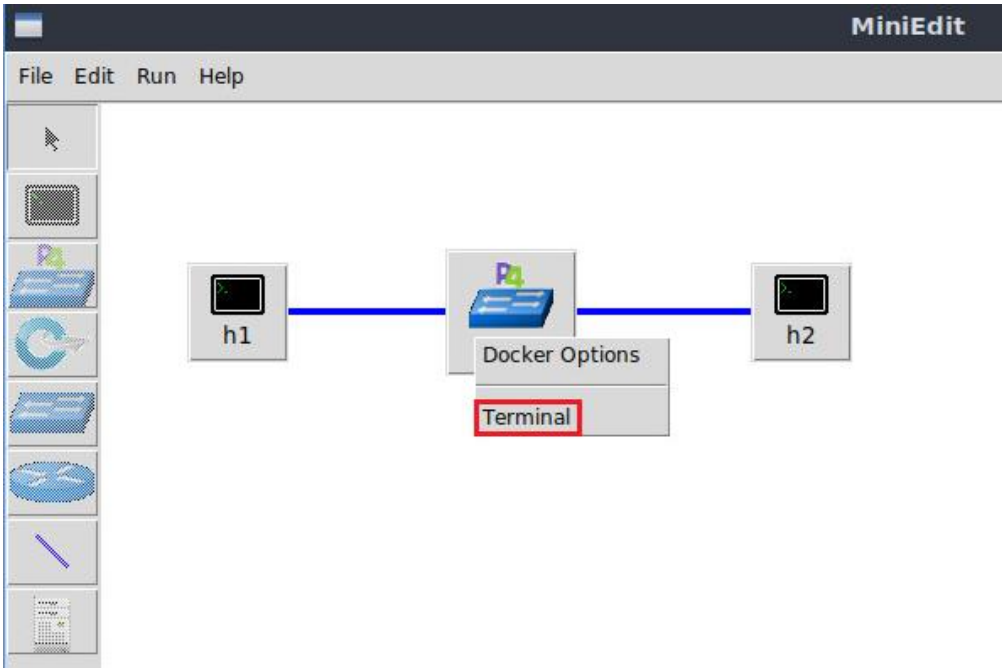
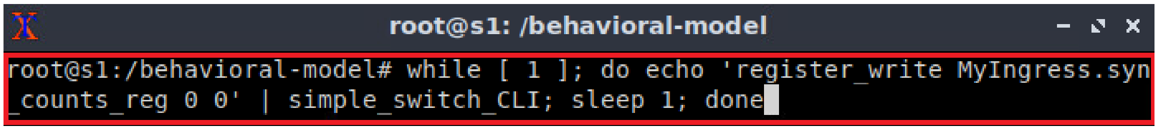


Figure 35. Starting the terminal on switch s1.

Step 4. On the new terminal, type the command below to reset the number of counts every second.

```
while [ 1 ]; do echo 'register_write MyIngress.syn_counts_reg 0 0' |
simple_switch_CLI; sleep 1; done
```



```
root@s1:/behavioral-model# while [ 1 ]; do echo 'register_write MyIngress.syn
_counts_reg 0 0' | simple_switch_CLI; sleep 1; done
```

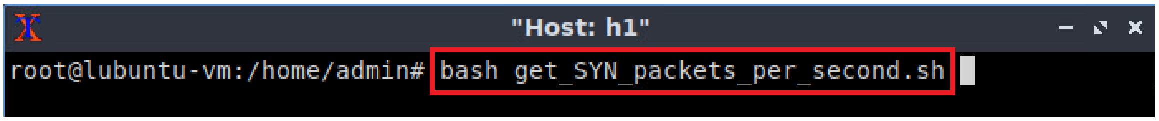
Figure 36. Resetting the dropping rate every 1 second

Note that the spaces in the previous command are mandatory. If you write while [1] instead of while [1], the command will produce an error.

4.2 Performing SYN flood attack

Step 1. On h1 terminal, type the command below to display number of received SYN packets per seconds.

```
bash get_SYN_packets_per_second.sh
```



```
root@lubuntu-vm:/home/admin# bash get_SYN_packets_per_second.sh
```

Figure 37. Displaying the number of received SYN packets per second.

Step 2. Hold the right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

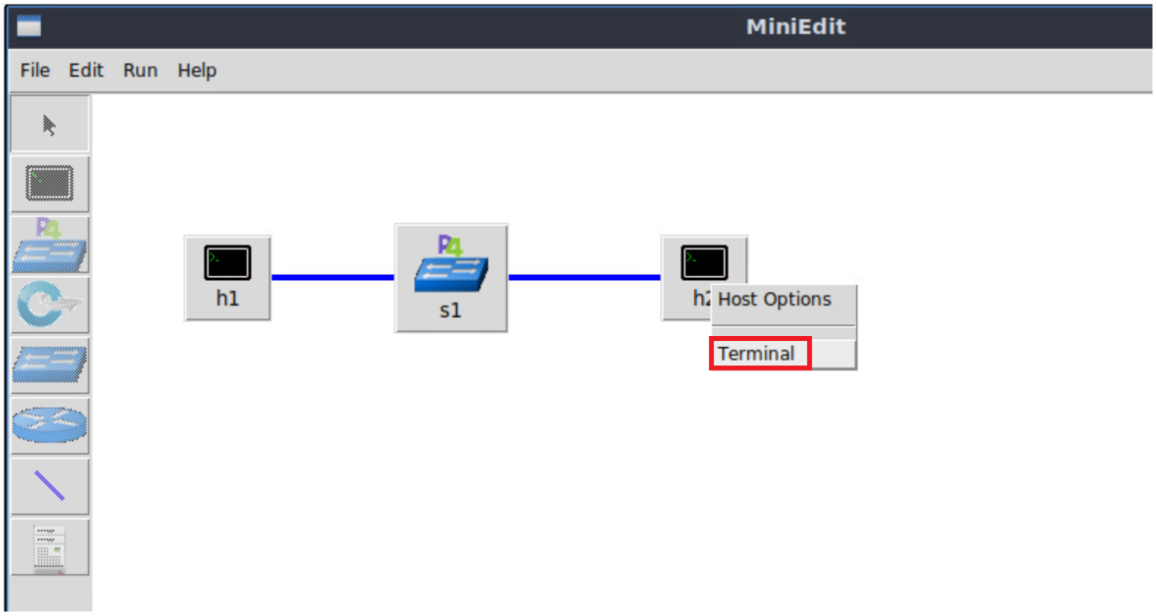
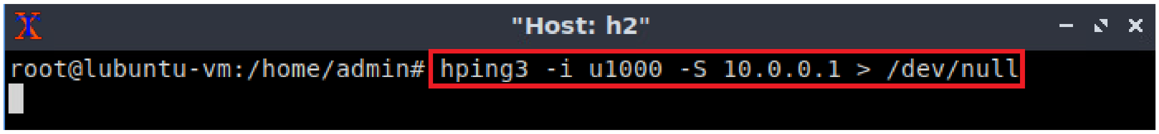


Figure 38. Opening a terminal on host h2.

Step 3. On h2 terminal, type the command below to perform SYN flood attack. `hping3` is a network tool able to send custom TCP/IP packets and to display target replies. `-i u1000` instructs hping3 to send 1000 packets per second. `-S` option sets SYN TCP flag for the packets to be sent. `10.0.0.1` is the destination IP of the packets generated by hping3. `> /dev/null` direct the output of the command to the null device.

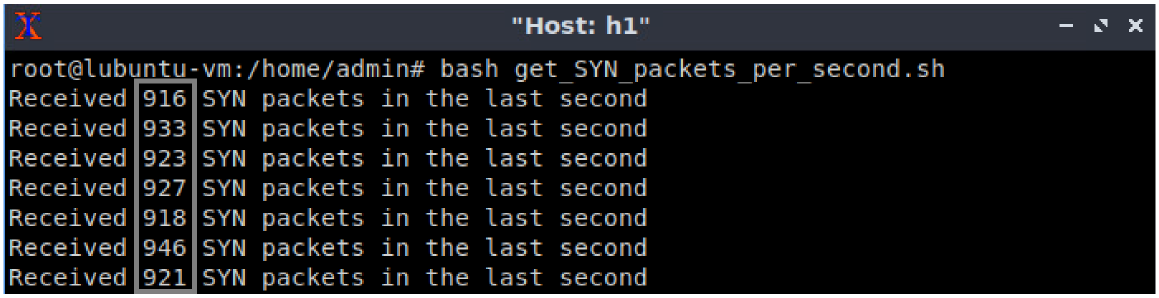
```
hping3 -i u1000 -S 10.0.0.1 > /dev/null
```



```
root@lubuntu-vm:/home/admin# hping3 -i u1000 -S 10.0.0.1 > /dev/null
```

Figure 39. Performing the SYN flood attack.

Step 4. Inspect the number of received SYN packets at h1.



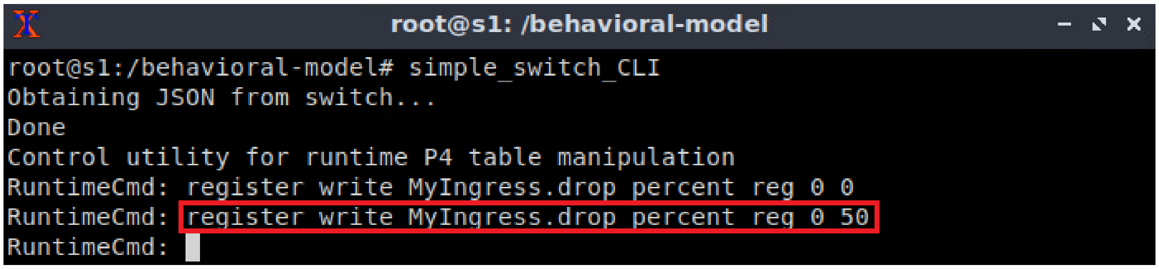
```
root@lubuntu-vm:/home/admin# bash get_SYN_packets_per_second.sh
Received 916 SYN packets in the last second
Received 933 SYN packets in the last second
Received 923 SYN packets in the last second
Received 927 SYN packets in the last second
Received 918 SYN packets in the last second
Received 946 SYN packets in the last second
Received 921 SYN packets in the last second
```

Figure 40. Inspecting the number of received SYN packets at h1.

The figure above shows that h1 is receiving around 920 SYN packets per second. Not that no packets are dropped by the switch because the dropping percentage is set to zero.

Step 5. On s1, use the terminal running the `simple_switch_CLI` to configure the dropping rate to be 50% by typing the command below.

```
register_write MyIngress.drop_percent_reg 0 50
```

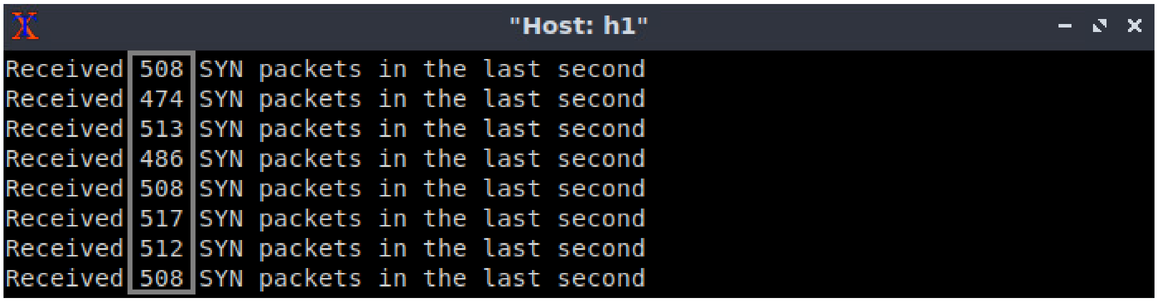


```
root@s1:/behavioral-model# simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: register_write MyIngress.drop_percent_reg 0 0
RuntimeCmd: register_write MyIngress.drop_percent_reg 0 50
RuntimeCmd: 
```

Figure 41. Configuring the dropping rate.

Because the received number of SYN packets per second is around 900, and the threshold is 100, the dropping threshold will be applied on 800 packets only (900 - 100). Note that the switch does not apply the dropping mechanism on the first 100 SYN packets. By setting the dropping percentage to 50%, we expect to receive 100 + 800/2 SYN packets, which is around 500 packets.

Step 6. Inspect the number of received SYN packets at h1.



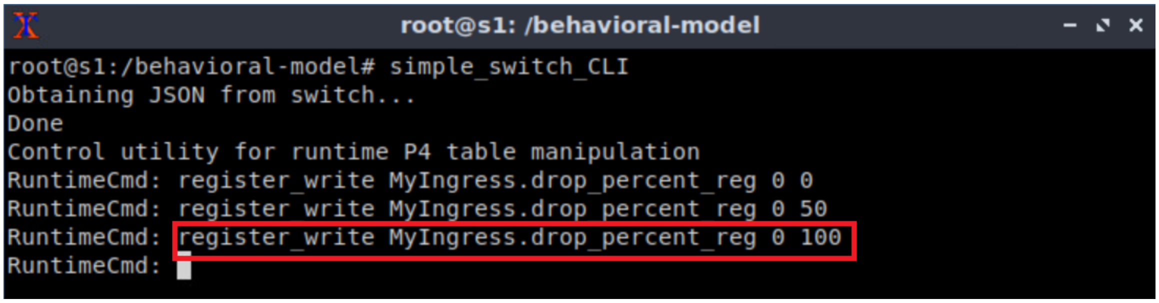
```
Received 508 SYN packets in the last second
Received 474 SYN packets in the last second
Received 513 SYN packets in the last second
Received 486 SYN packets in the last second
Received 508 SYN packets in the last second
Received 517 SYN packets in the last second
Received 512 SYN packets in the last second
Received 508 SYN packets in the last second
```

Figure 42. Inspecting the number of received SYN packets at h1.

The figure above shows that h1 is receiving around 500 SYN packets per second.

Step 7. Configure the dropping rate to be 100% by typing the command below.

```
register_write MyIngress.drop_percent_reg 0 100
```

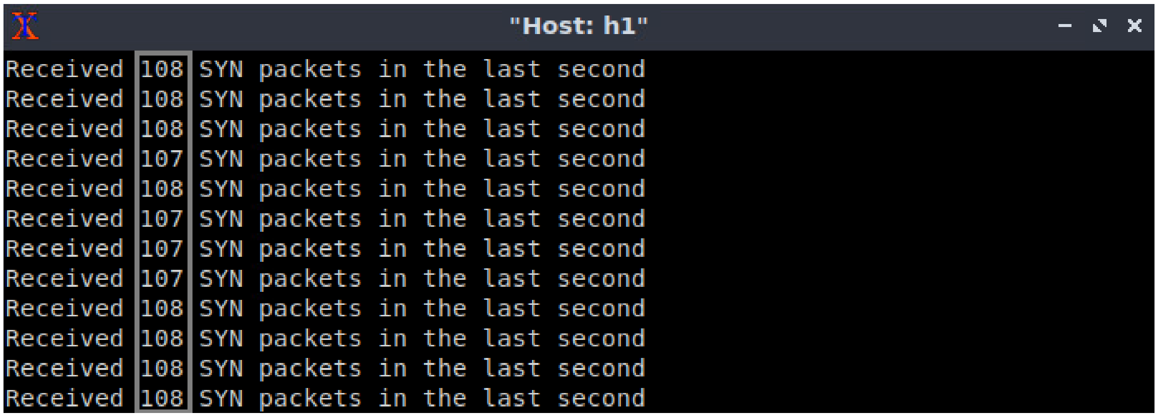


```
root@s1:/behavioral-model# simple_switch_CLI
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: register_write MyIngress.drop_percent_reg 0 0
RuntimeCmd: register_write MyIngress.drop_percent_reg 0 50
RuntimeCmd: register_write MyIngress.drop_percent_reg 0 100
RuntimeCmd: 
```

Figure 43. Configuring the dropping rate.

By setting the dropping threshold to 100%, the expected number of SYN packets to be received per second is around 100 because all the packets above the threshold will be dropped.

Step 8. Inspect the number of received SYN packets at h1.



```
Received 108 SYN packets in the last second
Received 108 SYN packets in the last second
Received 108 SYN packets in the last second
Received 107 SYN packets in the last second
Received 108 SYN packets in the last second
Received 107 SYN packets in the last second
Received 107 SYN packets in the last second
Received 107 SYN packets in the last second
Received 108 SYN packets in the last second
Received 108 SYN packets in the last second
Received 108 SYN packets in the last second
Received 108 SYN packets in the last second
```

Figure 44. Inspecting the number of received SYN packets at h1.

The figure above shows that h1 is receiving around 100 SYN packets per second.

This concludes lab 10. Stop the emulation and then exit out of MiniEdit.

References

1. NETSCOUT, “What is a Volumetric Attack?” [Online]. Available: https://tinyurl.com/4fcehbrb
2. Cloudflare, “SYN Flood Attack.” [Online]. Available: https://tinyurl.com/bdeef2uv
3. GURU99, “What is TCP Three-Way HandShake?.” [Online]. Available: https://tinyurl.com/bdhnd4xu
4. NETSCOUT, “What is a SYN flood attack and how do you to prevent it?” [Online]. Available: https://tinyurl.com/584ufywk
5. P4lang, “[PSA] meter and register initial state.” [Online]. Available: https://tinyurl.com/2s4zey2y

CYBERSECURITY APPLICATIONS ON P4 PROGRAMMABLE DATA PLANES

Lab 11: Blocking Application Layer Slow DDoS Attack (Slowloris)

Document Version: 04-20-2023

Contents

Overview

This lab introduces the slow DDoS Attack (SlowLoris) and provides the steps to implement a P4 program to mitigate the attack. In SlowLoris, the attacker occupies the resources of a web server by maintaining multiple simultaneous TCP connections, such that the attacker just sends enough packets for each connection to prevent it from terminating due to timeout. To mitigate this attack, the user will limit the number of TCP connections per IP address. In the P4 program, the user will define a register to track the number of TCP connections and will use the source IP address of the packets as the index of the register. If the count of connections for a specific IP address exceeds a predefined threshold, the switch will drop all new connections coming from that IP address.

Objectives

By the end of this lab, students should be able to:

1. Define the slow DDoS Attack.
2. Understand the workflow of the slow DDoS attack.
3. Perform a slow DDoS attack.
4. Write a P4 program that mitigates the slow DDoS attack.

Lab settings

Table 1 contains the credentials of the virtual machine used for this lab.

Table 1**.** Credentials to access Client machine.

Device	Account	Password
Client	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction.
2. Section 2: Lab topology.
3. Section 3: Loading a basic P4 program.
4. Section 4: Performing slow DDoS attack.
5. Section 5: Modifying the P4 program to mitigate slow DDoS attack.

1 Introduction

SlowLoris is a type of low-bandwidth, application-layer denial-of-service (DoS) attack that targets web servers by exploiting their connections and resources[1]. The attack targets servers using HTTP, making it highly effective against web servers that do not have proper protection mechanisms in place.

The primary objective of the SlowLoris attack is to exhaust the server's available connections, rendering it unable to serve legitimate requests. Unlike conventional DoS attacks that involve flooding the target with massive amounts of data, SlowLoris operates discreetly by opening multiple connections to the target server and maintaining them for an extended period. It does this by sending HTTP requests in a slow, fragmented manner, using partial request headers, and deliberately prolonging the completion of these requests.

The attacker keeps these connections alive by periodically sending additional headers or whitespace, without ever completing the request. Since most web servers have a limit on the number of concurrent connections they can handle, SlowLoris eventually causes the server to reach its connection limit, preventing it from accepting new, legitimate connections. Consequently, the server becomes unresponsive or significantly slowed down, affecting its ability to serve content to users.

Due to its stealthy nature and low bandwidth consumption, SlowLoris can be difficult to detect and mitigate. However, various countermeasures can help defend against this type of attack, including limiting the number of connections from a single IP address, adjusting server timeouts, employing reverse proxies, or using load balancers. Implementing these defenses can help improve a web server's resilience against SlowLoris and other similar attacks[3].

1.1 Lab scenario

In this lab, a P4 programmable switch will mitigate the SlowLoris attack by forcing clients to have a limited number of ongoing connections with the HTTP server. The switch tracks the number of flows per client by using registers that store the count of ongoing flows per IP address. The hash of the source IP is used as the index to the registers. The counters of the registers are incremented when a new SYN packet is received and are decremented when a FIN packet is received. If the register value exceeds a predefined threshold (i.e., a client has the maximum allowed number of connections), no new connections from the client will be allowed before terminating an existing one.

Consider Figure 1. The topology consists of a legitimate user, an attacker, a web server, and a P4 switch. The attacker performs a SlowLoris attack to occupy all the available connections at the web server and consequently makes it unavailable. The P4 switch mitigates the attack by dropping the connections initiated by the attacker when the number of ongoing connections exceeds the predefined threshold. The legitimate user will be able to communicate with the web server as the switch prevents the resources of the web server from being occupied by the SlowLoris attack.

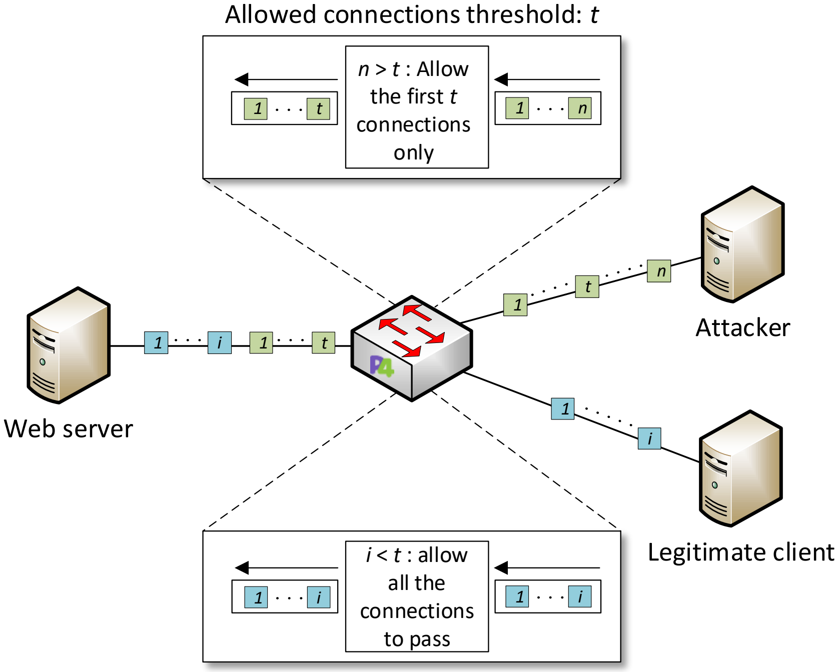


Figure 1. SlowLoris attack mitigation using P4 switch.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

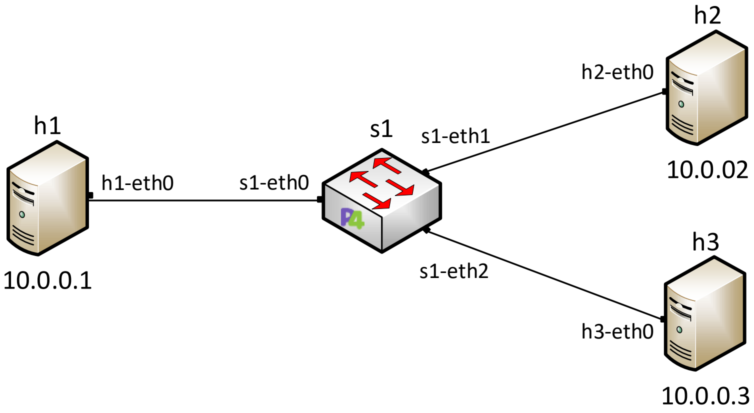


Figure 2. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's desktop. Start MiniEdit by double-clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

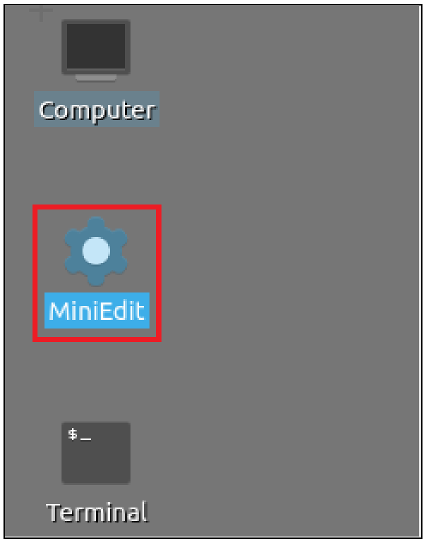
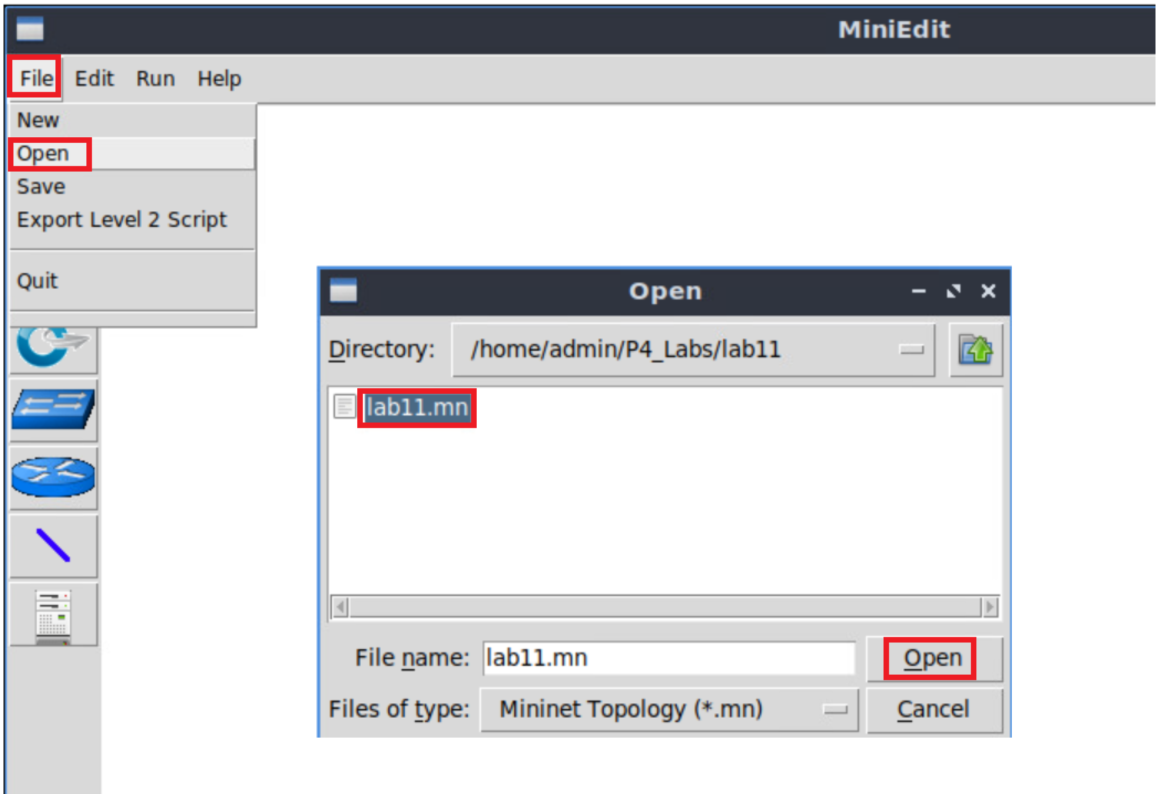


Figure 3. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. A window will emerge. Open the folder called *lab11*, select the file *lab11.mn,* and click on *Open*.



Directory: /home/admin/P4_Labs/lab11

lab11.mn

File name: lab11.mn

Files of type: Mininet Topology (*.mn)

Figure 4. Opening a topology in MiniEdit.

Step 3. The network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

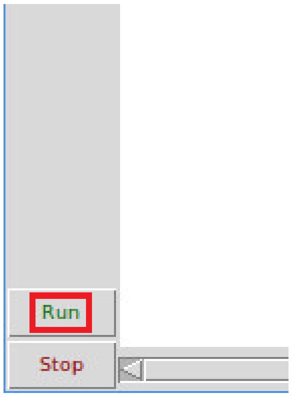


Figure 5. Running the emulation.

2.1 Verifying connectivity between host h1 and host h2

Step 1. Hold the right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

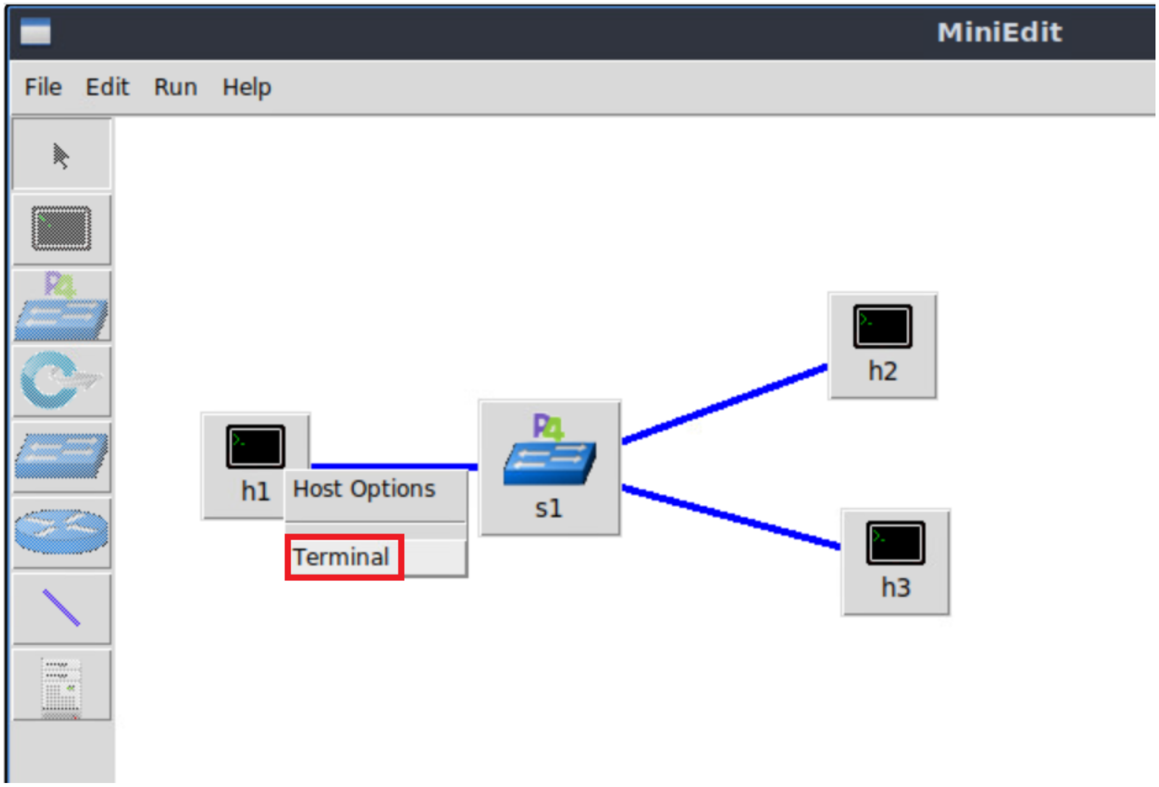
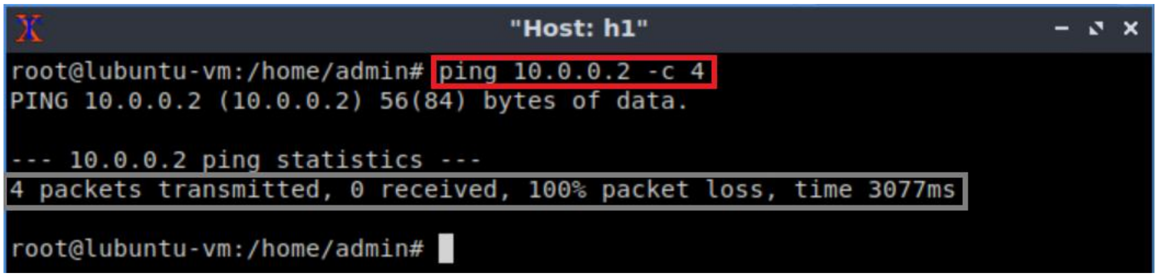


Figure 6. Opening a terminal on host h1.

Step 2. Test the connectivity between host h1 and host h2 by issuing the command below.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3077ms

root@lubuntu-vm:/home/admin#
```

Figure 7. Performing a connectivity test between host h1 and host h2.

The figure above indicates no connectivity between host h1 and host h2 because there is no program loaded into the switch. Note that there will be no connectivity between any two hosts in the topology before loading a P4 program to the switch.

3 Loading the basic P4 program

In this section, the user will compile and run a P4 program that implements the basic forwarding functionality. The switch will then be configured by mapping the P4 program's ports and loading the rules to the switch.

Step 1. Launch a Linux terminal by double-clicking on the Linux terminal icon located on the desktop. Alternatively, click on the Linux terminal icon located on the lower left-hand side.

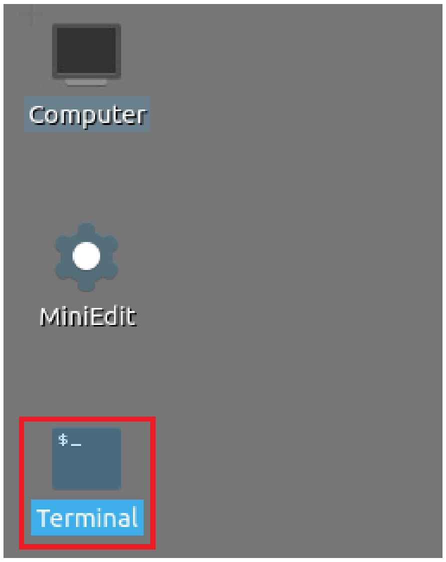
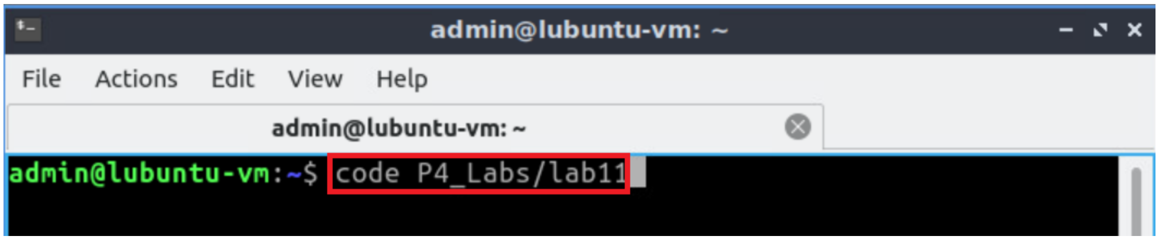


Figure 8. Shortcut to open a Linux terminal.

Step 2. In the terminal, type the command below. This command launches the VS Code and opens the directory where the P4 program for this lab is located.

```
code P4_Labs/lab11
```

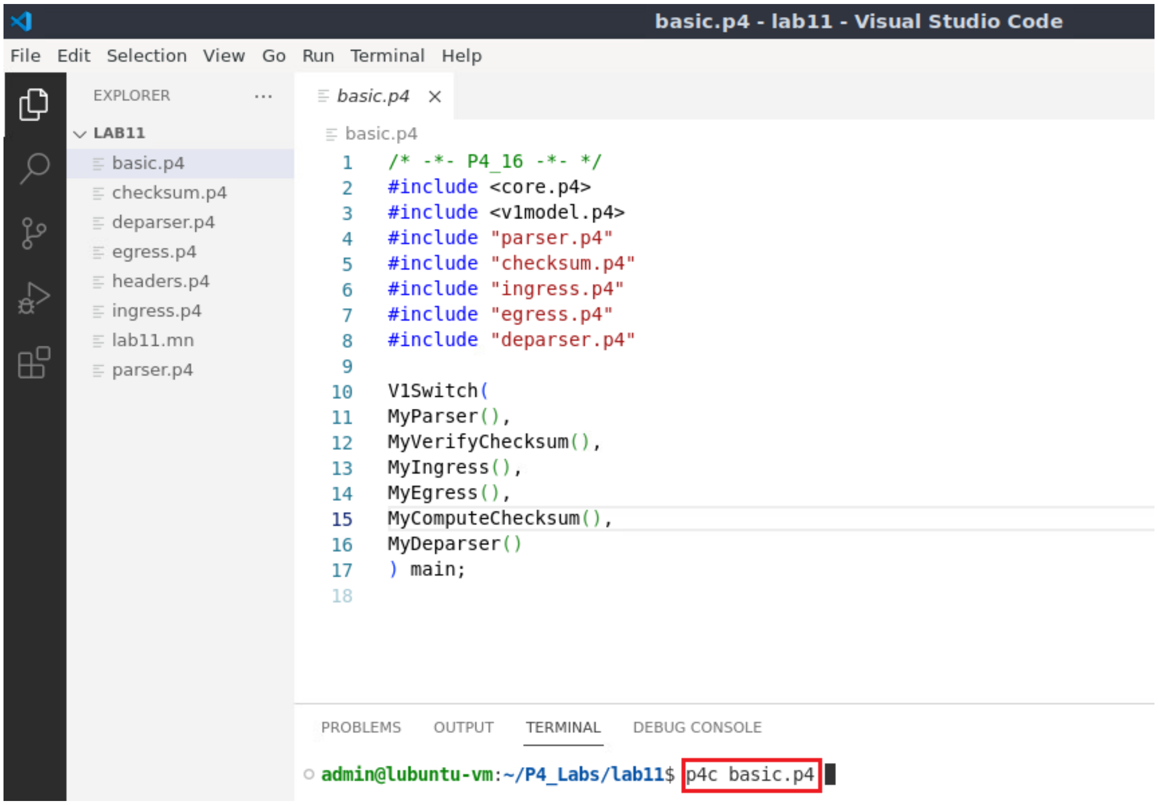


```
admin@lubuntu-vm:~$ code P4_Labs/lab11
```

Figure 9. Launching the editor and opening the lab11 directory.

Step 3. In this lab, we will not modify the P4 code. Instead, we will just compile it and download it to the switch s1. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```



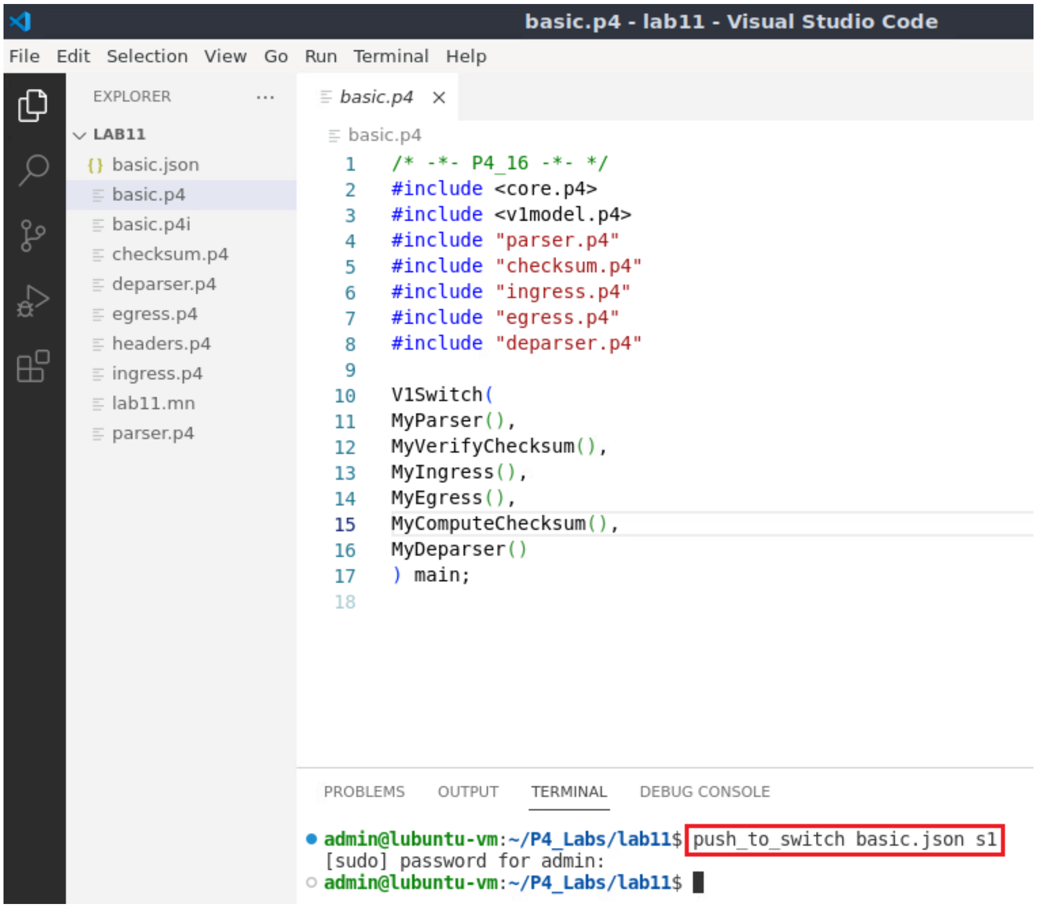
```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab11$ p4c basic.p4
```

Figure 10. Compiling the P4 program using the VS Code terminal.

Step 4. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. The script accepts as input the JSON output of the *p4c* compiler, and the target switch name (e.g., s1). If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```

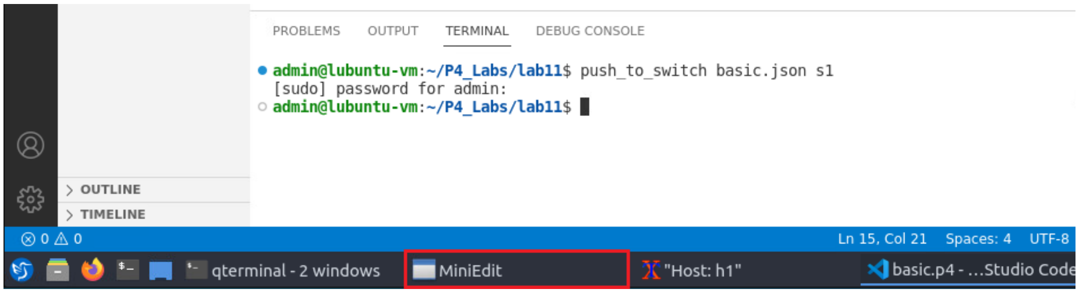

basic.p4 - lab11 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
LAB11
basic.json
basic.p4
basic.p4i
checksum.p4
deparser.p4
egress.p4
headers.p4
ingress.p4
lab11.mn
parser.p4
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>
#include "parser.p4"
#include "checksum.p4"
#include "ingress.p4"
#include "egress.p4"
#include "deparser.p4"

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

```
admin@lubuntu-vm:~/P4_Labs/lab11$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab11$
```

Figure 11. Downloading the compiled program to switch s1.

Step 5. Click on the MinEdit tab in the start bar to maximize the window.


PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
OUTLINE
TIMELINE
Ln 15, Col 21 Spaces: 4 UTF-8
qterminal - 2 windows
MiniEdit
"Host: h1"
basic.p4 - ...Studio Code

```
admin@lubuntu-vm:~/P4_Labs/lab11$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab11$
```

Figure 12. Maximizing the MiniEdit window.

Step 6. Right-click on the P4 switch icon in MiniEdit and select *Terminal*.

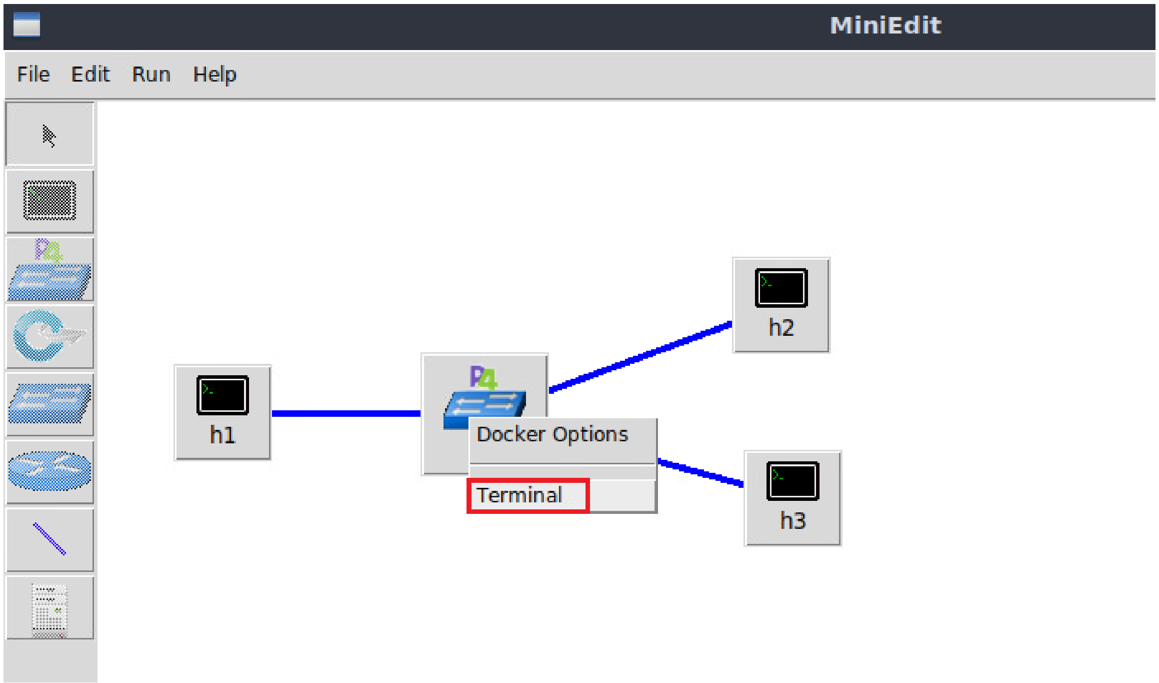
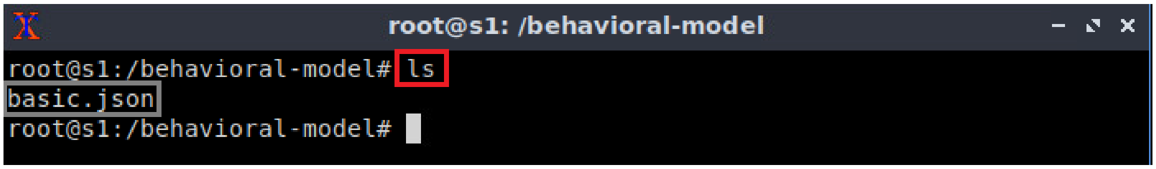


Figure 13. Starting the terminal on switch s1.

Note that the switch is running on an Ubuntu image started on a Docker container. Thus, you will be able to execute any Linux command on the switch's terminal.

Step 7. Issue the following command to list the files in the current directory.

```
ls
```



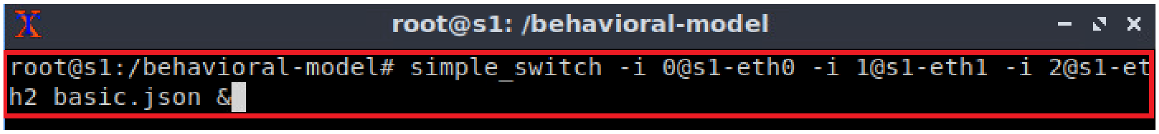
```
root@s1:/behavioral-model# ls
basic.json
root@s1:/behavioral-model# 
```

Figure 14. Displaying the contents of the current directory in the switch s1.

We can see that the switch contains the *basic.json* file that was downloaded after compiling the P4 program.

Step 8. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

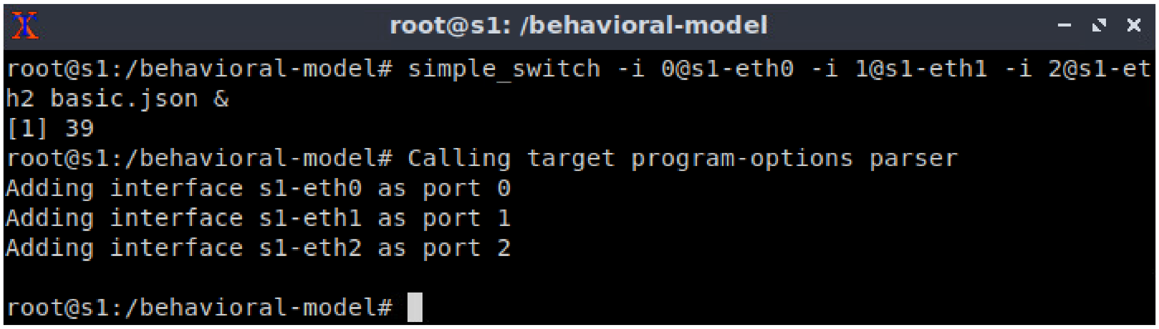
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
```

Figure 15. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 9. In switch s1 terminal, press *Enter* to return the CLI.

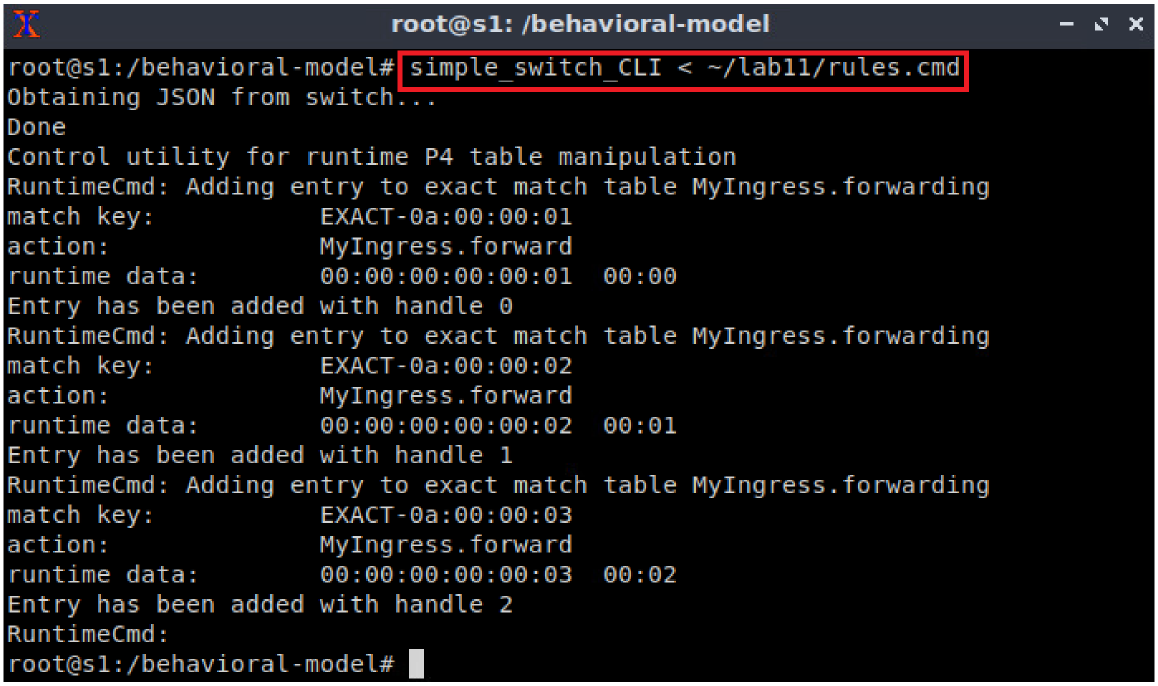


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[1] 39
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model# 
```

Figure 16. Returning to switch s1 CLI.

Step 10. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab11/rules.cmd
```



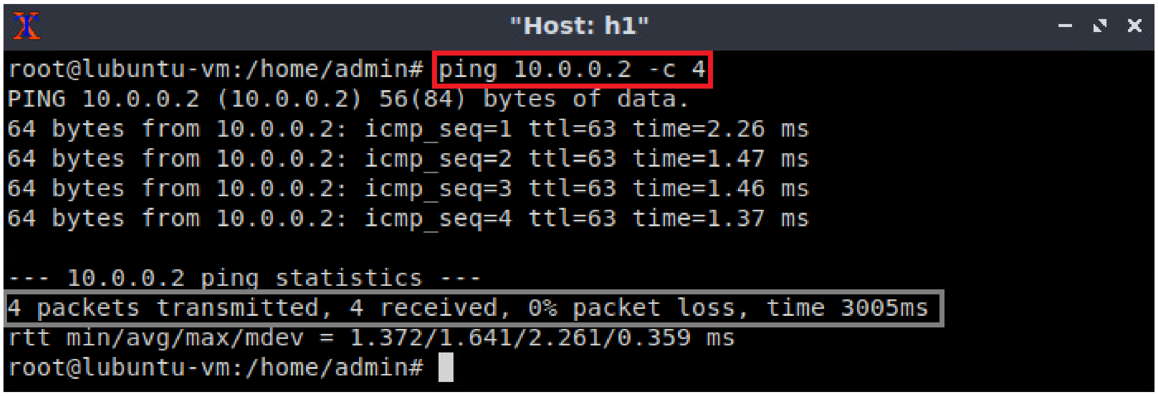
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab11/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:02
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:03
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 2
RuntimeCmd:
root@s1:/behavioral-model# 
```

Figure 17. Loading table entries to switch s1.

The figure above shows the table entries described in the file *rules.cmd*.

Step 11. Go back to host h1 terminal to test the connectivity between host h1 and host h2 by issuing the following command.

```
ping 10.0.0.2 -c 4
```



```
root@lubuntu-vm:/home/admin# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=63 time=2.26 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=63 time=1.47 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=63 time=1.46 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=63 time=1.37 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.372/1.641/2.261/0.359 ms
root@lubuntu-vm:/home/admin#
```

Figure 18. Performing a connectivity test between host h1 and host h2.

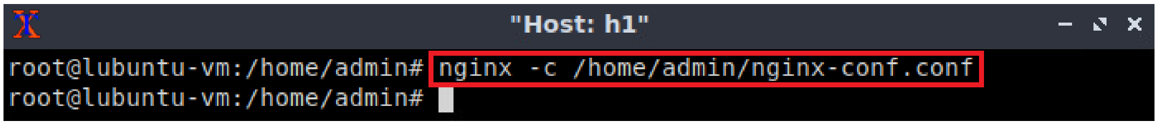
The figure above shows that there is connectivity between the two hosts. Note that at this stage there should be connectivity between any two hosts in the topology.

4 Performing SlowLoris attack

4.1 Starting the HTTP server

Step 1. Start a DNS server on h1 by issuing the command below.

```
nginx -c /home/admin/nginx-conf.conf
```



```
root@lubuntu-vm:/home/admin# nginx -c /home/admin/nginx-conf.conf
root@lubuntu-vm:/home/admin#
```

Figure 19. Starting the HTTP server on h1.

`nginx` is an HTTP server. `-c /home/admin/nginx-conf.conf` specifies to use *nginx-conf.conf* configuration file when starting the HTTP server.

Step 2. Hold the right-click on host h3 and select *Terminal*. This opens the terminal of host h3 and allows the execution of commands on that host.

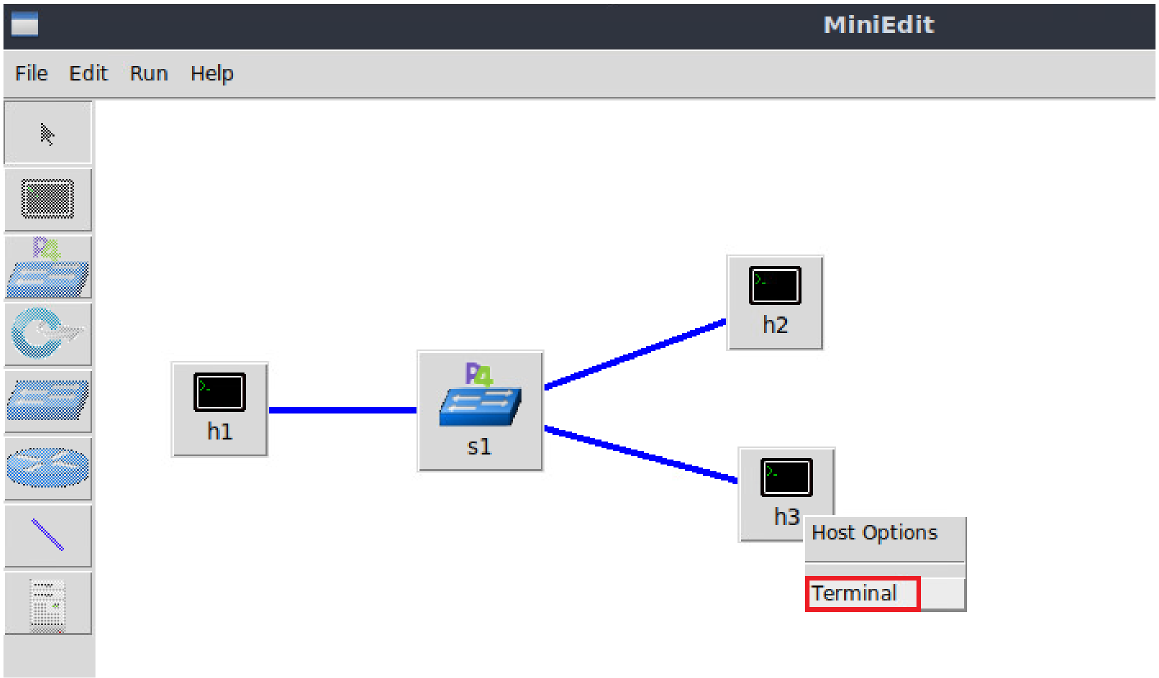
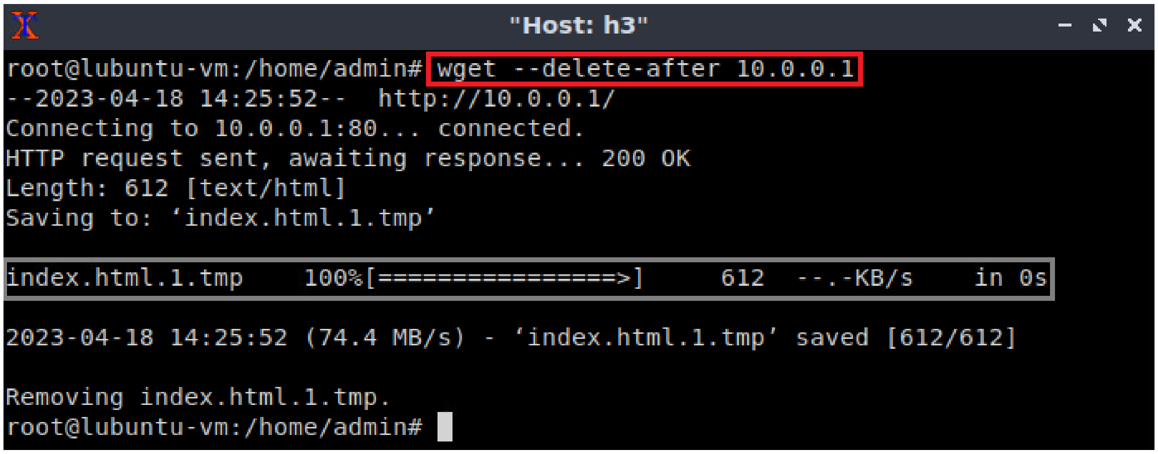


Figure 20. Opening a terminal on host h3.

Step 3. On h3 terminal, type the command below to validate that h1 operates as an HTTP server. `wget` is a utility for non-interactive download of files from the Web. `--delete-after` option tells `wget` to delete every single file it downloads, after having done so. `10.0.0.1` is the IP address of the HTTP server.

```
wget --delete-after 10.0.0.1
```



```
root@lubuntu-vm:/home/admin# wget --delete-after 10.0.0.1
--2023-04-18 14:25:52--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 612 [text/html]
Saving to: ‘index.html.1.tmp’

index.html.1.tmp    100%[================>]     612  --.-KB/s    in 0s

2023-04-18 14:25:52 (74.4 MB/s) - ‘index.html.1.tmp’ saved [612/612]

Removing index.html.1.tmp.
root@lubuntu-vm:/home/admin# 
```

Figure 21. Issuing HTTP Get request.

The figure above shows that h3 downloaded a file from h1 using HTTP GET request.

4.2 Performing the attack

In this section, h3 will perform SlowLoris attack against the web server running on h1.

Step 1. Hold the right-click on host h2 and select *Terminal*. This opens the terminal of host h2 and allows the execution of commands on that host.

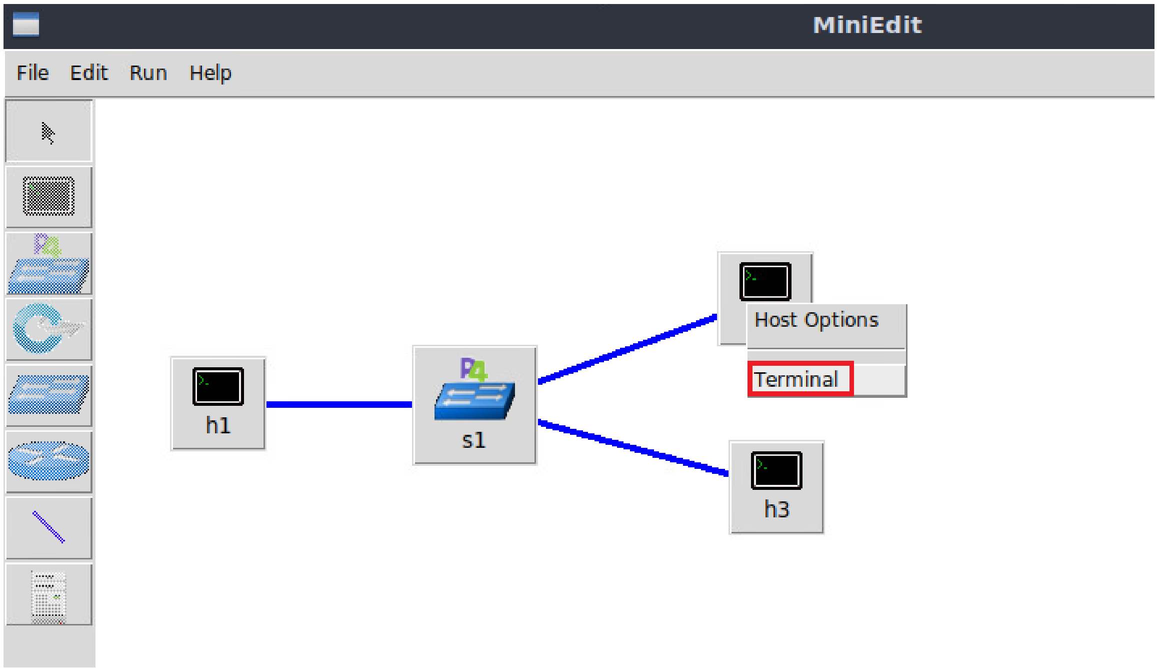
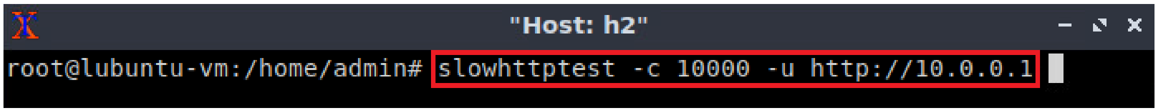


Figure 22. Opening a terminal on host h2.

Step 2. On h2 terminal, type the command below to perform SlowLoris attack on the web server running on h1. `slowhttptest` implements the most common low-bandwidth application Layer DoS attacks. `-c` sets the number of connections to be initiated by the attack. `-u` specifies the URL of the target server.

```
slowhttptest -c 10000 -u http://10.0.0.1
```



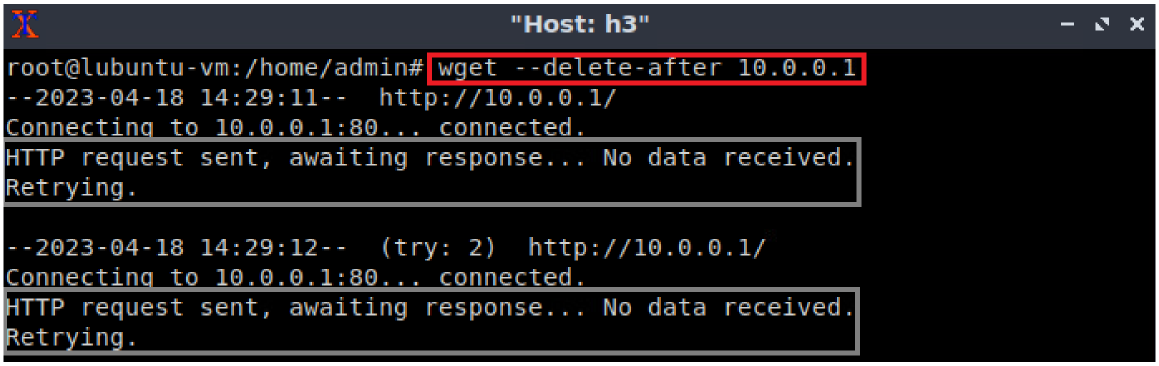
```
root@lubuntu-vm:/home/admin# slowhttptest -c 10000 -u http://10.0.0.1
```

Figure 23. Performing SlowLoris attack.

The attack needs around 1 minute to occupy all the available connection of the web server. Wait for one minute before moving to the next step.

Step 3. On h3 terminal, type the command below to perform a legitimate HTTP GET request.

```
wget --delete-after 10.0.0.1
```



```
root@lubuntu-vm:/home/admin# wget --delete-after 10.0.0.1
--2023-04-18 14:29:11--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... No data received.
Retrying.

--2023-04-18 14:29:12--  (try: 2)  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... No data received.
Retrying.
```

Figure 24. Issuing HTTP Get request.

The figure above shows that h3 was not able to perform an HTTP request because all the resources at the web server (i.e., h1) are occupied by the attack performed by h2.

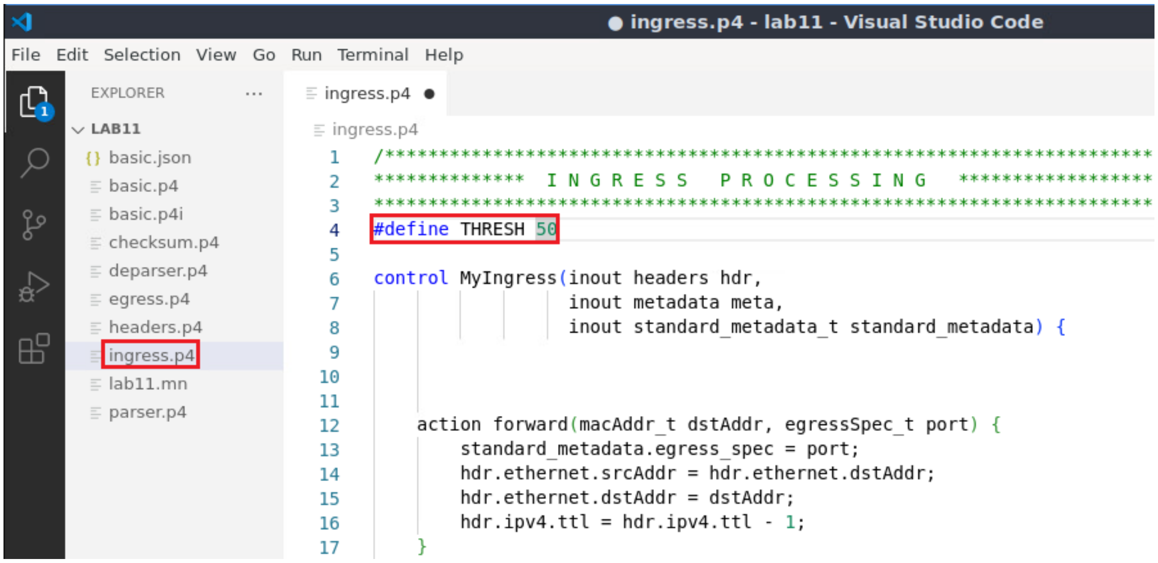
5 Modifying the P4 program to mitigate SlowLoris

In this section, the P4 program will be modified to mitigate slow DDoS attacks. To do this, a register array that stores the number of ongoing connections per host will be initiated. The array is indexed by the hash of the source IP address of the incoming SYN packets. The number of ongoing connections will be increased when a new SYN packet is received, and the counter will be decremented when a new FIN packet is received.

5.1 Modifying the ingress file

Step 1. Use VScode to access the *ingress.p4* file. In the *ingress.p4* file, define the variable `THRESH`. `THRESH` represents the maximum number of allowed connections per IP address. The maximum number of allowed connections is 50.

```
#define THRESH 50
```

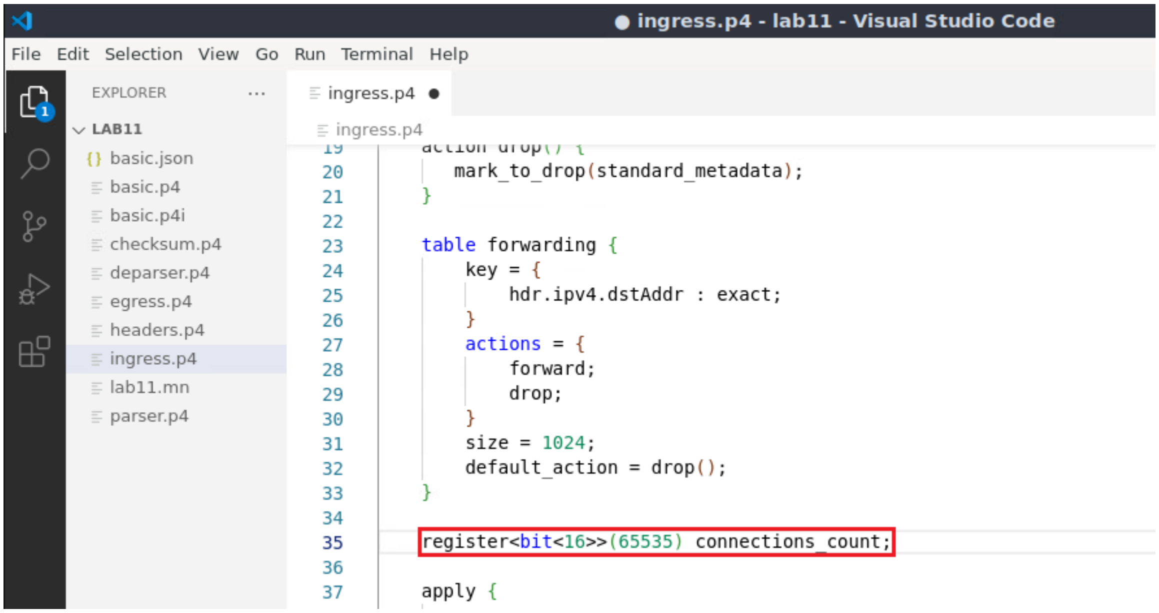


```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************
#define THRESH 50

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {


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

Figure 25. Defining the number of allowed connections.

Step 2. Define the register `connections_count` to store the number of ongoing connections per host.

```
register<bit<16>>(65536) connections_count;
```



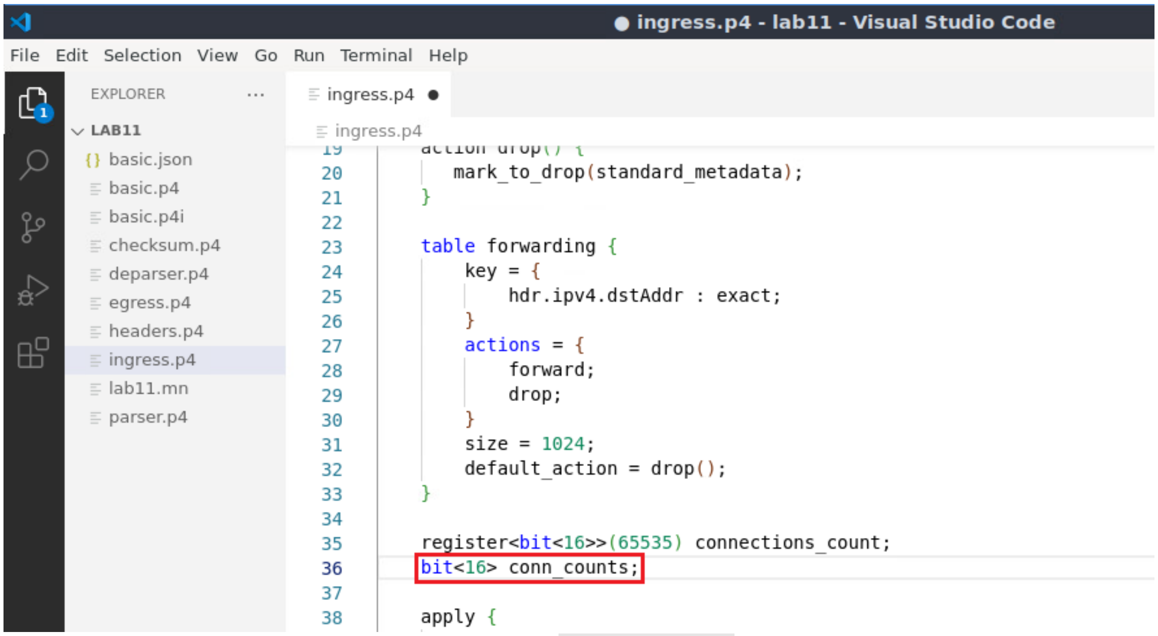
```
    action drop() {
        mark_to_drop(standard_metadata);
    }

    table forwarding {
        key = {
            hdr.ipv4.dstAddr : exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    register<bit<16>>(65535) connections_count;

    apply {
```

Figure 26. Defining register to store the number of connections.

The code above defines a register named `connections_count`. The register contains 65536 cells. Each cell will be indexed by the hash of the source IP address and will store the number of ongoing connections of that IP address.

Step 3. Define the variable `conn_counts` which will be used to temporarily hold the number of connections retrieved from the array.

```
bit<16> conn_counts;
```

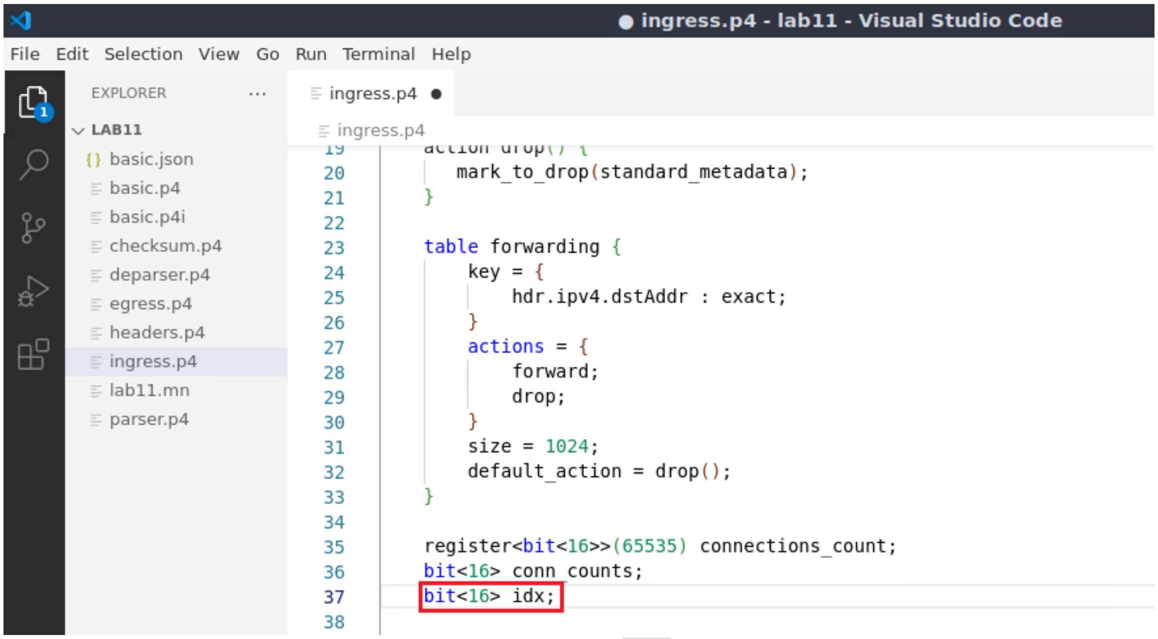


```
    action drop() {
        mark_to_drop(standard_metadata);
    }

    table forwarding {
        key = {
            hdr.ipv4.dstAddr : exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    register<bit<16>>(65535) connections_count;
    bit<16> conn_counts;

    apply {
```

Figure 27. Declaring `conn_counts` variable.

Step 4. Define the variable `idx` to store the hash of the source IP address.

```
bit<16> idx;
```



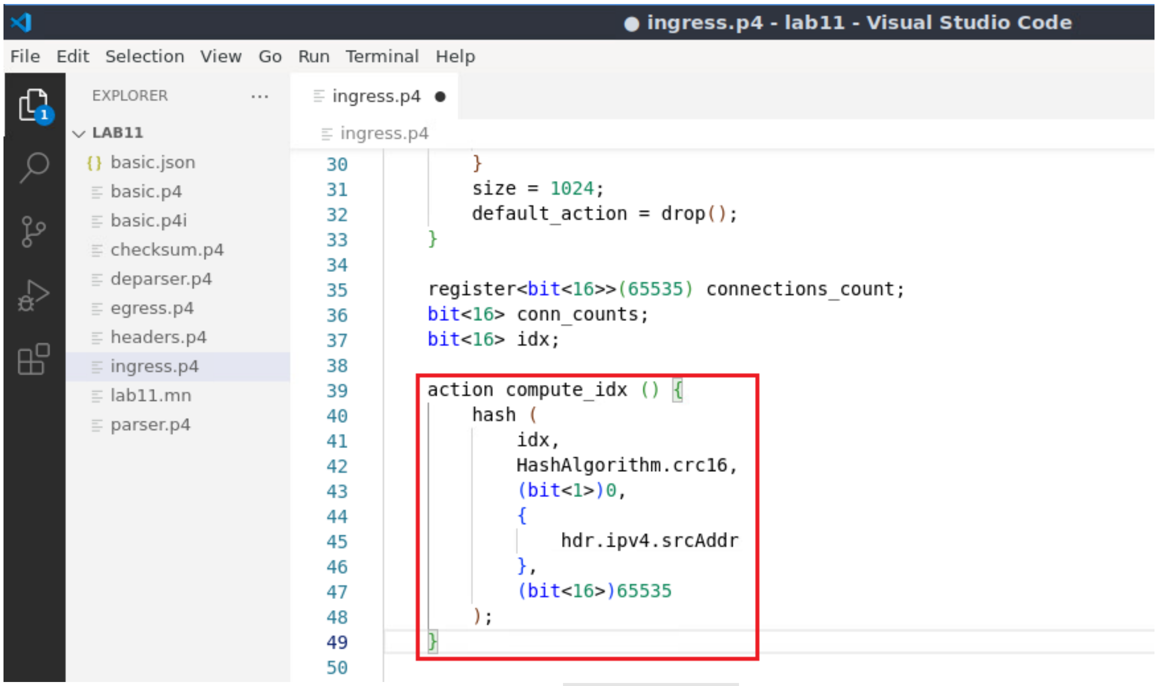
```
    action drop() {
        mark_to_drop(standard_metadata);
    }

    table forwarding {
        key = {
            hdr.ipv4.dstAddr : exact;
        }
        actions = {
            forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    register<bit<16>>(65535) connections_count;
    bit<16> conn_counts;
    bit<16> idx;

```

Figure 28. Declaring `idx` variable.

Step 5. Define the action `compute_idx` by typing the following code.

```
action compute_idx(){
   hash (
       idx,
       HashAlgorithm.crc16,
       (bit<1>)0,
       {
           hdr.ipv4.srcAddr
```

```
        },
        (bit<16>)65535
        );
}
```



```
        }
        size = 1024;
        default_action = drop();
    }

    register<bit<16>>(65535) connections_count;
    bit<16> conn_counts;
    bit<16> idx;

    action compute_idx () {
        hash (
            idx,
            HashAlgorithm.crc16,
            (bit<1>)0,
            {
                hdr.ipv4.srcAddr
            },
            (bit<16>)65535
        );
    }
```

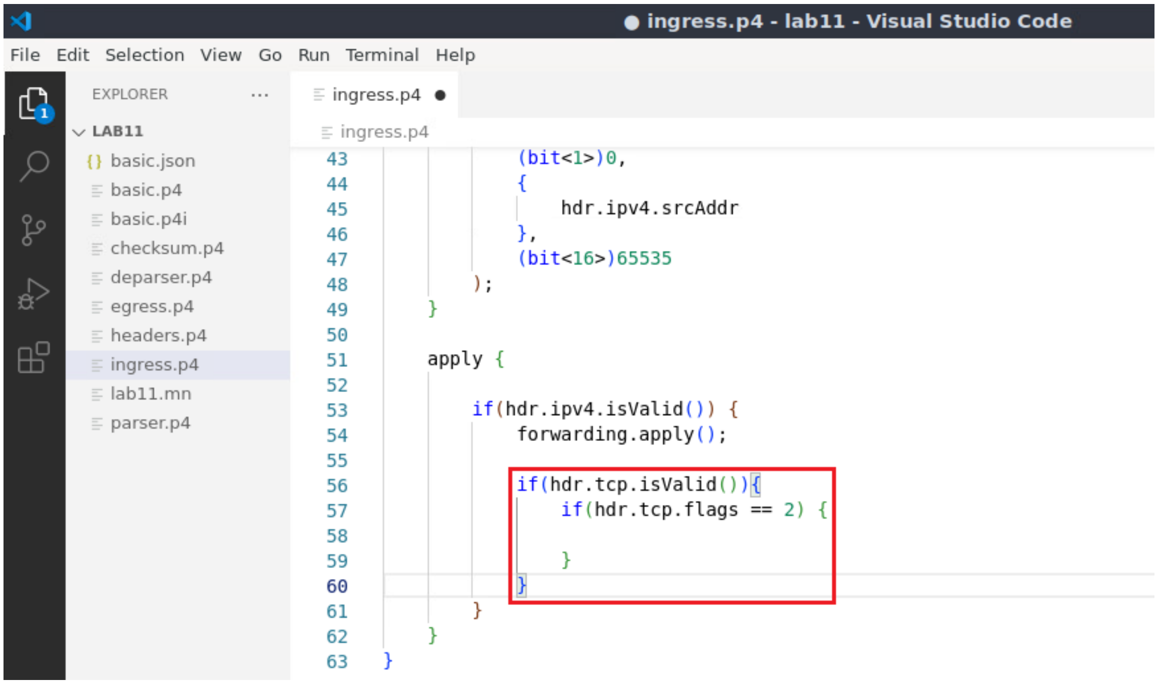
Figure 29. Defining `compute_idx` action.

The code in the figure above hashes flows based on their source IP address. The hash function produces a 16-bits output using the following parameters:

- `idx`: The variable used to store the output.
- `HashAlgorithm.crc16`: the hash algorithm.
- `(bit<1>)0`: the minimum (or base) value produced by the hash algorithm.
- `hdr.ipv4.srcAddr`: the data to be hashed.
- `(bit<32>)65535`: the maximum value produced by the hash algorithm.

Step 6. Add the following code to the *apply* block to check if the packet is a SYN packet.

```
if(hdr.tcp.isValid()){
    if(hdr.tcp.flags == 2){

    }
}
```



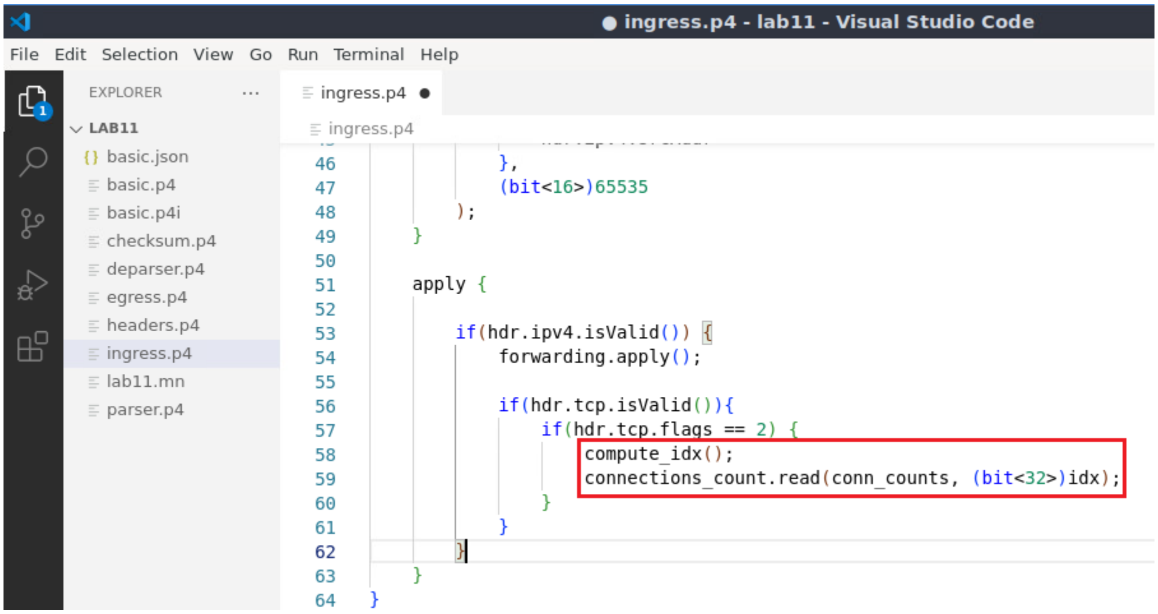
```
            (bit<1>)0,
            {
                hdr.ipv4.srcAddr
            },
            (bit<16>)65535
        );
    }

    apply {

        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()){
                if(hdr.tcp.flags == 2) {

                }
            }
        }
    }
}
```

Figure 30. Checking the type of the SYN packet.

In the code above, `if(hdr.tcp.isValid())` checks if the packet is a TCP packet. For TCP packets, `if(hdr.tcp.flags == 2)` checks if the TCP packet is a SYN packet by inspecting the flags field.

Step 7. Add the following code to retrieve the number of connections originated from the same IP address.

```
compute_idx();
connections_count.read(conn_counts,(bit<32>)idx);
```



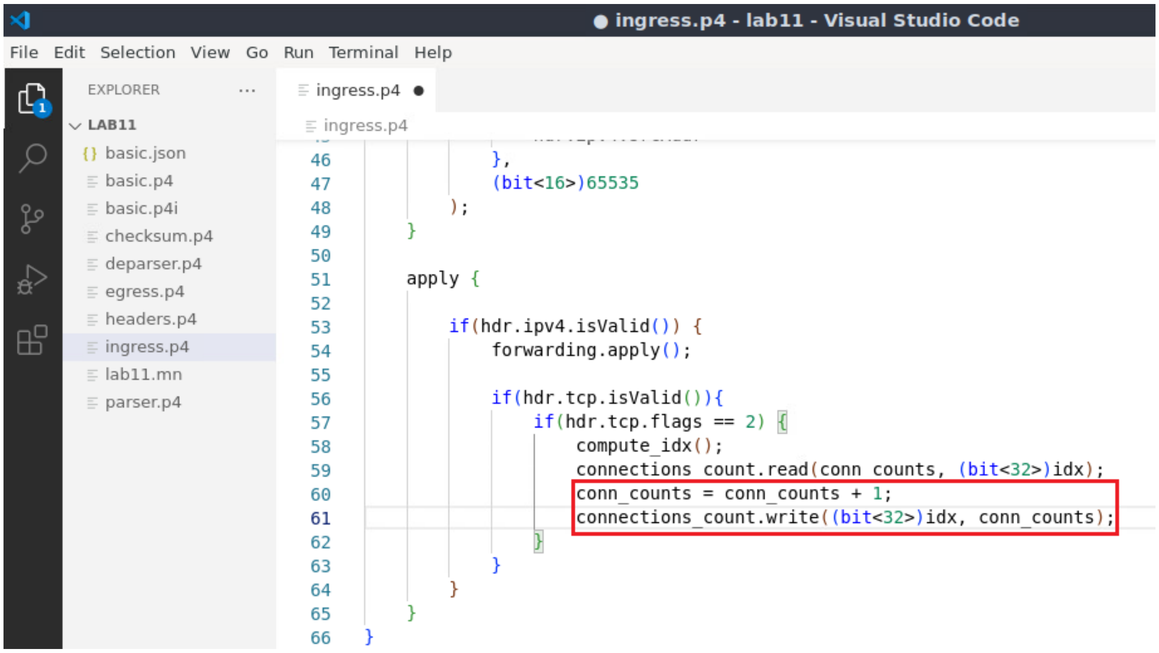
```
            },
            (bit<16>)65535
        );
    }

    apply {

        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()){
                if(hdr.tcp.flags == 2) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                }
            }
        }
    }
}
```

Figure 31. Implementing the apply block.

In the code above, `compute_idx` action calculates the hash of the incoming packet. `connections_count.read (conn_counts,(bit<32>)idx)` retrieves the number of connections stored at index `idx` and store it in the variable `conn_counts`.

Step 8. Add the following code to increment the count of connections and store the incremented value in the register `connections_count`.

```
conn_counts = conn_count + 1;
connections_count.write((bit<32>)idx, conn_counts);
```

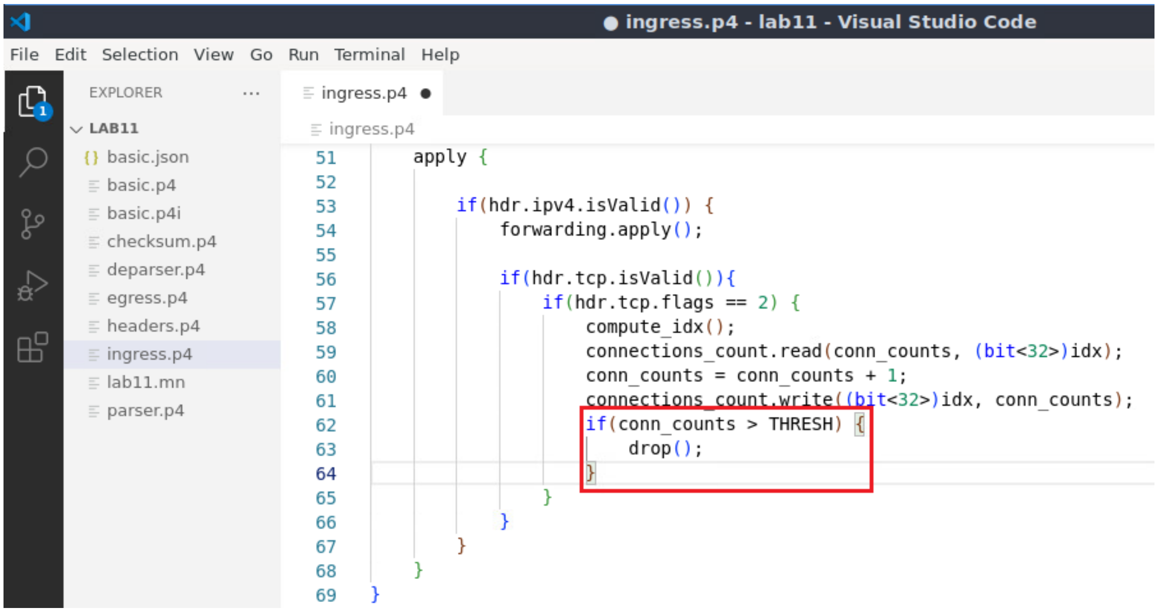


```
            },
            (bit<16>)65535
        );
    }

    apply {

        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()){
                if(hdr.tcp.flags == 2) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                    conn_counts = conn_counts + 1;
                    connections_count.write((bit<32>)idx, conn_counts);
                }
            }
        }
    }
}
```

Figure 32. Incrementing the number of connections.

Step 9. Add the following code to drop the current packet if `conn_counts` is larger than `THRESH`.

```
if(conn_counts > THRESH){
    drop();
}
```

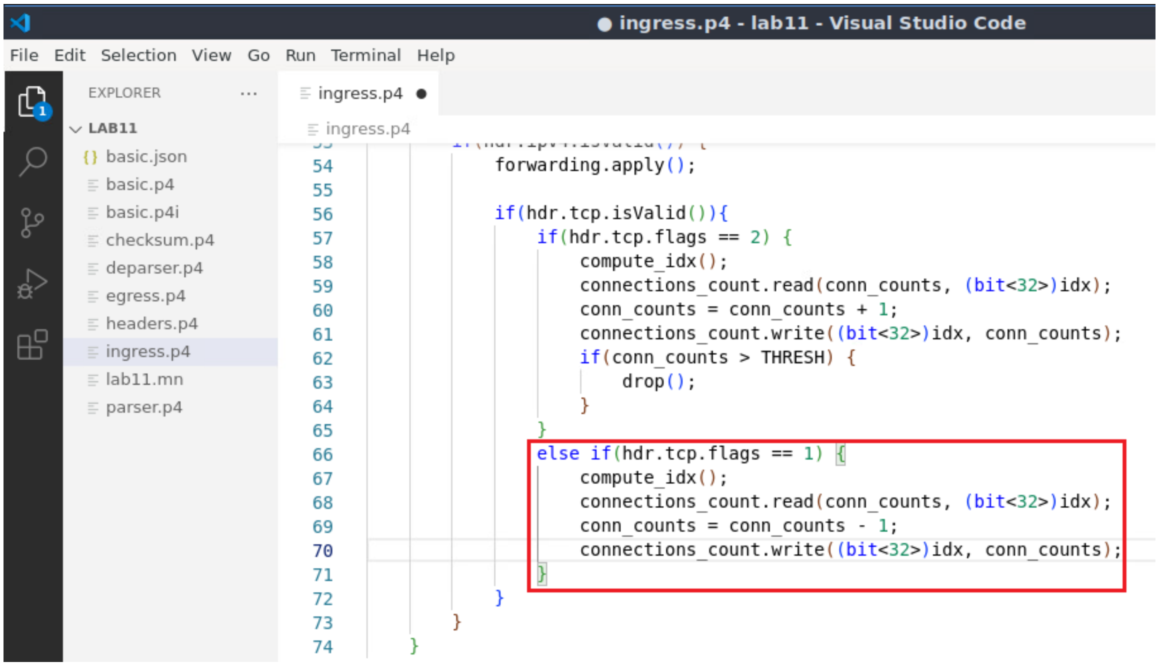


```
    apply {

        if(hdr.ipv4.isValid()) {
            forwarding.apply();

            if(hdr.tcp.isValid()){
                if(hdr.tcp.flags == 2) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                    conn_counts = conn_counts + 1;
                    connections_count.write((bit<32>)idx, conn_counts);
                    if(conn_counts > THRESH) {
                        drop();
                    }

                }
            }
        }
    }
}
```

Figure 33. Dropping SYN packet.

Step 10. Add the following code to decrease the number of connections when receiving a FIN packet.

```
else if(hdr.tcp.flags == 1) {
   compute_idx();
   connections_count.read(conn_counts, (bit<32>)idx);
   conn_counts = conn_counts – 1;
   connections_count.write((bit<32>)idx, conn_counts);
}
```



```
            forwarding.apply();

            if(hdr.tcp.isValid()){
                if(hdr.tcp.flags == 2) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                    conn_counts = conn_counts + 1;
                    connections_count.write((bit<32>)idx, conn_counts);
                    if(conn_counts > THRESH) {
                        drop();
                    }
                }
                else if(hdr.tcp.flags == 1) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                    conn_counts = conn_counts - 1;
                    connections_count.write((bit<32>)idx, conn_counts);
                }
            }
        }
    }
```

Figure 34. Decrementing the number of connections.

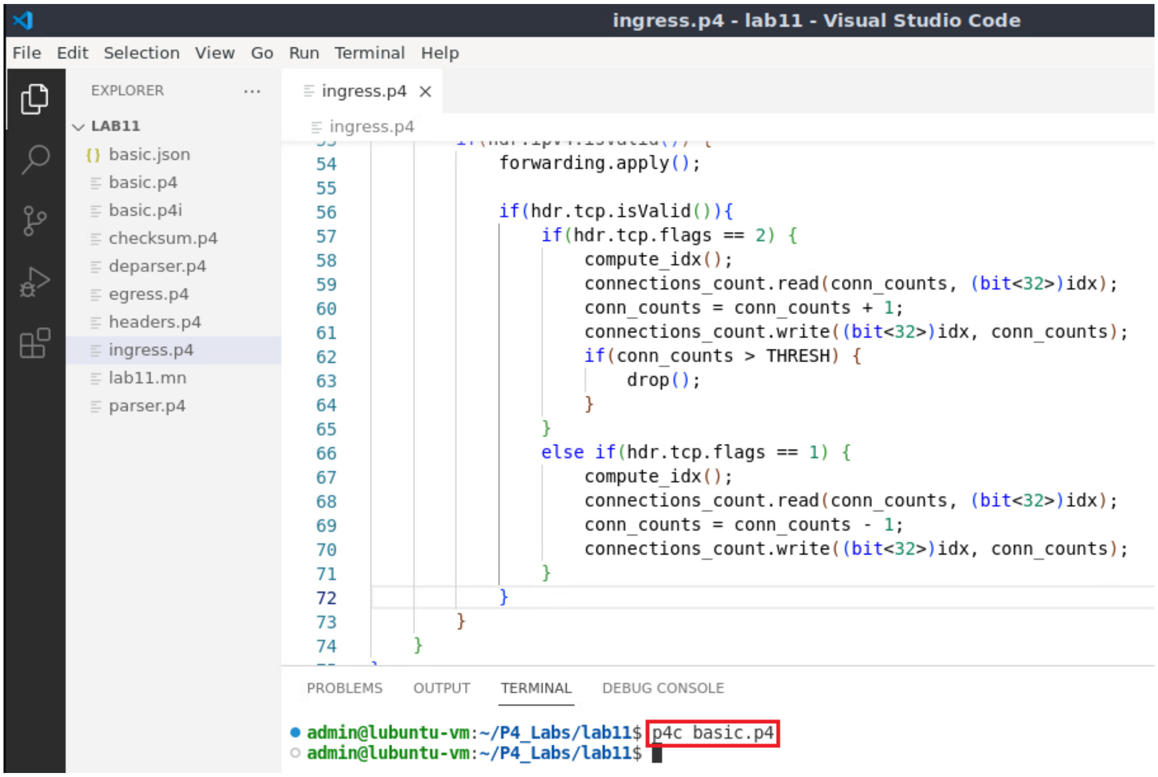
In the code above, `else if(hdr.tcp.flags == 1 )` checks if the packet is a FIN packet. For FIN packets, the corresponding index is calculated using `compute_idx` function. The number of connections is retrieved from `connections_count` register, decremented, and then stored back in the register.

Step 11. Save the changes to the file by pressing `Ctrl + s`.

5.2 Loading the program and configuring the switch

Step 1. To compile the P4 program, issue the following command in the terminal panel inside the VS Code.

```
p4c basic.p4
```



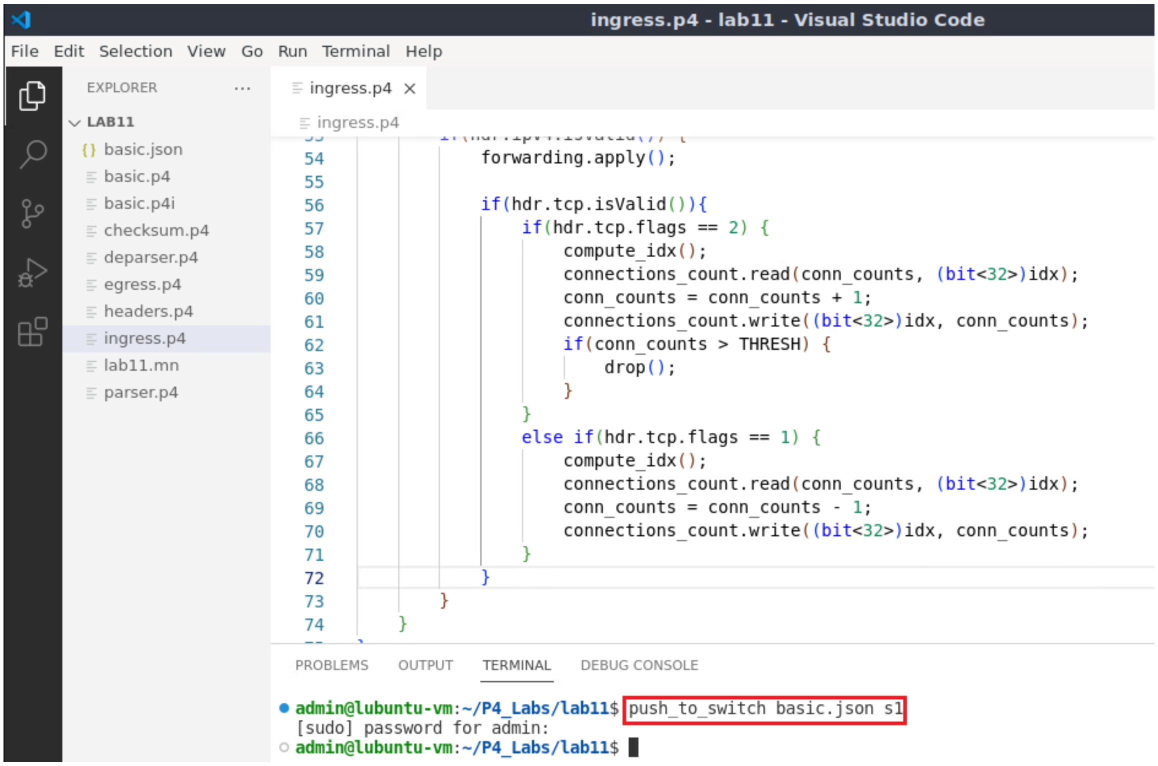
```
            forwarding.apply();

            if(hdr.tcp.isValid()){
                if(hdr.tcp.flags == 2) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                    conn_counts = conn_counts + 1;
                    connections_count.write((bit<32>)idx, conn_counts);
                    if(conn_counts > THRESH) {
                        drop();
                    }
                }
                else if(hdr.tcp.flags == 1) {
                    compute_idx();
                    connections_count.read(conn_counts, (bit<32>)idx);
                    conn_counts = conn_counts - 1;
                    connections_count.write((bit<32>)idx, conn_counts);
                }
            }
        }
    }
```

```
admin@lubuntu-vm:~/P4_Labs/lab11$ p4c basic.p4
admin@lubuntu-vm:~/P4_Labs/lab11$
```

Figure 35. Compiling the P4 program using the VS Code terminal.

Step 2. Type the command below in the terminal panel to download the *basic.json* file to the switch s1. If asked for a password, type the password `password`.

```
push_to_switch basic.json s1
```



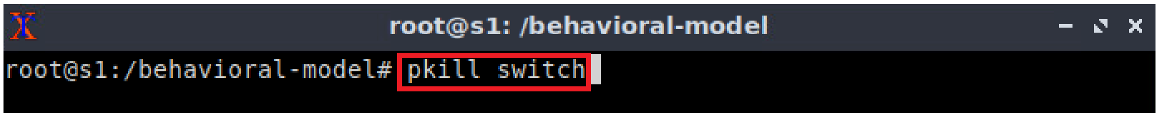
```
        forwarding.apply();

        if(hdr.tcp.isValid()){
            if(hdr.tcp.flags == 2) {
                compute_idx();
                connections_count.read(conn_counts, (bit<32>)idx);
                conn_counts = conn_counts + 1;
                connections_count.write((bit<32>)idx, conn_counts);
                if(conn_counts > THRESH) {
                    drop();
                }
            }
            else if(hdr.tcp.flags == 1) {
                compute_idx();
                connections_count.read(conn_counts, (bit<32>)idx);
                conn_counts = conn_counts - 1;
                connections_count.write((bit<32>)idx, conn_counts);
            }
        }
    }
}
```

```
admin@lubuntu-vm:~/P4_Labs/lab11$ push_to_switch basic.json s1
[sudo] password for admin:
admin@lubuntu-vm:~/P4_Labs/lab11$
```

Figure 36. Downloading the compiled program to switch s1.

Step 3. In switch s1 terminal, type the command below to kill the simple switch daemon, so that the new P4 program can be loaded.

```
pkill switch
```

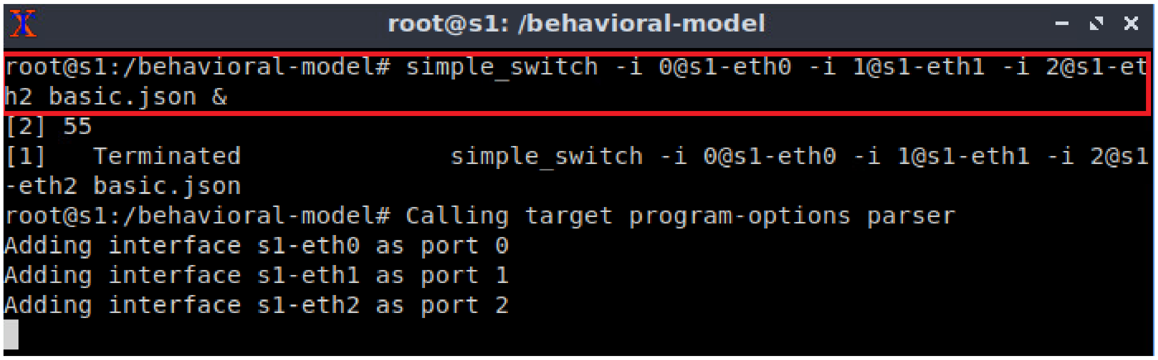


```
root@s1:/behavioral-model# pkill switch
```

Figure 37. Killing the simple switch daemon.

Step 4. Start the switch daemon and map the ports to the switch interfaces by typing the following command.

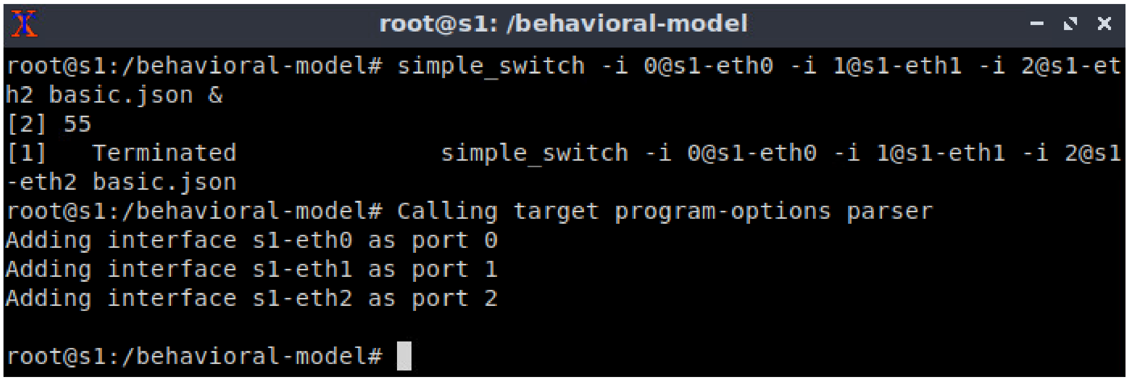
```
simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
```



```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-et
h2 basic.json &
[2] 55
[1]   Terminated              simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1
-eth2 basic.json
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2
```

Figure 38. Starting the switch daemon and mapping the logical interfaces to Linux interfaces.

Step 5. In switch s1 terminal, press *Enter* to return the CLI.

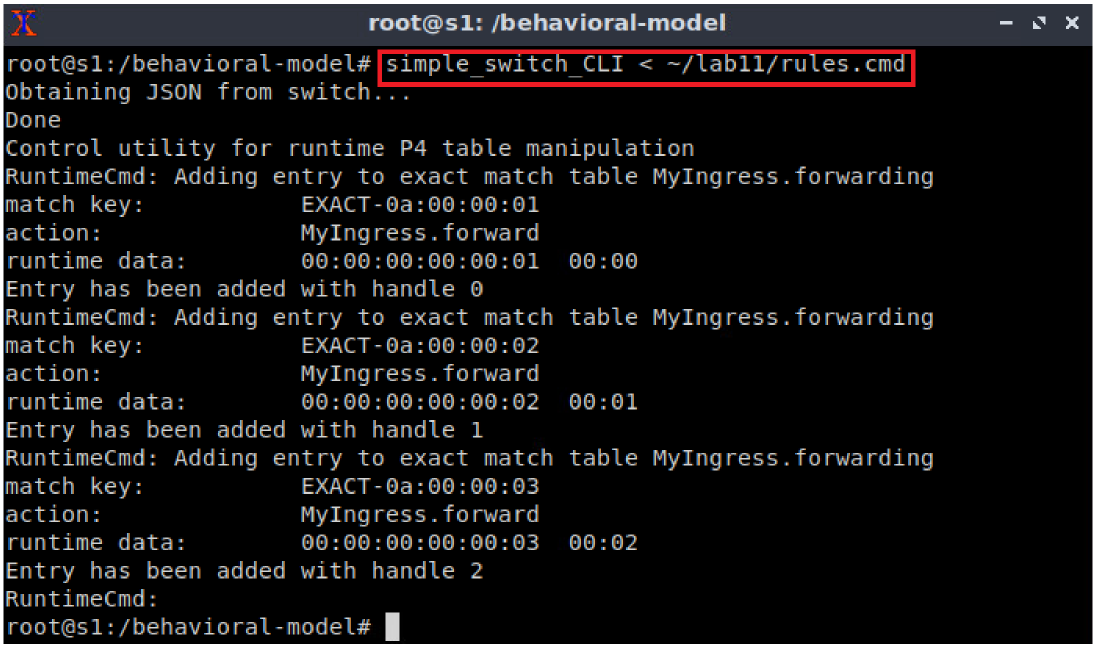


```
root@s1:/behavioral-model# simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json &
[2] 55
[1]   Terminated              simple_switch -i 0@s1-eth0 -i 1@s1-eth1 -i 2@s1-eth2 basic.json
root@s1:/behavioral-model# Calling target program-options parser
Adding interface s1-eth0 as port 0
Adding interface s1-eth1 as port 1
Adding interface s1-eth2 as port 2

root@s1:/behavioral-model#
```

Figure 39. Returning to switch s1 CLI.

Step 6. Populate the table with forwarding rules by typing the following command.

```
simple_switch_CLI < ~/lab11/rules.cmd
```



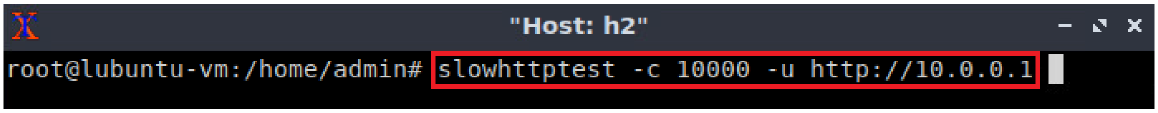
```
root@s1:/behavioral-model# simple_switch_CLI < ~/lab11/rules.cmd
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:01
action:              MyIngress.forward
runtime data:        00:00:00:00:00:01   00:00
Entry has been added with handle 0
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:02
action:              MyIngress.forward
runtime data:        00:00:00:00:00:02   00:01
Entry has been added with handle 1
RuntimeCmd: Adding entry to exact match table MyIngress.forwarding
match key:           EXACT-0a:00:00:03
action:              MyIngress.forward
runtime data:        00:00:00:00:00:03   00:02
Entry has been added with handle 2
RuntimeCmd:
root@s1:/behavioral-model#
```

Figure 40. Loading table entries to switch s1.

5.3 Testing the P4 program

Step 1. On h2 terminal, type the command below to perform SlowLoris attack on the web server running on h1.

```
slowhttptest -c 10000 -u http://10.0.0.1
```



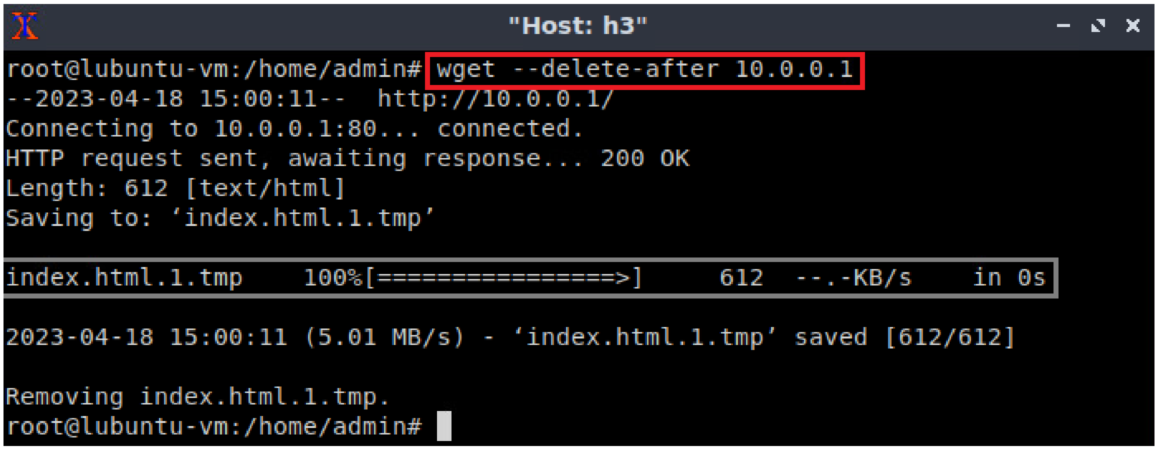
```
root@lubuntu-vm:/home/admin# slowhttptest -c 10000 -u http://10.0.0.1
```

Figure 41. Performing SlowLoris attack.

The attack needs around 1 minute to occupy all the available connection of the web server. Wait for one minute before moving to the next step.

Step 2. On h3 terminal, type the command below to perform a legitimate HTTP GET request.

```
wget --delete-after 10.0.0.1
```



```
root@lubuntu-vm:/home/admin# wget --delete-after 10.0.0.1
--2023-04-18 15:00:11--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 612 [text/html]
Saving to: ‘index.html.1.tmp’

index.html.1.tmp    100%[================>]     612  --.-KB/s    in 0s

2023-04-18 15:00:11 (5.01 MB/s) - ‘index.html.1.tmp’ saved [612/612]

Removing index.html.1.tmp.
root@lubuntu-vm:/home/admin# 
```

Figure 42. Issuing HTTP Get request.

The figure above shows that h3 downloaded the file from the web server. The P4 switch was able to stop the SlowLoris attack.

This concludes lab 11. Stop the emulation and then exit out of MiniEdit.

References

1. NETSCOUT, “What is a Slowloris Attack?” [Online]. Available: https://tinyurl.com/3awxn2ws
2. Cloudflare, “Slowloris DDoS attack?” [Online]. Available: https://tinyurl.com/mrarw9ub
3. Zach Norton, “How to Mitigate a Slowloris DDoS Attack.” [Online]. Available: https://tinyurl.com/5n6h8brm