Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

SDN Project Work (Firewall) - Network and Cloud Infrastructures

Professore
Prof. Giorgio Ventre

Candidati
Francesco Brunello - M63/1655
Vincenzo Luigi Bruno - M63/1670
Salvatore Cangiano - M63/1647

Anno Accademico 2023–2024

Contents

Chapter 1

Network Setup and Performance

1.1 Topology Definition

The topology implemented consists of 3 **hosts** (*h1*, *h2*, and *h3*) and 4 **switches** (*s1*, *s2*, *s3*, and *s4*). The hosts are connected to the switches in a manner that simulates a small-scale, multi-switch network. The entire setup is controlled by a **remote OpenFlow controller.**

```
class Environment(object):
    def __init__(self):
        "Create a network."
        self.net = Mininet(controller=RemoteController, link=TCLink)
        info("*** Starting controller\n")
        #Controller
        c1 = self.net.addController('c1', controller=RemoteController)
        c1.start()
```

Figure 1.1: Environment Class Initialization

The **Environment class** is designed to encapsulate the network setup. The controller (*c1*) is added to the network and started, ensuring that the OpenFlow switches in the topology will be managed by this remote controller.

```python
info("*** Adding hosts and switches\n")
self.h1 = self.net.addHost('h1', mac='00:00:00:00:00:01', ip='10.0.0.1')
self.h2 = self.net.addHost('h2', mac='00:00:00:00:00:02', ip='10.0.0.2')
self.h3 = self.net.addHost('h3', mac='00:00:00:00:00:03', ip='10.0.0.3')

self.s1 = self.net.addSwitch('s1', cls=OVSKernelSwitch)
self.s2 = self.net.addSwitch('s2', cls=OVSKernelSwitch)
self.s3 = self.net.addSwitch('s3', cls=OVSKernelSwitch)
self.s4 = self.net.addSwitch('s4', cls=OVSKernelSwitch)
```

Figure 1.2: Hosts and Switches

Three hosts (*h1, h2, h3*) are added to the network with specified **MAC** and **IP addresses**. Four switches (*s1, s2, s3, s4*) are also added, each instance being an **OVSKernelSwitch**, which is the default switch type in Mininet and based on Open vSwitch (OVS).

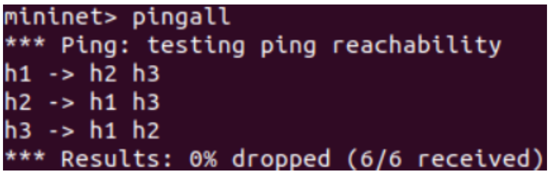
```python
info("*** Adding links\n")
self.net.addLink(self.h1, self.s1, bw=10, delay='0.0025ms')
self.s1_to_s3 = self.net.addLink(self.s1, self.s3, bw=6, delay='25ms')
self.net.addLink(self.h2, self.s2, bw=6, delay='25ms')
self.s2_to_s3 = self.net.addLink(self.s2, self.s3, bw=6, delay='25ms')
self.s3_to_s4 = self.net.addLink(self.s3, self.s4, bw=6, delay='25ms')
self.net.addLink(self.s4, self.h3, bw=10, delay='0.0025ms')
```

Figure 1.3: Adding Links

The code in figure 1.3 establishes the **links between hosts and switches**, as well as between the switches themselves. Switch-to-switch links have a bandwidth of 6 Mbps and a delay of 25 ms, representing slower, long-distance interconnections between network segments.

These link parameters are crucial for testing and understanding the performance and behavior of the network under different conditions.

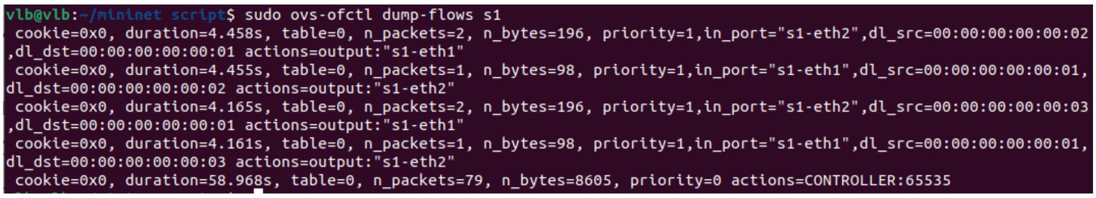
1.2 Basic Reachability through the Controller



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Figure 1.4: Pingall command

The results from the pingall command, as shown in the screenshot, indicate 0% packet loss, which verifies that the initial network setup is functioning correctly



```
vlb@vlb:~/mininet script$ sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=4.458s, table=0, n_packets=2, n_bytes=196, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:02
,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=4.455s, table=0, n_packets=1, n_bytes=98, priority=1,in_port="s1-eth1",dl_src=00:00:00:00:00:01,
dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
 cookie=0x0, duration=4.165s, table=0, n_packets=2, n_bytes=196, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:03
,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=4.161s, table=0, n_packets=1, n_bytes=98, priority=1,in_port="s1-eth1",dl_src=00:00:00:00:00:01,
dl_dst=00:00:00:00:00:03 actions=output:"s1-eth2"
 cookie=0x0, duration=58.968s, table=0, n_packets=79, n_bytes=8605, priority=0 actions=CONTROLLER:65535
```

Figure 1.5: Flow Table Verification in Open vSwitch

To further validate the network's functionality, I examined the **flow entries** on the **switch** (*s1*) using the command *sudo ovs-ofctl dump-flows s1*. The screenshot demonstrates the flow table entries that were automatically generated by the OpenFlow controller during the ping tests. These entries indicate that the **switch is correctly forwarding packets** based on the destination MAC addresses (*dl_dst*) to the appropriate output ports (*actions=output*). Additionally, the default flow entry with a priority of 0 sends **unmatched packets to the controller** (*actions=CONTROLLER:65535*), ensuring that the controller can dynamically manage traffic flows. The presence of multiple entries with different source and destination MAC addresses and corresponding packet counts confirms that the **switch is handling traffic as expected**.

1.3 Traffic Simulation

In this phase, I generated **standard traffic flows** in the network to establish a baseline before simulating any attack. Using ***iperf***, We configured *h1* to send **UDP** traffic and *h2* to send **TCP** traffic, both targeting *h3* with a constant bitrate of 2 Mbps.

The screenshot illustrates the results of these traffic tests, where *h1* and *h2* successfully established connections to *h3* on UDP port 5001 and TCP port 5002, respectively. Both connections maintained a **stable bandwidth of 2.10 Mbps**, reflecting the normal operation of the network under typical traffic conditions.

This setup is crucial for later comparison when testing the network's resilience under higher traffic loads, particularly in the context of a simulated DoS attack.

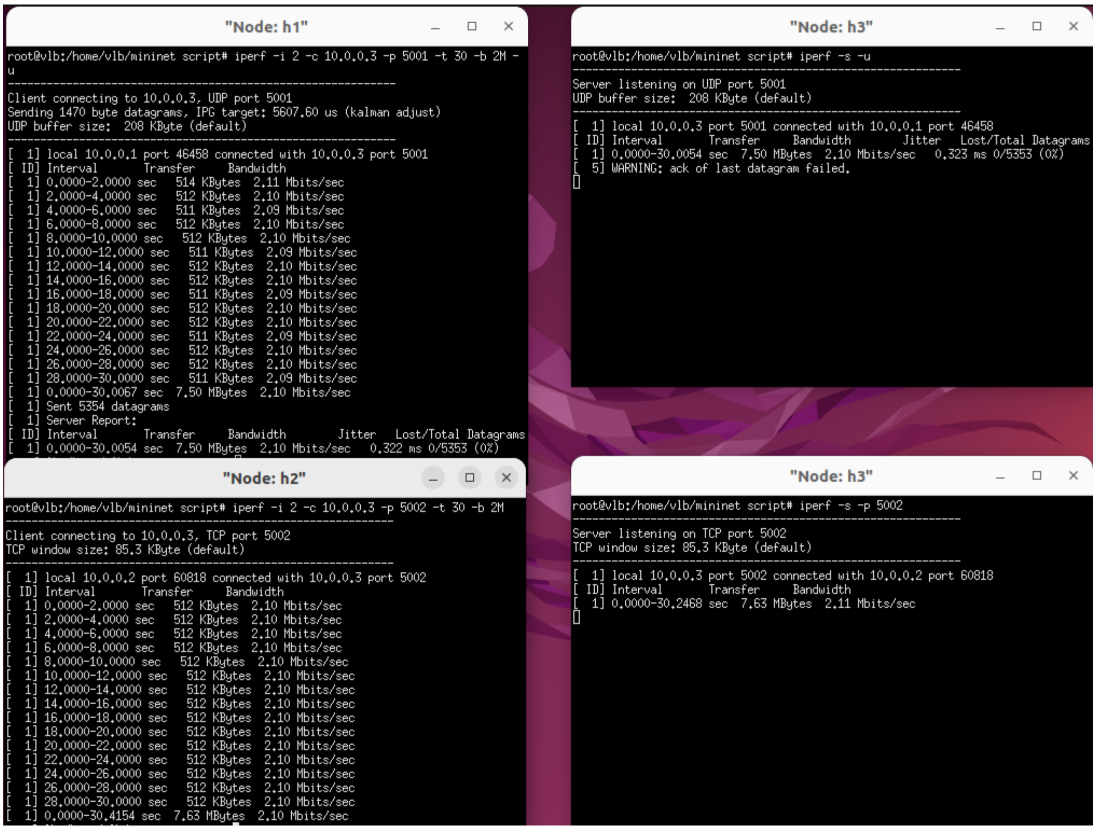


Figure 1.6: Normal Traffic Flow: UDP and TCP Traffic from h1 and h2 to h3

To simulate a DoS attack, Host h1 was configured to generate a large amount of UDP traffic towards Host h3, causing congestion in the network. Meanwhile, Host h2 continued to generate normal TCP traffic towards Host h3, allowing us to observe the impact of the attack on TCP performance.

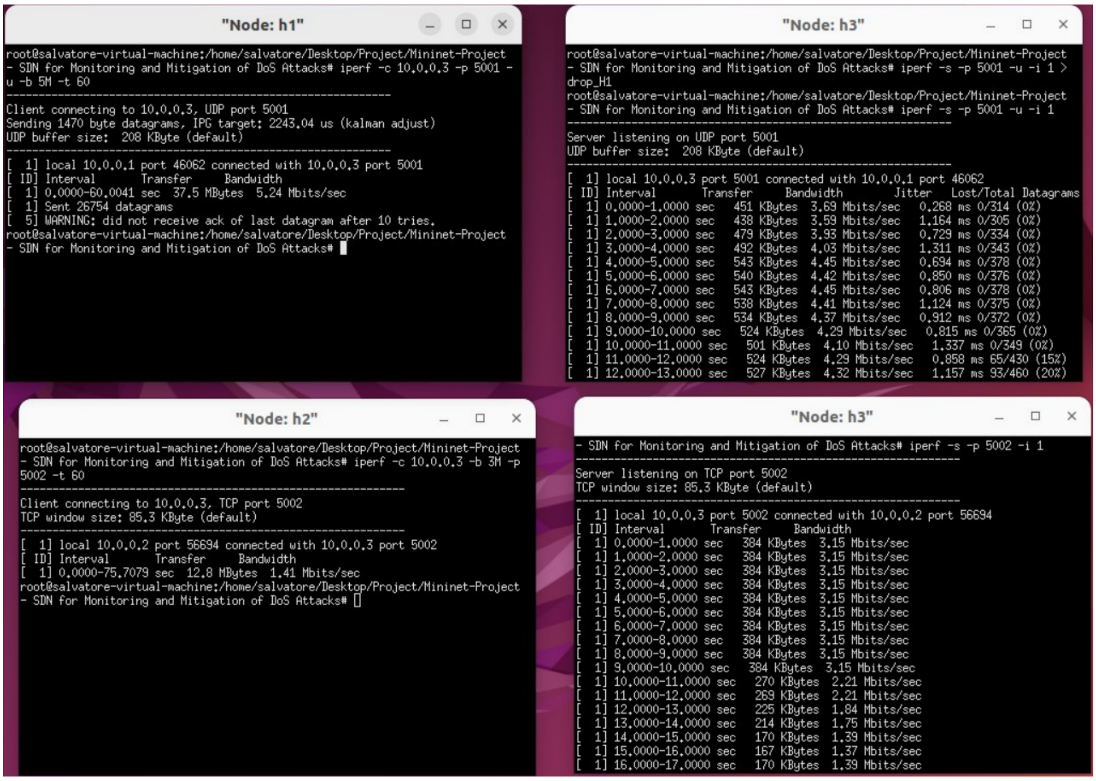


Figure 1.7: iperf outputs on nodes h1, h2, h3

To simulate the DoS attack, the attacker (*h1*) was configured to generate a high volume of UDP traffic towards *h3* using the command:

`iperf -c 10.0.0.3 -p 5001 -u -b 5M -t 60`

This command instructs *h1* to send UDP packets to *h3* on port 5001 with a bandwidth of 5 Mbps for a duration of 60 seconds. Meanwhile, to assess the impact on normal traffic, *h2* continued to send TCP traffic to *h3* using the following command:

`iperf -c 10.0.0.3 -b 3M -p 5002 -t 60`

Here, *h2* maintains a TCP connection to *h3* on port 5002 with a target bandwidth of 3 Mbps, also for 60 seconds. The results showed that the overwhelming UDP traffic generated by *h1* caused significant congestion, severely degrading the TCP throughput from *h2* to *h3*.

1.4 Performance Impact of DoS Attacks

This section presents a comparative analysis of network performance under normal conditions and during a simulated DoS (Denial of Service) attack.

Initially, the baseline performance was established by observing the network's behavior without any external disruptions.

The two graphs represent the baseline network performance when simulating standard traffic flows using iperf between hosts h1 and h3 (UDP flow) and hosts h2 and h3 (TCP flow). Both connections aimed to achieve a consistent bitrate of 2 Mbps.

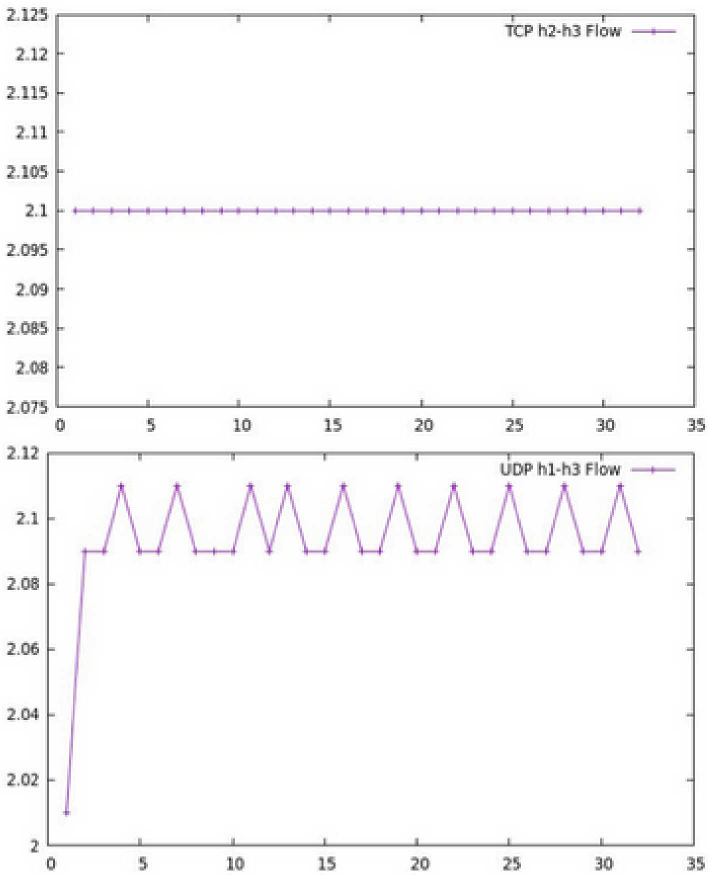


Figure 1.8: Graphs of network performance during standard traffic flows

1. **TCP Flow**: The graph indicates a steady bandwidth of approximately 2.1 Mbps throughout the duration of the test. The line is flat with minimal fluctuations, suggesting that the TCP connection maintained stable performance without significant variation.

2. **UDP Flow**: This flow shows a slightly more variable bandwidth, with small oscillations around 2.1 Mbps after an initial increase from 2.0 Mbps.

These fluctuations are a typical characteristic of UDP traffic since it doesn't include mechanisms for flow control or congestion avoidance like TCP.

The results demonstrated stable throughput for both TCP and UDP traffic, indicating a well-functioning network under standard traffic loads.

Following the baseline assessment, an **attack scenario** was introduced to evaluate the network's resilience under stress.

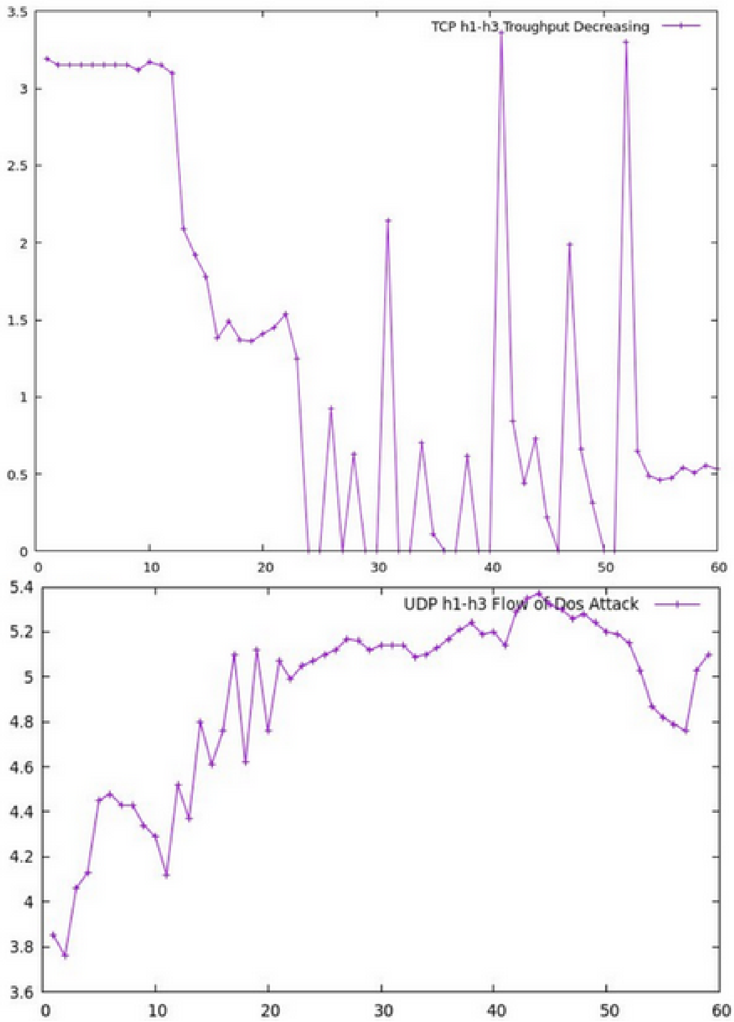


Figure 1.9: Graphs of network performance during attack

The graph shows a significant increase in the UDP bandwidth, peaking around 5.2 Mbps. This is indicative of the high volume of traffic generated by the attacker (*h1*), **overwhelming the network and leading to congestion**. The continuous increase and fluctuations in bandwidth reflect the unregulated nature of UDP traffic under heavy load conditions.

The **severe degradation in TCP throughput** is a direct result of the **network congestion** caused by the excessive UDP traffic. This congestion led to dramatic fluctuations in TCP bandwidth, with throughput dropping as low as 0.5 Mbps. These disruptions highlight the significant impact of the DoS attack, where the overwhelming volume of UDP traffic severely compromised the stability and performance of TCP connections. This underscores the critical importance of implementing effective mitigation strategies in modern networks to protect against such attacks.

Chapter 2

Mitigation & Remediation for the attack

2.1 Traffic Monitoring

The monitoring process consists of sending each switch a request for statistics regarding the incoming and outgoing traffic for each port. The **EventOFP-PortStatsRequest** and **EventOFPPortStatsReply** events were used to do so. The monitoring is handled by a thread to which the **_monitor** function is passed, which requests the previously mentioned statistics.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    global timeInterval
    global RED
    global RESET
    global GREEN

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.send_req = 0
        self.rec_res = 0
        self.threshold=700000 #80-90% del percorso critico
        self.time = 0
        self.datapaths = {}
        self.mac_to_port = {}
        self.monitoring_stats = {}
        self.alarm_switch_port = {}
        self.monitor_thread = hub.spawn(self._monitor)
```

Figure 2.1: Monitoring Thread Inizialization

The request is made periodically, every 10 seconds. By default, the request methods provided by Ryu offer cumulative statistics since the network was created, so an appropriate data handling mechanism is needed to make them relative to a specific time interval.

```
timeInterval = 10

def _monitor(self):
        while True:
            for dp in self.datapaths.values():
                self._request_stats(dp)

 self.logger.info('\n###############################################################')
            hub.sleep(timeInterval)
```

Figure 2.2: Monitoring Method

Below is the method called for each switch: **_request_stats**

```
# Funzione di richiesta delle stats agli switch
    def _request_stats(self, datapath):
        self.logger.debug('send stats request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
        datapath.send_msg(req)
        self.send_req = time.perf_counter()
```

Figure 2.3: Request Stats Method

When a switch responds with a message containing the statistics, the controller handles this message with a dedicated handler that reacts to the event. The **ev** *event* object will contain the data needed by the controller to manage any anomalies.

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
   def _port_stats_reply_handler(self, ev):
       body = ev.msg.body

       self.rec_res = time.perf_counter()

       #Precise time interval
       self.time = timeInterval + (self.rec_res - self.send_req)
       print(self.time)
```

Figure 2.4: PortStat Reply Handler

The structure we have implemented in the controller is a dictionary that stores, for each switch, the port and for each port the statistics reported in the message. This structure will be updated periodically to always show the latest statistics for the most recent time interval (the last 10 seconds). Only in the first iteration are the statistics presented without processing. The structure is self.monitoring_stats and can be represented as follows:{id_switch: {port_number: [stats], ...}, ...}

```
if ev.msg.datapath.id not in self.monitoring_stats.keys():
            self.logger.info('datapath         port     '
                             'rx-pkts   rx-bytes/s   rx-error   '
                             'tx-pkts   tx-bytes/s   tx-error')
            self.logger.info('---------------- --------   '
                             '--------   --------   --------   '
                             '--------   --------   --------')
            for stat in sorted(body, key=attrgetter('port_no')):
                self.logger.info('%016x %8d   %8d   %8d    %8d    %8d    %8d   %8d',
                                 ev.msg.datapath.id, stat.port_no,
                                 stat.rx_packets, stat.rx_bytes / self.time, stat.rx_errors,
                                 stat.tx_packets, stat.tx_bytes / self.time, stat.tx_errors)

            self.monitoring_stats[ev.msg.datapath.id] = {
                stat.port_no: [stat.rx_packets, stat.rx_bytes, stat.rx_errors, stat.tx_packets, stat.tx_bytes, stat.tx_errors]
                for stat in sorted(body, key=attrgetter('port_no'))
            }

            #Inizializzazione della struttura di alarm, questo sarà un contatore, a 3 (dopo 30 secondi) scatterà l'allarme per quella porta.
            self.alarm_switch_port[ev.msg.datapath.id] = {stat.port_no: [0, 0] for stat in sorted(body, key=attrgetter('port_no'))}

        else:
            previous = self.monitoring_stats[ev.msg.datapath.id]
            self.logger.info('datapath         port     '
                             'rx-pkts   rx-bytes/s   rx-error   '
                             'tx-pkts   tx-bytes/s   tx-error')
            self.logger.info('---------------- --------    '
                             '--------   --------   --------   '
                             '--------   --------   --------')
            for stat in sorted(body, key=attrgetter('port_no')):
                self.logger.info('%016x %8x   %8d   %8d   %8d   %8d   %8d   %8d',
                                 ev.msg.datapath.id, stat.port_no,
                                 (stat.rx_packets - previous[stat.port_no][0]),
                                 (stat.rx_bytes - previous[stat.port_no][1]) / self.time,
                                 stat.rx_errors - previous[stat.port_no][2],
                                 (stat.tx_packets - previous[stat.port_no][3]),
                                 (stat.tx_bytes - previous[stat.port_no][4]) / self.time,
                                 stat.tx_errors - previous[stat.port_no][5])
```

Figure 2.5: Showing Stasts

Here are the monitoring results where no active flows are present, but only the network management traffic caused by the switches and the controller. Below are the results when hosts send packets, resulting in simulated traffic using iperf.

```
datapath         port     rx-pkts  rx-bytes/s  rx-error  tx-pkts  tx-bytes/s  tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000004        1        8      101        0        3       36        0
0000000000000004        2        1        6        0       10      130        0
0000000000000004 fffffffe        0        0        0        0        0        0
10.004712673999165
datapath         port     rx-pkts  rx-bytes/s  rx-error  tx-pkts  tx-bytes/s  tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000001        1        1        6        0        8      116        0
0000000000000001        2        7       94        0        3       36        0
0000000000000001 fffffffe        0        0        0        0        0        0
10.005316150996805
datapath         port     rx-pkts  rx-bytes/s  rx-error  tx-pkts  tx-bytes/s  tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000003        1        3       36        0        7       94        0
0000000000000003        2        3       36        0        8      101        0
0000000000000003        3        3       36        0        8      101        0
0000000000000003 fffffffe        0        0        0        0        0        0
10.00805259399931
datapath         port     rx-pkts  rx-bytes/s  rx-error  tx-pkts  tx-bytes/s  tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000002        1        1        6        0       10      130        0
0000000000000002        2        8      101        0        3       36        0
0000000000000002 fffffffe        0        0        0        0        0        0
```

Figure 2.6: Monitoring Ports Table conf.

```
#########################################################
10.002033788998233
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000001        1       2057     310808          0          2          8          0
0000000000000001        2          2          8          0       1217     183826          0
0000000000000001 fffffffe          0          0          0          0          0          0
10.004288568001357
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000002        1        584     167444          0        452       2980          0
0000000000000002        2        456       3007          0        578     165668          0
0000000000000002 fffffffe          0          0          0          0          0          0
10.005044810000982
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000003        1       1217     183770          0          2          8          0
0000000000000003        2        578     165655          0        455       3000          0
0000000000000003        3        459       3026          0       1225     246972          0
0000000000000003 fffffffe          0          0          0          0          0          0
10.00670888699824
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000004        1       1225     246931          0        459       3026          0
0000000000000004        2        462       3045          0       1225     246931          0
0000000000000004 fffffffe          0          0          0          0          0          0

#########################################################
10.002324218003196
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000004        1       4509     746782          0        519       3890          0
0000000000000004        2        517       3880          0       4509     746782          0
0000000000000004 fffffffe          0          0          0          0          0          0
10.003243855000619
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000001        1       8056    1217672          0          0          0          0
0000000000000001        2          0          0          0       4961     749859          0
0000000000000001 fffffffe          0          0          0          0          0          0
10.004885930000455
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000003        1       4962     749888          0          0          0          0
0000000000000003        2        658     163248          0        523       3915          0
0000000000000003        3        519       3889          0       4510     746741          0
0000000000000003 fffffffe          0          0          0          0          0          0
10.005367830002797
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000002        1        653     161615          0        526       3935          0
0000000000000002        2        522       3909          0        658     163240          0
0000000000000002 fffffffe          0          0          0          0          0          0
```

Figure 2.7: Monitoring Ports Table with active flows

2.2 Remediation

Directly in the PortStatsReply handler, a port blocking policy for the switches has been implemented. Using the monitoring data, the controller checks the throughput of each port on the current switch. The blocking threshold, was chosen based on the critical path of our topology and the number of active hosts on it.

To set the Alarm-State to the port of the switch that overflows the threshold, we created a new dictionary "alarm switch port" represented as follows:

{id switch: port no: [counter threshold overflows, binary counter alarm], ... }

The dictionary is initiated in code showed in Fig. 2.5.
So, in the for cycle for each stat, we can see this conditional code below:

```
        if  (((stat.rx_bytes - previous[stat.port_no][1]) / self.time) > self.threshold  or  ((stat.tx_bytes
- previous[stat.port_no][4]) / self.time) > self.threshold):

            if  self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] < 3:

                self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] =
self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] + 1
        else:
            if  self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] > 0:

                self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] =
self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] - 1



        if  self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] == 3: #blocco della porta
            print(RED + "ALLARME SULLA PORTA " + str(stat.port_no) + " dello Switch " +
str(ev.msg.datapath.id) + RESET)

            self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][1] =  1

            print(self.alarm_switch_port)

            #BLOCCARE FLUSSO
            time.sleep(1)

            self.lock_flow(ev, stat.port_no)

        elif self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] == 2 and
self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][1] == 1:
            print( RED + "ALLARME SULLA PORTA " + str(stat.port_no) + " dello Switch " +
str(ev.msg.datapath.id) + RESET)


        elif self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] == 1 and
self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][1] == 1 : #sblocco della porta
            self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][1] = 0
            self.unlock_flow(ev, stat.port_no)
```

Figure 2.8: Alarm Logic

When the new stats (rx_bytes or tx_bytes) difference with previous stats (of the last monitoring) overflows the threshold in a certain setted *time interval* (10 seconds in this case) on that port_no, the counter (first position of the list for each port) is increased by one. When the counter hits the number 3, the port is blocked with the function *lock_flow* and the binary_alarm_counter is setted to one.

```
def lock_flow (self, ev, port_no):

    ofproto = ev.msg.datapath.ofproto
    parser = ev.msg.datapath.ofproto_parser

    match = parser.OFPMatch(in_port=port_no)

    instructions=[]

    flow_mod = parser.OFPFlowMod(datapath=ev.msg.datapath, priority=2, match=match, instructions=instructions,
command=ofproto.OFPFC_ADD, out_port= ofproto.OFPP_ANY, out_group = ofproto.OFPG_ANY,
flags=ofproto.OFPFF_SEND_FLOW_REM)

    ev.msg.datapath.send_msg(flow_mod)
    print(RED + "Blocked traffic on port %s of switch %s " + RESET, port_no, ev.msg.datapath.id)
```

Figure 2.9: Function Lock Flow

This function, simply creates a new flow rule for the envolved switch (ev.message.datapath.id): with *instructions* = *[]* we are saying that all the incoming packets in port_no should be dropped.

After, when then counter reachs value 1 later an alarm state (so when binary_alarm_counter value is 1), the binary_alarm_counter is setted to 0 and the port unlocked with the function *unlock_flow*, that removes the rule previously created in *lock_flow*, matching the port and the datapath id. (**This solution refers to the section 4.1 in "Capitolo 4"**)

```
def unlock_flow(self, ev, port_no):

    ofproto = ev.msg.datapath.ofproto
    parser = ev.msg.datapath.ofproto_parser

    match = parser.OFPMatch(in_port=port_no)

    flow_mod = parser.OFPFlowMod(datapath=ev.msg.datapath, priority=2, match=match,
command=ofproto.OFPFC_DELETE, out_port= ofproto.OFPP_ANY, out_group = ofproto.OFPG_ANY,
flags=ofproto.OFPFF_SEND_FLOW_REM)

    ev.msg.datapath.send_msg(flow_mod)

    print(GREEN + "Unlocked traffic on port %s of switch %s" + RESET , port_no, ev.msg.datapath.id)
```

Figure 2.10: Unlock Function

In the following images are showed the archieved results.

```
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000001        1       8033    1011046          0          0          0          0
ALLARME SULLA PORTA 1 dello Switch 1
Blocked traffic on port %s of switch %s  1 1
0000000000000001        2          0          0          0       4964     624928          0
0000000000000001 fffffffe          0          0          0          0          0          0
13.013066748997517
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000003        1       4964     576771          0          0          0          0
0000000000000003        2        136      19489          0        112        780          0
0000000000000003        3        112        780          0       4933     576522          0
0000000000000003 fffffffe          0          0          0          0          0          0
13.014262538999901
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000002        1        136      19487          0        112        780          0
0000000000000002        2        112        780          0        136      19487          0
0000000000000002 fffffffe          0          0          0          0          0          0
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 3 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 2 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 2

########################################################
10.001733047000016
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000001        1       1622     243881          0          0          0          0
ALLARME SULLA PORTA 1 dello Switch 1
0000000000000001        2          0          0          0       1998     301898          0
0000000000000001 fffffffe          0          0          0          0          0          0
10.004198613998597
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000002        1          7       1059          0          6         51          0
0000000000000002        2          6         51          0          7       1059          0
0000000000000002 fffffffe          0          0          0          0          0          0
10.004429434997292
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000004        1       3000     453106          0          5         46          0
ALLARME SULLA PORTA 1 dello Switch 4
0000000000000004        2          5         46          0        501      75571          0
ALLARME SULLA PORTA 2 dello Switch 4
0000000000000004 fffffffe          0          0          0          0          0          0
10.004609080999217
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000003        1       1998     301811          0          0          0          0
0000000000000003        2          7       1059          0          6         51          0
0000000000000003        3          5         46          0       2999     452947          0
0000000000000003 fffffffe          0          0          0          0          0          0

########################################################
10.003845654999168
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000001        1         10         41          0          0          0          0
Unlocked traffic on port %s of switch %s 1 1
0000000000000001        2          0          0          0          0          0          0
0000000000000001 fffffffe          0          0          0          0          0          0
10.005442178000521
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000003        1          0          0          0          0          0          0
0000000000000003        2          1        151          0          0          0          0
0000000000000003        3          0          0          0          1        151          0
0000000000000003 fffffffe          0          0          0          0          0          0
10.006378917998518
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000002        1          1        151          0          0          0          0
0000000000000002        2          0          0          0          1        151          0
0000000000000002 fffffffe          0          0          0          0          0          0
10.007284408999112
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------    --------   --------   --------   --------   --------   --------
0000000000000004        1          1        151          0          0          0          0
Unlocked traffic on port %s of switch %s 1 4
0000000000000004        2          0          0          0          0          0          0
Unlocked traffic on port %s of switch %s 2 4
0000000000000004 fffffffe          0          0          0          0          0          0
```

Figure 2.11: Lock and Unlock Ports

Before the unlock:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
```

Figure 2.12: Ping after lock function.

After the unlock:

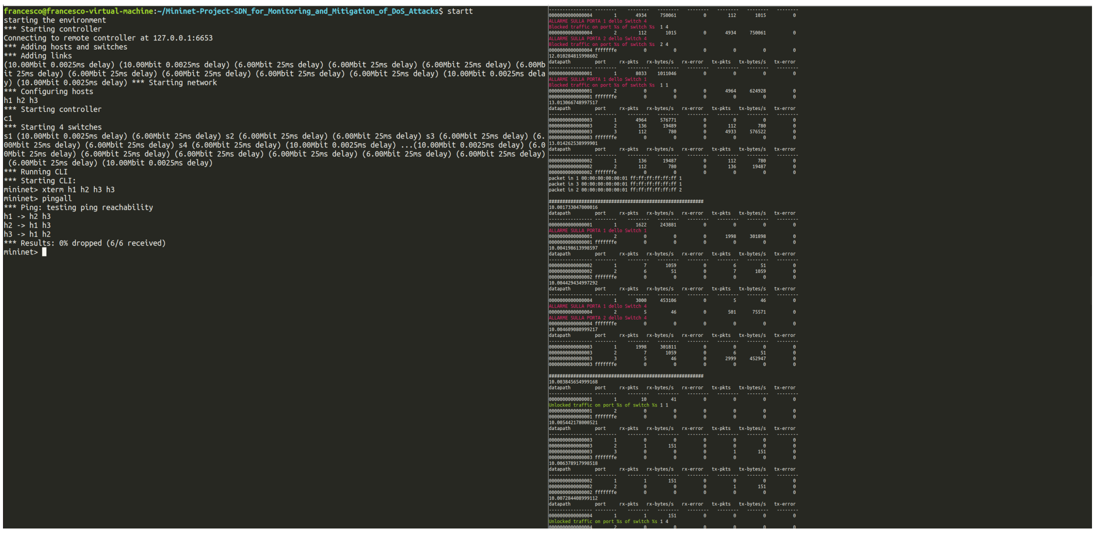


Figure 2.13: Ping after unlock function.

Chapter 3

Integration & Execution

3.1 Context of the problem

All that remains is to integrate the monitoring and port-blocking technique implemented and demonstrated in the previous paragraphs. Before presenting the results obtained, it is better to summarize the situation by recapping the context in which the DoS attack occurs.

The topology, which has been implemented according to the code shown in Chapter 1, is as follows:

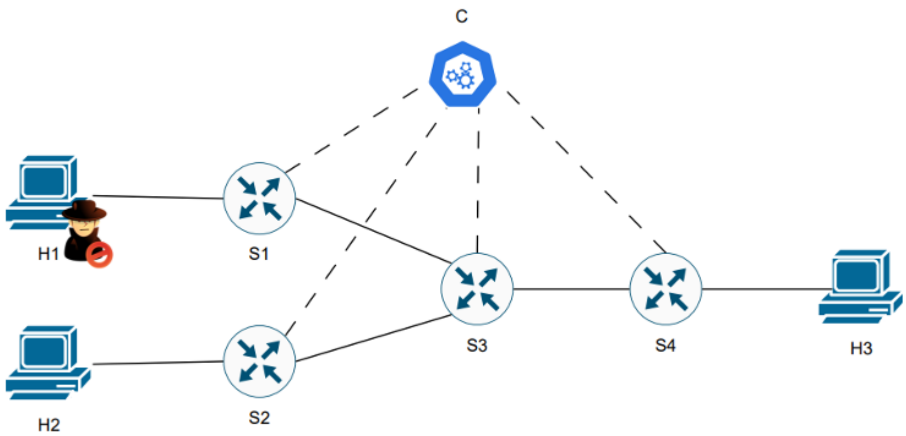


Figure 3.1: Topology of the Problem

The three Hosts shown in the figure have different roles: **H1** will act as the attacker, launching the DoS attack by sending a number of UDP packets that will congest the network, simulating a continuous flow of 9Mbps packets. **H2** will be a normal user of our network, sending 3Mbps of TCP traffic. The server will be simulated by host **H3**, which will provide a UDP connection on port 5001 and a TCP connection on port 5002. It is important to note that the critical path of our topology is located between switch S3 and switch S4, with the link between the two components having a bandwidth of 7Mbps, which is insufficient for both hosts.

3.2 Results

Two different situations are shown: the first one involves applying a partial remedy. When the controller detects abnormal traffic on a port, it will block all incoming packets on that port.

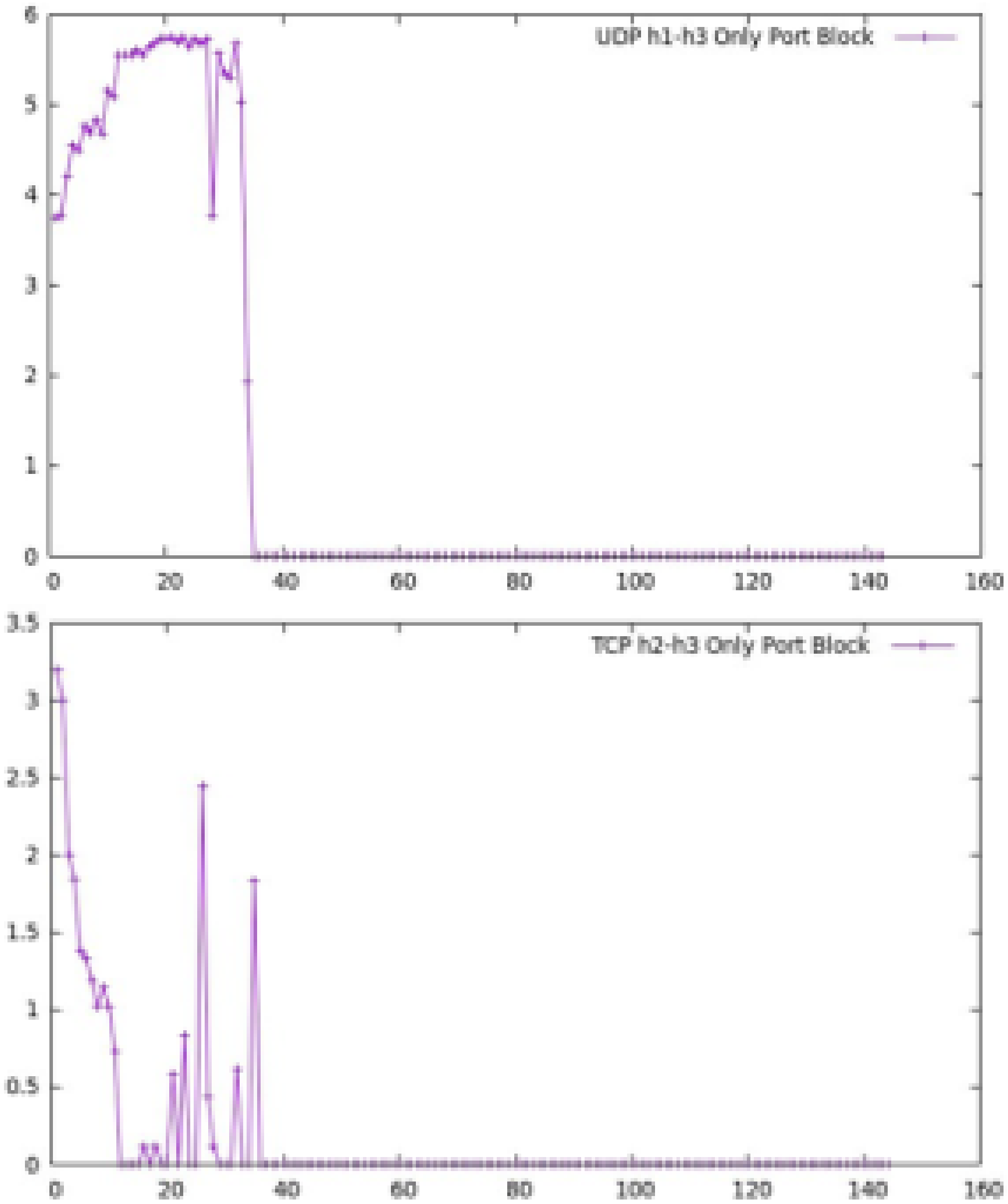


Figure 3.2: UDP with Port Block

The second graph also shows the unlocking of H2's port while host H1 has stopped attacking the server.

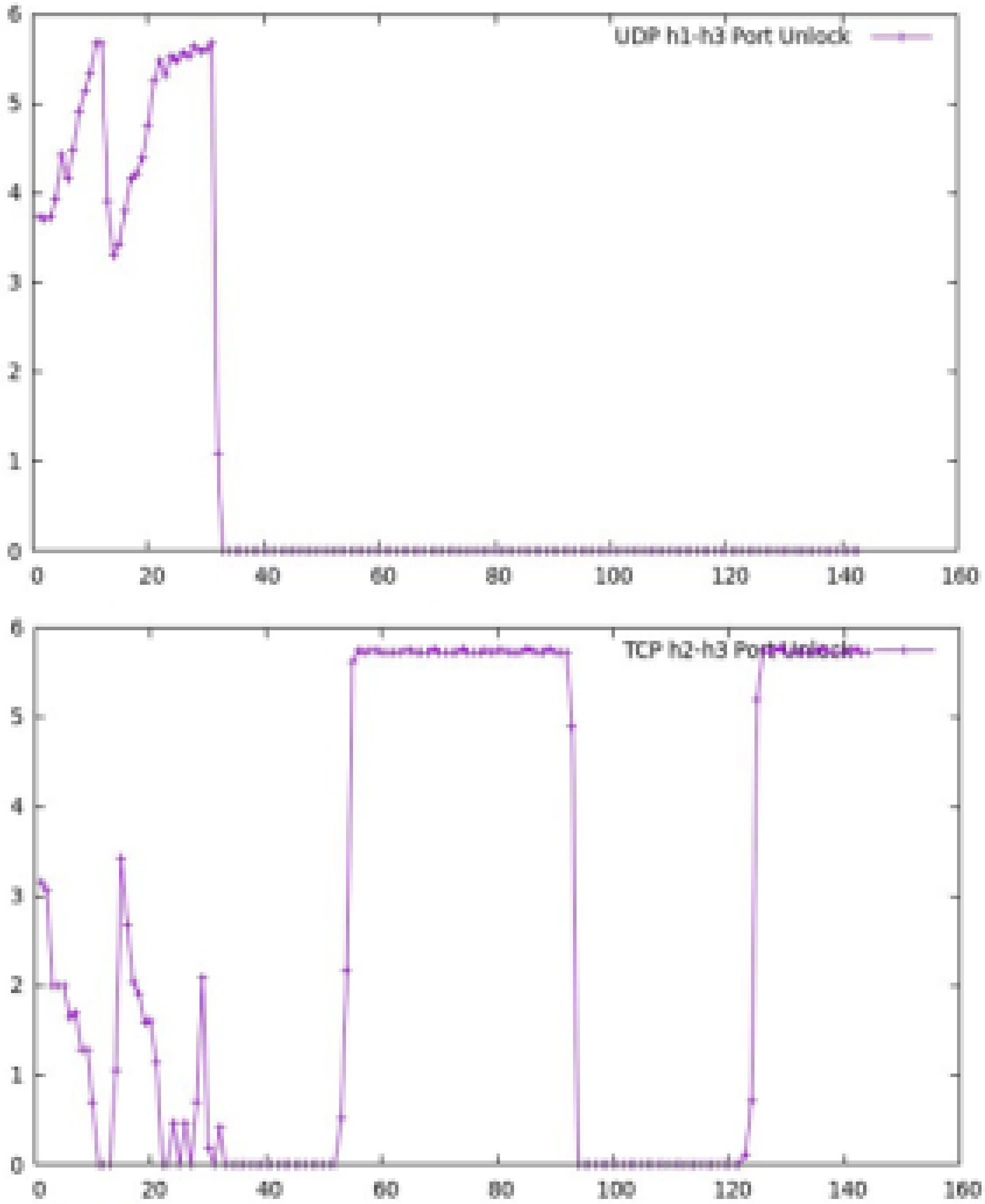


Figure 3.3: TCP with Port Block

Chapter 4

Optional Features

4.1 Dynamic Remediation Mechanism

4.2 Minimizing Collateral Impact on Legitimate Hosts

In the updated network topology, a new host (H4) has been added and is connected to switch S1, generating traffic directed towards H3.

```
self.h4 = self.net.addHost('h4', mac='00:00:00:00:00:04', ip= '10.0.0.4')

...

self.net.addLink(self.h4, self.s1, bw=10, delay='0.0025ms')
```

Figure 4.1: Small changes in topology

This change introduces an additional source of traffic within the network. If we apply a rule to block all traffic from the port connecting S3 to S1, the impact on the network would be significant. Specifically, **legitimate traffic from both H1 and H4, destined for H3, would be disrupted** since their traffic routes through S3.

This scenario illustrates the importance of carefully considering the placement and scope of traffic blocking rules to minimize collateral damage to legitimate hosts while addressing security concerns, such as in the case of the piracy shield.

The solution to the problem of selectively blocking malicious traffic in the network involves **identifying the MAC address of the attacking host** and then applying **fine-grained blocking to that specific address**.

This approach is implemented by **monitoring the flow statistics** of the network and setting up an **alarm system** based on the amount of traffic being generated.

```python
# Richiedi le statistiche di flusso da tutti gli switch
req = parser.OFPFlowStatsRequest(datapath)
datapath.send_msg(req)
```

Figure 4.2: Flow statistics request code snippet

In the *_request_stats* function of the monitor, we added the request for flow statistics from all switches.

The **event handler for the Flow Stats Reply** is the most crucial part of the entire implementation. Therefore, an analysis of the relevant parts of the code will now follow.

```python
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def _flow_stats_reply_handler(self, ev):
        body = ev.msg.body
        self.rec_res = time.perf_counter()

        # Calculate the time elapsed since the request was sent
        self.time = timeInterval + (self.rec_res - self.send_req)
        print(self.time)

        # Filter flows
        low_priority_flows = sorted(
            (flow for flow in body if flow.priority == 1),
            key=lambda flow: (flow.match['in_port'],
flow.match.get('eth_src'), flow.match.get('eth_dst'))
        )

        # If there are no flows, exit without updating flow structures
        if len(low_priority_flows) == 0:
            return
```

Figure 4.3: First part of Event Handler

The *_flow_stats_reply_handler* method is triggered when the controller receives an *OFPFlowStatsReply* message. Upon receiving this message, the **current time** is recorded to calculate the elapsed time since the request was sent. The code then filters out **flows with priority 1** and sorts them based on their input port (*in_port*), **source MAC address** (*eth_src*), and destination MAC address (*eth_dst*).

If no such flows are found, the function exits early to avoid unnecessary processing.

```
        if ev.msg.datapath.id not in self.flow_stats.keys():
            self.logger.info('datapath  in-port   eth-src    eth-dst  out-port   packets   bytes/s')
            self.logger.info('---------------- -------- ----------------- ----------------- --------   --------  ')

            for stat in low_priority_flows:
                self.logger.info('%016x %8x %17s %17s %8x %8d %8d',
                                 ev.msg.datapath.id, stat.match['in_port'], stat.match['eth_src'], stat.match['eth_dst'],
                                 stat.instructions[0].actions[0].port, stat.packet_count,
                                 stat.byte_count / self.time)

            # Initialize the alarm counter for each flow
            self.alarm_flow[ev.msg.datapath.id] = {
                # First field is the counter, second field indicates whether the alarm is raised
                (stat.match['in_port'], stat.match.get('eth_src'), stat.match.get('eth_dst')): [0, 0]
                for stat in low_priority_flows
            }

            # Update the dictionary with the new statistics
            self.flow_stats[ev.msg.datapath.id] = {
                (stat.match['in_port'], stat.match.get('eth_src'), stat.match.get('eth_dst')): [stat.packet_count,
stat.byte_count]
                for stat in low_priority_flows
            }
```

Figure 4.4: Initializing and Logging Flow Statistics

This block is executed if the switch's *datapath.id* is not already present in the *flow_stats* dictionary. The method logs the **headers** and **flow information**, including the **number of packets** and **bytes per second**. Since this is the first time the switch's data is being processed, the **alarm dictionary is initialized**, associating each flow with a **counter** and an **alarm state** (both initially set to 0). Finally, the *flow_stats dictionary* is updated with the latest packet and byte counts for all flows.

If flow statistics for the switch are already recorded, the code first **retrieves the previous statistics** for comparison. It then calculates the **differences** in packet and byte counts (*packet_diff* and *byte_diff*) for each flow.

If a flow is not already present in the *alarm_flow* structure, it is identified as new and added accordingly.

```
else:
    previous = self.flow_stats[ev.msg.datapath.id]
    self.logger.info('datapath         in-port  eth-src           eth-ds      out-port   packets    bytes/s')
    self.logger.info('---------------- -------- ----------------- ----------------- --------   --------  ')

    for stat in low_priority_flows:
        in_port = stat.match['in_port']
        eth_src = stat.match.get('eth_src')
        eth_dst = stat.match.get('eth_dst')
        out_port = stat.instructions[0].actions[0].port
        packet_diff = stat.packet_count - previous.get((in_port, eth_src, eth_dst), [0, 0])[0]
        byte_diff = stat.byte_count - previous.get((in_port, eth_src, eth_dst), [0, 0])[1]

        self.logger.info('%016x %8x %17s %17s %8x %8d %8d',
                         ev.msg.datapath.id, in_port, eth_src, eth_dst, out_port,
                         packet_diff, byte_diff / self.time)

        if (in_port,eth_src,eth_dst) not in self.alarm_flow[ev.msg.datapath.id]:
            # If the flow is not in the structure, add it
            self.alarm_flow[ev.msg.datapath.id][(in_port,eth_src,eth_dst)] = [0,0]
```

Figure 4.5: Retrieving previous statistics

```
        # Alarm management
        if (byte_diff/self.time) > self.threshold: #If bytes per second exceed the threshold
            if self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][0] < 3:
                self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][0] += 1
        else:
            if self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][0] > 0:
                self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][0] -= 1

        # If the alarm counter reaches 3, block the flow
        if self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][0] == 3:
            self.logger.warning(f"ALLARME: L'host {eth_src} sta inviando troppi byte/s. Blocca il flusso.")
            self.lock_flow(ev, eth_src, eth_dst, in_port)

        elif self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][0] == 2 and
self.alarm_flow[ev.msg.datapath.id][(in_port, eth_src, eth_dst)][1] >= 1:
            self.logger.info(f"Monitoraggio: L'host {eth_src} sta tornando normale.")
```

Figure 4.6: Alarm Management

The **alarm management section** of the code is designed to monitor the **flow's behavior over time**, specifically how often it exceeds a defined threshold of bytes per second. The **counter** serves to track how many consecutive intervals the flow has surpassed this threshold. If the flow exceeds the threshold for **three consecutive intervals** (e.g., 30 seconds if each interval is 10 seconds), the flow is **blocked** to prevent potential network congestion or abuse.

Conversely, if the flow's behavior **normalizes** —meaning it falls below the threshold for two consecutive intervals— the counter is decreased, which eventually leads to **unblocking** the flow.

This mechanism ensures that **flows are only blocked if they consistently exhibit abnormal behavior**, while also **allowing for recovery** if the traffic stabilizes.

```
def lock_flow(ev, src, dst, port):
    ofproto = ev.msg.datapath.ofproto
    parser = ev.msg.datapath.ofproto_parser

    # Create a match rule based on the source MAC address
    match = parser.OFPMatch(eth_src=src)

    # Empty instructions list to drop the traffic
    instructions = []

    # Create a flow modification message to add a rule that blocks the matching flow
    flow_mod = parser.OFPFlowMod(
        datapath=ev.msg.datapath, priority=2, match=match, instructions=instructions,
        command=ofproto.OFPFC_ADD, out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY,
        flags=ofproto.OFPFF_SEND_FLOW_REM
    )

    # Send the flow modification message to the switch
    ev.msg.datapath.send_msg(flow_mod)
    # Print a message indicating the flow has been blocked
    print(RED + "Blocked traffic of flow %s of switch %s " + RESET, src, ev.msg.datapath.id)

    # Increment the alarm status to indicate the flow has been blocked
    self.alarm_flow[ev.msg.datapath.id][(port, src, dst)][1] += 1

    # Start a thread to unlock the flow after a delay
    t = Thread(target=unlock_func, args=(ev, src, self.alarm_flow[ev.msg.datapath.id][(port, src, dst)][1]))
    t.start()
```

Figure 4.7: Block function

When an alarm is triggered, the ***lock_function*** function is executed to **block traffic** from the offending source. This function creates a flow rule with a higher priority, ensuring that **any packets matching the source MAC address are dropped**.

Additionally, the system keeps track of the number of times a flow has been blocked, **updating the alarm status** accordingly.

```python
def unlock_func(ev, src, value):
    # Calculate the wait time based on the number of times the flow has been blocked
    wait = pow(7, value)

    print("Source " + str(src) + " blocked for " + str(wait) + " seconds on Switch " +  str(ev.msg.datapath.id))
    # Sleep for the calculated wait time
    time.sleep(wait)

    ofproto = ev.msg.datapath.ofproto
    parser = ev.msg.datapath.ofproto_parser

    # Create a match rule based on the source MAC address
    match = parser.OFPMatch(eth_src=src)

    # Create a flow modification message to delete the blocking rule
    flow_mod = parser.OFPFlowMod(
        datapath=ev.msg.datapath,
        priority=2,
        match=match,
        command=ofproto.OFPFC_DELETE,
        out_port=ofproto.OFPP_ANY,
        out_group=ofproto.OFPG_ANY,
        flags=ofproto.OFPFF_SEND_FLOW_REM
    )

    ev.msg.datapath.send_msg(flow_mod)
    print(GREEN + "Unlocked traffic of flow source %s of switch %s" + RESET, src, ev.msg.datapath.id)
```

Figure 4.8: Unlock Thread Function

The ***unlock_func*** function is responsible for eventually **lifting the block on the flow after a certain delay**.

The **delay** is calculated based on the **number of times the flow has been previously blocked**, with the wait time increasing **exponentially**. Once the wait time has passed, the function sends a new flow modification message to the switch, instructing it to **delete the blocking rule**, thus allowing traffic from the source MAC address to resume. This method ensures that legitimate traffic is not permanently blocked while still protecting the network from sustained attacks.

To optimize the system's responsiveness and efficiency, the *unlock_func* is executed in a separate thread. This **multithreaded approach** offers significant advantages. By handling the unblock operation in a dedicated thread, the main flow monitoring and blocking processes remain unaffected, allowing the system to continue monitoring and reacting to network events without interruption. This ensures that the system can **handle multiple flows and alarms simultaneously**, improving **scalability** and **performance**. Additionally, the use of threads allows the system to **manage the timing of unblock operations more precisely**, without blocking or delaying other critical tasks.

The ***alarm_flow*** structure is the key component that enables us to monitor and manage network traffic effectively, particularly in **identifying** and **controlling malicious flows**. This structure acts as a comprehensive **record of all active flows** within the network, organized by the unique identifier of each switch (*datapath ID*).

Within each switch, *alarm_flow* stores entries corresponding to specific flows, identified by their **input port** (*in_port*), **source MAC address** (*eth_src*), and **destination MAC address** (*eth_dst*). Each of these flow entries maintains a pair of values: a **counter** that tracks the number of times the flow has triggered an alarm and an **alarm_status** that indicates whether the flow is currently blocked.

```
alarm_flow
│
├── [datapath_id_1]           # Switch ID (datapath ID)
│   ├── (in_port_1, eth_src_1, eth_dst_1): [counter, alarm_status]
│   ├── (in_port_2, eth_src_2, eth_dst_2): [counter, alarm_status]
│   └── ...
│
├── [datapath_id_2]           # Another switch (datapath ID)
│   ├── (in_port_1, eth_src_1, eth_dst_1): [counter, alarm_status]
│   ├── (in_port_2, eth_src_2, eth_dst_2): [counter, alarm_status]
│   └── ...
│
└── ...
```

Figure 4.9: alarm_flow structure

By continuously updating this structure, the system can detect when a flow exceeds predefined thresholds for traffic volume, triggering an alarm. Conversely, when the flow returns to normal behavior, the *alarm_flow* structure facilitates the automatic unblocking of the flow, ensuring that legitimate traffic can resume without manual intervention.

We now proceed to the attack experiment, where standard TCP flows are generated from hosts h4 and h2, and a UDP flow with a substantial volume of data is initiated from host h1.

```
datapath         in-port   eth-src           eth-dst           out-port   packets   bytes/s
---------------- -------- ----------------- ----------------- --------   --------  --------
0000000000000004        1 00:00:00:00:00:01 00:00:00:00:00:03        2       237    35807
0000000000000004        1 00:00:00:00:00:02 00:00:00:00:00:03        2       380   107848
0000000000000004        1 00:00:00:00:00:04 00:00:00:00:00:03        2       650   187581
0000000000000004        2 00:00:00:00:00:03 00:00:00:00:00:01        1         0        0
0000000000000004        2 00:00:00:00:00:03 00:00:00:00:00:02        1       370     2443
0000000000000004        2 00:00:00:00:00:03 00:00:00:00:00:04        1       651     4294
10.009728535995237
datapath         port      rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------   --------   --------   --------   --------   --------   --------
0000000000000003        1       1178     272147          0        654       4306          0
0000000000000003        2        432     122052          0        372       2449          0
0000000000000003        3       1025       6756          0       1288     333896          0
0000000000000003 fffffffe          0          0          0          0          0          0
10.009942445998604
datapath         in-port   eth-src           eth-dst           out-port   packets   bytes/s
---------------- -------- ----------------- ----------------- --------   --------  --------
0000000000000003        1 00:00:00:00:00:01 00:00:00:00:00:03        3       470    70993
0000000000000003        1 00:00:00:00:00:04 00:00:00:00:00:03        3       687   198269
0000000000000003        2 00:00:00:00:00:02 00:00:00:00:00:03        3       428   121446
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:01        1         0        0
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:02        2       368     2429
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:04        1       651     4293
```

Figure 4.10: Information log of standard TCP flows

This is the information log from the alarm management section during the experiment, which clearly shows that **only the standard TCP flows are active**. Notably, the traffic originating from the MAC address 00:00:01, associated with host h1, is recorded as null, indicating that it has been effectively blocked.

As the experiment progresses, the malicious UDP flow from h1 is introduced. This flow, characterized by an unusually high data rate, quickly surpasses the predefined traffic threshold within 30 seconds, triggering the alarm system. The system accurately identifies the excessive traffic and responds by **blocking the source**, specifically **targeting the MAC address associated with h1**. This blocking action is implemented across multiple switches through the automatic addition of the blocking rule.

The effectiveness of the system is further demonstrated when, **after the flow from h1 returns to a normal traffic level, the block is lifted**, and the MAC address is once again allowed to transmit. This dynamic process, which involves both the detection and subsequent unblocking of the source, ensures that the network remains secure while minimizing the impact on legitimate traffic. The sequence of events, including the precise identification, blocking, and eventual unblocking of the MAC address, is illustrated in Figure 4.11.

```
0000000000000001        1 00:00:00:00:00:01 00:00:00:00:00:03        3     8041  1215061
ALLARME: L'host 00:00:00:00:00:01 sta inviando troppi byte/s. Blocca il flusso.
Blocked traffic of flow %s of switch %s  00:00:00:00:00:01 1
Source 00:00:00:00:00:01 bloccata per 7 secondi sullo Switch 1
0000000000000001        2 00:00:00:00:00:04 00:00:00:00:00:03        3        1      151
0000000000000001        3 00:00:00:00:00:03 00:00:00:00:00:01        1        0        0
0000000000000001        3 00:00:00:00:00:03 00:00:00:00:00:04        2        0        0
10.007110616999853
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------   --------   --------   --------   --------   --------   --------
0000000000000003        1       4964     750023          0          0          0          0
0000000000000003        2        116      22896          0        108        988          0
0000000000000003        3        108        988          0       4932     749833          0
0000000000000003 fffffffe          0          0          0          0          0          0
10.007300352001039
datapath         in-port   eth-src           eth-dst           out-port   packets   bytes/s
---------------- -------- ----------------- ----------------- --------   --------  --------
0000000000000003        1 00:00:00:00:00:01 00:00:00:00:00:03        3     4971   751066
ALLARME: L'host 00:00:00:00:00:01 sta inviando troppi byte/s. Blocca il flusso.
Blocked traffic of flow %s of switch %s  00:00:00:00:00:01 3
Source 00:00:00:00:00:01 bloccata per 7 secondi sullo Switch 3
0000000000000003        1 00:00:00:00:00:04 00:00:00:00:00:03        3        0        0
0000000000000003        2 00:00:00:00:00:02 00:00:00:00:00:03        3      118    22909
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:01        1        0        0
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:02        2      107      978
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:04        1        0        0
10.008347251001396
datapath         port     rx-pkts   rx-bytes/s   rx-error   tx-pkts   tx-bytes/s   tx-error
---------------- --------   --------   --------   --------   --------   --------   --------
0000000000000002        1        116      22893          0        107        978          0
0000000000000002        2        108        988          0        116      22893          0
0000000000000002 fffffffe          0          0          0          0          0          0
10.008526096004061
datapath         in-port   eth-src           eth-dst           out-port   packets   bytes/s
---------------- -------- ----------------- ----------------- --------   --------  --------
0000000000000002        1 00:00:00:00:00:02 00:00:00:00:00:03        2      118    22906
0000000000000002        2 00:00:00:00:00:03 00:00:00:00:00:02        1      107      978
Unlocked traffic of flow source %s of switch %s 00:00:00:00:00:01 1
Unlocked traffic of flow source %s of switch %s 00:00:00:00:00:01 3
packet in 1 00:00:00:00:00:01 00:00:00:00:00:03 1
packet in 1 00:00:00:00:00:01 00:00:00:00:00:03 1
```

Figure 4.11: Locking and unlocking the h1 flow

To verify the effectiveness of the MAC address blocking mechanism, we conducted a ***pingall test*** across the entire network. This comprehensive test attempts to ping every host from every other host, ensuring that connectivity is functioning as expected. The results from this pingall test provided clear **confirmation that our solution was working as intended.**

Specifically, host **h1**, whose MAC address had been blocked due to its malicious activity, **was unable to communicate** with any other hosts in the network. Moreover, no other hosts were able to reach h1. This **isolation** of h1 demonstrates that the MAC address block was successfully enforced, effectively neutralizing the potential threat while maintaining the integrity of the network for all other legitimate traffic.

```
ALLARME: L'host 00:00:00:00:00:01 sta inviando troppi byte/s. Blocca il flusso.
Blocked traffic of flow %s of switch %s  00:00:00:00:00:01 3
Source 00:00:00:00:00:01 bloccata per 343 secondi sullo Switch 3
0000000000000003        1 00:00:00:00:00:04 00:00:00:00:00:02        2        0        0
0000000000000003        1 00:00:00:00:00:04 00:00:00:00:00:03        3        1      151
0000000000000003        2 00:00:00:00:00:02 00:00:00:00:00:01        1        0        0
0000000000000003        2 00:00:00:00:00:02 00:00:00:00:00:03        3      235    53441
0000000000000003        2 00:00:00:00:00:02 00:00:00:00:00:04        1        0        0
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:01        1        0        0
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:02        2      114     1025
0000000000000003        3 00:00:00:00:00:03 00:00:00:00:00:04        1        2       18
packet in 2 0e:7d:b9:b0:21:b8 33:33:00:00:00:02 2

##########################################################
10.012706410998362
datapath         port     rx-pkts  rx-bytes/s  rx-error  tx-pkts  tx-bytes/s  tx-error
---------------- -------- -------- --------    --------  -------- --------    --------
0000000000000004        1     3011    721907         0      1939     12986         0
0000000000000004        2     1945     13026         0      3011    721907         0
0000000000000004 fffffffe        0         0         0         0         0         0
10.01322818799963
datapath         in-port  eth-src           eth-dst           out-port  packets  bytes/s
---------------- -------- ----------------- ----------------- --------  -------- --------
0000000000000004        1 00:00:00:00:00:01 00:00:00:00:00:03        2      1217   183767
0000000000000004        1 00:00:00:00:00:02 00:00:00:00:00:03        2      1439   405134
0000000000000004        1 00:00:00:00:00:04 00:00:00:00:00:03        2       473   134711
0000000000000004        2 00:00:00:00:00:03 00:00:00:00:00:01        1         0        0
0000000000000004        2 00:00:00:00:00:03 00:00:00:00:00:02        1      1376     9233
0000000000000004        2 00:00:00:00:00:03 00:00:00:00:00:04        1       460     3079
```

The screen capture of this test further solidifies our confidence in the solution's capability to secure the network against such attacks.

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

Figure 4.12: Pingall after lock

Following the **unblocking** of h1's MAC address, we conducted another ***pingall test*** to ensure that h1's connectivity was fully restored. The results confirmed that, after the lifting of the block, h1 **regained its original reachability** within the network. Host h1 was once again able to successfully communicate with all other hosts, and vice versa, without any issues.

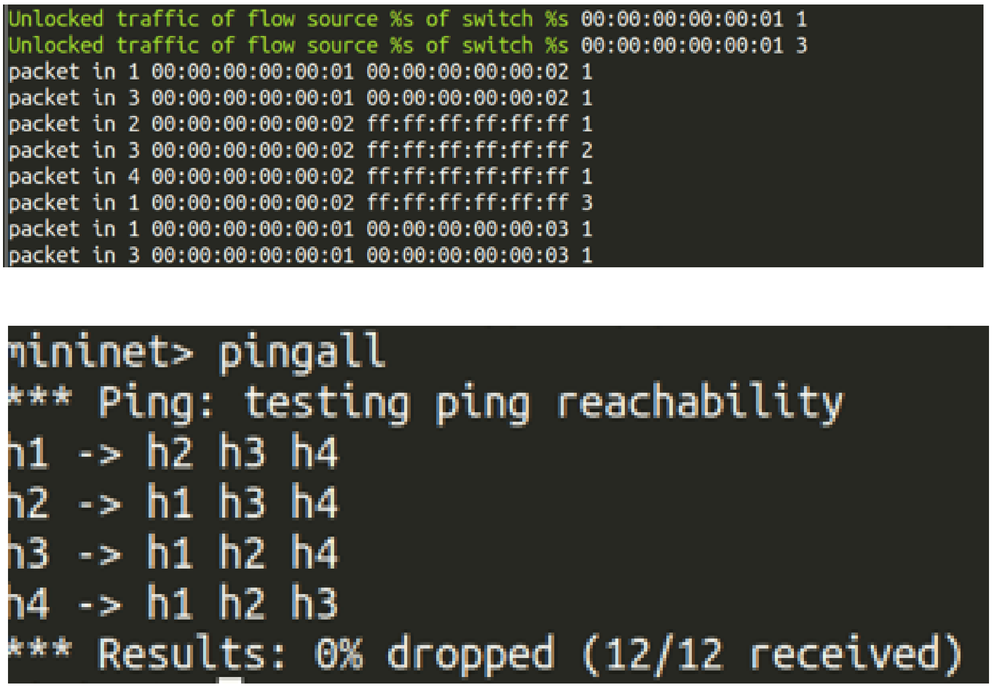


Figure 4.13: Pingall after lock

This demonstrates that our solution not only effectively blocks malicious activity but also accurately restores normal network functionality once the threat subsides, ensuring minimal disruption to legitimate traffic.

4.3 Telegram Bot

For a better user experience, we've created a Telegram Bot to simplify the access to useful informations. With this bot, the user can:

1. Require an image of current topology,
2. Require the counter and alarm_status stats of each flow, for each switch,
3. Receive messages when a specific MAC address is blocked and afterwards unlocked.

In this final section, screenshots will be shown illustrating interactions between a potential administrator and the Bot. The images focus on the two main types of requests an admin can make: querying the network topology and retrieving specific switch statistics. The last image captures a warning message triggered by a DoS attack and the corresponding resolution steps.

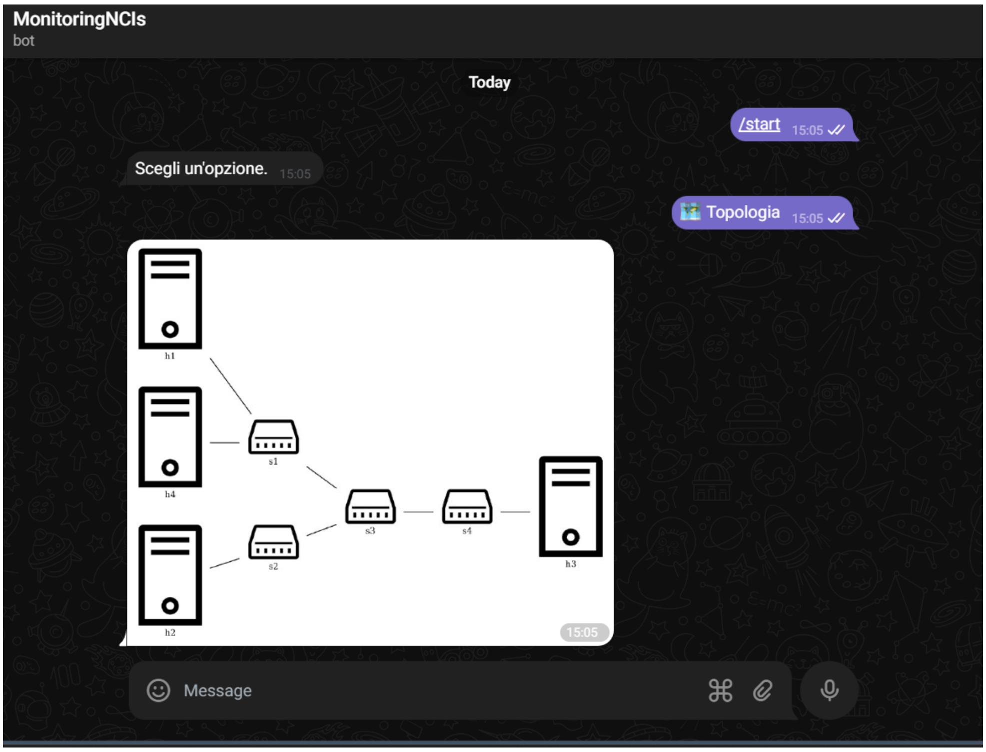


Figure 4.14: Bot Topology

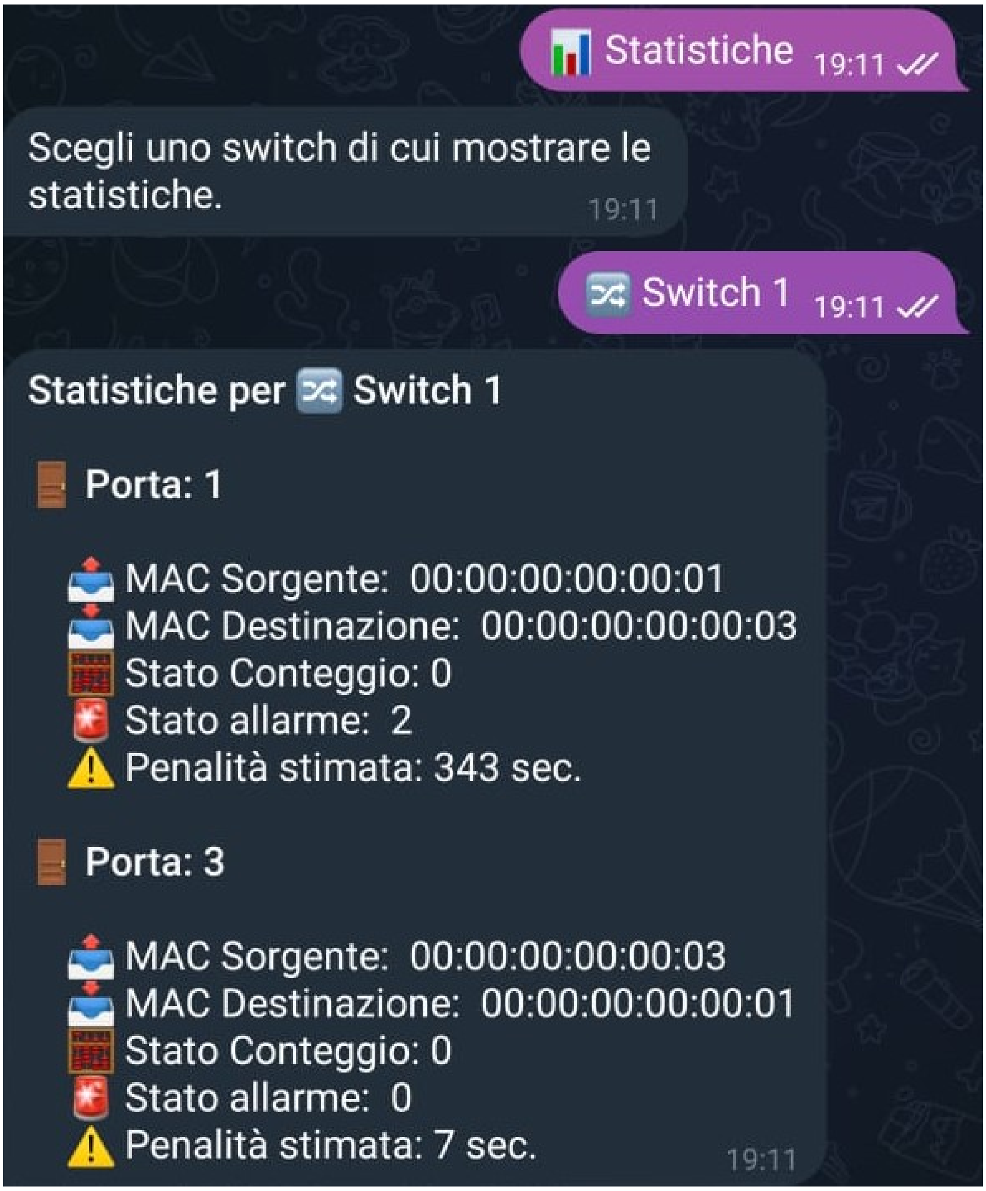


Figure 4.15: Bot Switch Stats

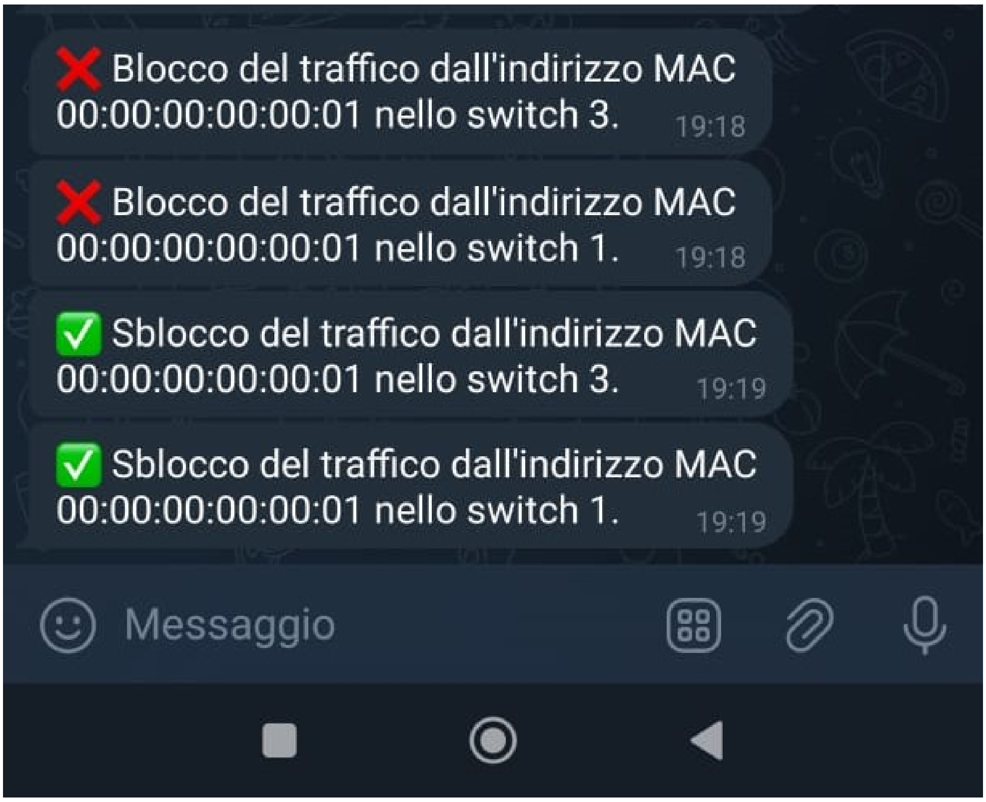


Figure 4.16: Bot Warning Messages