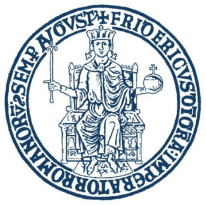

UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

Elaborato di Networks and Cloud Infrastructure

Anno Accademico 2024/2025

Professori:

Prof. Giorgio Ventre
Prof. Alessio Botta
Prof. Roberto Canonico

Studenti:

Matteo Ciliberti (DE9000023)
Francesco Riccio (M63001643)

Contents

1 Introduzione **2**

1.1 Topologia 2

1.2 Punti deboli 4

2 Over blocking **5**

2.1 Monitoraggio 5

2.2 Rilevamento delle anomalie 5

2.3 Calcolo della penalty 6

2.4 Blocco 7

2.5 Recovery 8

3 Debolezze architetturali **10**

3.1 Controller-Centric Blocking Decisions 10

3.1.1 Shared Data Structure 10

3.1.2 Multi-Source Blocking 11

3.1.3 External Policy Integration 11

3.2 Modular Architecture Design 11

3.2.1 Traffic Monitor 12

3.2.2 Policy Engine 12

3.2.3 Flow Enforcer 12

3.2.4 Decoupled Communication 12

4 Simulazione **13**

4.1 Topologia semplice 13

4.2 Topologia allargata 17

1 Introduzione

Il seguente testo si propone di risolvere alcune delle lacune presenti in una rete basata su SDN precedentemente implementata. La rete è stata virtualizzata tramite **Mininet**, con un controller basato sul framework **Ryu**, che gestisce gli switch attraverso il protocollo **OpenFlow 1.3**.

1.1 Topologia

La topologia implementata presenta quattro host (*h1, h2, h3, h4*), ciascuno dotato di un indirizzo IP univoco all'interno della stessa sottorete, e quattro switch (*s1, s2, s3, s4*).

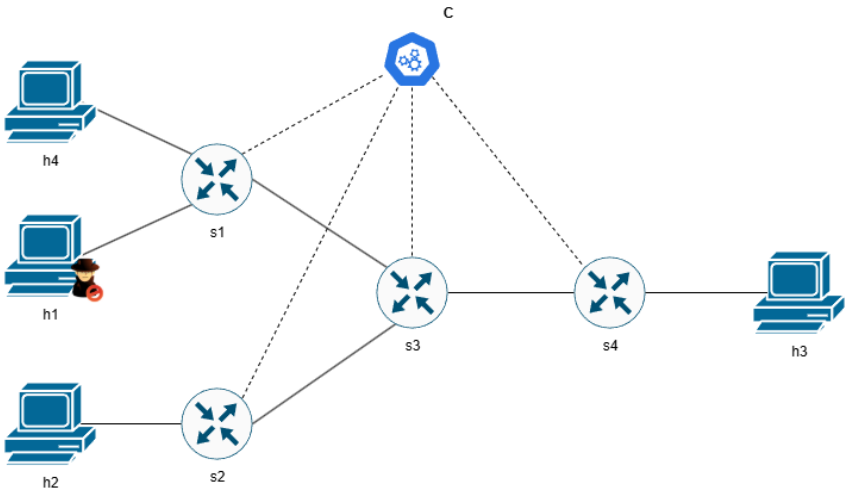

Figure 1: Topologia semplice

Gli host *h1* e *h4* sono collegati allo switch di accesso *s1*, mentre *h2* e *h3* sono rispettivamente collegati agli switch *s2* e *s4*. Gli switch stessi sono interconnessi tramite lo switch centrale *s3*, che funge da nodo di aggregazione e garantisce la comunicazione tra tutti gli host.

I collegamenti tra gli host e i rispettivi switch di accesso sono stati configurati con una larghezza di banda costante di **10 Mbps** e un ritardo di **2 ms**, mentre i collegamenti tra gli switch presentano una larghezza di banda maggiore, pari a **15 Mbps**, e un ritardo di **25 ms**.

```
info("*** Starting controller\n")
self.net.addController('C1', controller = RemoteController, port = 6633)

info("*** Adding hosts\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.h4 = self.net.addHost('h4', mac='00:00:00:00:00:04', ip='10.0.0.4')

info("*** Adding switches\n")
self.s1 = self.net.addSwitch('s1')
self.s2 = self.net.addSwitch('s2')
self.s3 = self.net.addSwitch('s3')
self.s4 = self.net.addSwitch('s4')

info("*** Adding links\n")
# Collegamenti host - switch
self.net.addLink(self.h1, self.s1, bw = 50, delay = '2ms')
self.net.addLink(self.h4, self.s1, bw = 50, delay = '2ms')
self.net.addLink(self.h2, self.s2, bw = 50, delay = '2ms')
self.net.addLink(self.h3, self.s4, bw = 50, delay = '2ms')

# Collegamenti tra switch
self.net.addLink(self.s1, self.s3, bw = 50, delay = '25ms')
self.net.addLink(self.s2, self.s3, bw = 50, delay = '25ms')
self.net.addLink(self.s4, self.s3, bw = 50, delay = '25ms')

info("*** Starting network\n")
self.net.build()
self.net.start()
```

Figure 2: Implementazione .py della topologia di rete

Per semplicità, il controller viene containerizzato in un container Docker, in modo da evitare problematiche legate alle dipendenze tra Ryu e Python.

```
FROM python:3.9-slim

RUN
 apt-get update && apt-get install -y build-essential && rm -rf /var/lib/apt/lists/*

RUN pip install ryu eventlet==0.30.2 dnspython==1.16.0

WORKDIR /app

COPY controller.py /app/controller.py
COPY host_info.json /app/host_info.json
COPY shared/ /app/shared/
COPY monitoring/ /app/monitoring/
COPY policies/ /app/policies/
COPY enforcement/ /app/enforcement/

CMD ["ryu-manager", "controller.py"]
```

Figure 3: Dockerfile

1.2 Punti deboli

Le attuali debolezze del controller Ryu sono le seguenti:

- **Over blocking:** la strategia di mitigazione blocca tutto il traffico su una porta dello switch una volta superata una soglia di throughput. Inoltre, potrebbe bloccare più switch del necessario.
- **Controller-Centric Blocking Decisions:** solo il controller prende le decisioni di blocco, basandosi esclusivamente sulla propria logica di monitoraggio.
- **Lack of Modular Detection and Mitigation Design:** monitoraggio, processo decisionale e applicazione delle regole non sono chiaramente separati.
- **Static threshold:** la soglia di throughput è statica, mentre sarebbe preferibile un approccio dinamico, adattato all'andamento del traffico.
- **Complex topology:** il codice deve essere reso scalabile e adattabile, in modo da non dover essere modificato ogni volta che cambia la topologia.

2 Over blocking

In questo progetto si è deciso di partire dal controller di default di Ryu, *SimpleSwitch13.py*, per una più semplice implementazione delle nuove funzionalità. La strategia di difesa dagli attacchi DoS è stata suddivisa in cinque punti fondamentali:

- **Monitoraggio**
- **Rilevamento delle anomalie**
- **Calcolo della penalty**
- **Blocco**
- **Recovery**

2.1 Monitoraggio

La funzione *monitor()* raccoglie le statistiche ogni 2 secondi e le salva all'interno della struttura *shared_data*.

```python
class TrafficMonitor:

    def __init__(self, sleep_time=2):
        self.sleep_time = sleep_time
        hub.spawn(self.monitor)

    def monitor(self):
        while True:
            for dp in shared_data.datapaths.values():
                req = dp.ofproto_parser.OFPPortStatsRequest(dp, 0, dp.ofproto.OFPP_ANY)
                dp.send_msg(req)
            hub.sleep(self.sleep_time)
```

Figure 4: Classe Monitor

2.2 Rilevamento delle anomalie

Per quanto riguarda il rilevamento delle anomalie, è stata implementata la funzione *evaluate()*, la quale, dopo aver raccolto un campione adeguato di dati, ne calcola la media e la varianza. Tali valori vengono poi confrontati con la nostra soglia dinamica (*dynamic threshold*).

```python
class PolicyEngine:
    def __init__(self, history_length=10, var_threshold=2e13):
        self.history_length = history_length
        self.var_threshold = var_threshold
        self.traffic_history = defaultdict(lambda: deque(maxlen=history_length))

    def update(self, dpid, port, rx_bps):
        self.traffic_history[(dpid, port)].append(rx_bps)
        return self.traffic_history[(dpid, port)]

    def evaluate(self, dpid, port, rx_bps):
        history = self.traffic_history[(dpid, port)]
        if len(history) < 5:
            return False, 0, 0

        avg = statistics.mean(history)
        var = statistics.variance(history)
        threshold_dyn = max(avg * 1.5, 10e6)

        ABSOLUTE_HIGH_THRESHOLD = 30e6  # 30 Mbps

        # Suspicious se:
        # 1. Spike + alta varianza (attacco bursty) O
        # 2. Traffico costantemente sopra soglia assoluta (attacco sustained)
        spike_and_unstable = rx_bps > threshold_dyn * 1.2 and var > self.var_threshold
        sustained_high = rx_bps > ABSOLUTE_HIGH_THRESHOLD and avg > ABSOLUTE_HIGH_THRESHOLD

        suspicious = spike_and_unstable or sustained_high

        return suspicious, var, threshold_dyn
```

Figure 5: Classe PolicyEngine

Le condizioni necessarie affinché venga rilevato traffico anomalo sono due: la presenza di **spike di traffico** e un'elevata **variabilità**. In particolare, il traffico attuale deve superare del 20% la **soglia dinamica** e mostrare picchi irregolari, poiché il traffico legittimo tende ad essere più stabile.

Nonostante l'adozione della soglia dinamica, è comunque presente un parametro *ABSOLUTE_HIGH_THRESHOLD*, che garantisce una protezione ulteriore nel caso in cui un host tenti di saturare la rete.

2.3 Calcolo della penalty

Il controller non si limita a un blocco statico: in presenza di attacchi ripetuti, reagisce con periodi di blocco esponenzialmente più lunghi.

```python
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
 def port_stats_reply_handler(self, ev):
     dp = ev.msg.datapath
     dpid = dp.id

     for stat in ev.msg.body:
         port = stat.port_no
         if port == 4294967294:
             continue

         key_port = (dpid, port)

         if key_port in self.prev_bytes:
             rx_bytes_diff = stat.rx_bytes - self.prev_bytes[key_port]['rx']
             tx_bytes_diff = stat.tx_bytes - self.prev_bytes[key_port]['tx']

             rx_bps = (rx_bytes_diff * 8) / self.SLEEP_TIME
             tx_bps = (tx_bytes_diff * 8) / self.SLEEP_TIME
         else:
             rx_bps = 0
             tx_bps = 0

         self.prev_bytes[key_port] = {
             'rx': stat.rx_bytes,
             'tx': stat.tx_bytes
         }

         total_bps = rx_bps + tx_bps

         print(f"[S{dpid}:P{port}] ↓ RX{rx_bps/1e6:.2f} Mbps - ↑ TX {tx_bps/1e6:.2f} Mbps | Tot: {total_bps/1e6:.2f} Mbps")

         self.policy_engine.update(dpid, port, rx_bps)
         suspicious, var, threshold_dyn = self.policy_engine.evaluate(dpid, port, rx_bps)

         if rx_bps > 1e6 or suspicious:
             print(f"  └─ Threshold: {threshold_dyn/1e6:.2f} Mbps | Var: {var:.2e}")

         key = f"{dpid}-{port}"

         if suspicious and key in shared_data.host_info and not shared_data.is_blocked(key):
             attacker = shared_data.host_info[key]
             shared_data.block_counts[key] = shared_data.block_counts.get(key, 0) + 1
             num_blocks = shared_data.block_counts[key]
             unblock_delay = min(2 ** num_blocks * self.BASE_DELAY, self.MAX_DELAY)
             self.flow_enforcer.block(dp, key, attacker['ip'], unblock_delay)
```

Figure 6: Controller

Grazie al meccanismo di penalty esponenziale, un falso positivo verrà sbloccato rapidamente al primo rilevamento (ad esempio dopo 5 secondi), mentre un attaccante persistente subirà tempi di blocco progressivamente più lunghi.

2.4 Blocco

Per coordinare il processo di blocco si utilizza la funzione ausiliaria *block()*, presente nella classe *FlowEnforcer*. Questa funzione aggiunge l'host malevolo alla blacklist e installa le nuove regole di drop.

```
def block(self, dp, key, attacker_ip, unblock_delay):
        shared_data.blocked.add(key)

        self.controller.logger.warning(f"{RED}[BLOCK] Connection UDP from {attacker_ip}
 to 10.0.0.3 | restoring the connection after {unblock_delay}s {RESET}")
        self.controller.block_udp_flow(dp, attacker_ip, "10.0.0.3")
        hub.spawn(self.unblock, dp, key, attacker_ip, unblock_delay)
```

Figure 7: Funzione *block()* della classe FlowEnforcer

Per l'implementazione delle regole di *drop* di OpenFlow, necessarie a bloccare i pacchetti dell'attaccante, viene invece chiamata la funzione *block_udp_flow()* all'interno del file *controller.py*.

```
def block_udp_flow(self, dp, src_ip, dst_ip):
    parser = dp.ofproto_parser
    ofproto = dp.ofproto
    match = parser.OFPMatch(eth_type=0x0800, ip_proto=17, ipv4_src=src_ip, ipv4_dst=dst_ip)
    actions = []
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
    mod = parser.OFPFlowMod(
        datapath=dp,
        priority=100,
        match=match,
        instructions=inst,
        command=ofproto.OFPFC_ADD,
        idle_timeout=0,
        hard_timeout=0
    )
    dp.send_msg(mod)
    self.logger.info(f"[BLOCKLIST] Flow rule applied on S{dp.id}: {src_ip} → {dst_ip}")
```

Figure 8: Aggiunta di una regola tramite la funzione *block_udp_flow()*

2.5 Recovery

Infine, sarà la stessa classe *FlowEnforcer* a occuparsi della fase di sblocco: al termine del tempo di blocco, l'host precedentemente identificato come malevolo verrà rimosso dalla blacklist.

```python
def unblock(self, dp, key, src_ip, delay):
    hub.sleep(delay)
    parser = dp.ofproto_parser
    ofproto = dp.ofproto
    match = parser.OFPMatch(eth_type=0x800, ip_proto=17, ipv4_src=src_ip)
    actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
    mod = parser.OFPFlowMod(datapath=dp, priority=20, match=match, instructions=inst, command=ofproto.OFPFC_MODIFY)
    dp.send_msg(mod)

    self.controller.logger.info(f"{GREEN}[UNBLOCK] Connection UDP from {src_ip} restored{RESET}")

    shared_data.blocked.discard(key)
```

Figure 9: Funzione *unblock()* della classe FlowEnforcer

3 Debolezze architetturali

3.1 Controller-Centric Blocking Decisions

Una delle principali limitazioni dell'architettura originale risiedeva nella centralizzazione delle decisioni di blocco esclusivamente all'interno del controller, impedendo così l'integrazione di policy esterne o di interventi amministrativi. Per risolvere questa criticità è stato implementato un framework estensibile basato su strutture dati condivise.

La soluzione proposta si articola in tre componenti fondamentali:

- **Shared Data Structure**
- **Multi-Source Blocking**
- **External Policy Integration**

3.1.1 Shared Data Structure

È stata implementata la classe *SharedData*, che centralizza tutte le informazioni relative ai blocchi, sostituendo le variabili locali del controller con una struttura globalmente accessibile.

```
class SharedData:
    def __init__(self):
        self.blocked = set()
        self.external_blocks = set()
        self.block_counts = {}

        self.datapaths = {}  # Switch connessi
        self.host_info = {}  # Info host dal JSON

        self.on_suspicious_detected = None  # PolicyEngine -> FlowEnforcer
        self.on_block_applied = None       # FlowEnforcer -> Logger
```

Figure 10: Struttura dati condivisa per la gestione della blocklist

La distinzione tra *blocked* (blocchi interni) ed *external_blocks* (blocchi esterni) consente una gestione granulare delle diverse fonti di policy.

3.1.2 Multi-Source Blocking

Il sistema ora supporta blocchi provenienti da fonti multiple tramite il metodo unificato *is_blocked()*, che verifica simultaneamente entrambe le blacklist.

```
def is_blocked(self, key):
        return key in self.blocked or key in self.external_blocks
```

Figure 11: Verifica unificata dello stato di blocco

3.1.3 External Policy Integration

L'API dedicata ai blocchi esterni consente a moduli terzi o ad amministratori di contribuire dinamicamente alle policy di sicurezza, senza la necessità di modificare il core del controller.

```
def add_external_block(self, key, reason="external"):
    self.external_blocks.add(key)
    print(f"[EXTERNAL BLOCK] {key} - {reason}")
def remove_external_block(self, key):
    self.external_blocks.discard(key)
    print(f"[EXTERNAL UNBLOCK] {key}")
```

Figure 12: API per l'integrazione di policy esterne

Questo approccio garantisce estensibilità e coerenza nelle decisioni di blocco attraverso un'interfaccia unificata.

3.2 Modular Architecture Design

L'architettura monolitica originale accoppiava strettamente monitoraggio, rilevamento e mitigazione all'interno di un'unica classe, riducendo la manutenibilità e l'estensibilità del sistema. La nuova soluzione separa invece tali responsabilità in moduli indipendenti, che comunicano tra loro mediante strutture dati condivise.

La nuova architettura si basa su tre moduli specializzati:

- **Traffic Monitor**

- **Policy Engine**
- **Flow Enforcer**

3.2.1 Traffic Monitor

Il modulo di monitoraggio opera in un thread dedicato, occupandosi esclusivamente della raccolta delle statistiche di rete, senza alcuna logica decisionale (Figura 4). L'utilizzo di *shared_data.datapaths* elimina la dipendenza diretta dal controller, permettendo un'esecuzione completamente autonoma.

3.2.2 Policy Engine

Il motore delle policy (*PolicyEngine*) è responsabile dell'analisi statistica e della classificazione del traffico come sospetto o legittimo (Figura 5). La separazione logica consente di modificare o sostituire gli algoritmi di rilevamento senza impattare i moduli di monitoraggio o di enforcement.

3.2.3 Flow Enforcer

Il modulo di enforcement (*FlowEnforcer*) gestisce esclusivamente l'applicazione e la rimozione delle regole di blocco, interfacciandosi con OpenFlow tramite il controller (Figura 7).

3.2.4 Decoupled Communication

I moduli comunicano esclusivamente attraverso *shared_data* (Figure 10, 11, 12), eliminando dipendenze circolari e accoppiamenti stretti. Questo design pattern facilita il testing unitario, la sostituzione dei componenti e la scalabilità orizzontale.

In conclusione, l'architettura risultante offre una maggiore flessibilità operativa, mantenendo al contempo performance ottimali.

4 Simulazione

Le simulazioni sono state eseguite su due diverse topologie: una più semplice, riportata in figura 1, e una più complessa, mostrata in figura 23. Questo approccio ha consentito di verificare sia la correttezza del funzionamento di base, sia la scalabilità e la flessibilità del *controller* implementato.

4.1 Topologia semplice

Per una prima validazione del sistema è stata utilizzata una topologia ridotta. Per verificare la connettività tra i nodi è stato eseguito il comando `pingAll` di Mininet, che testa la raggiungibilità reciproca. I risultati, mostrati in figura 13, confermano la piena connettività della rete.

```
*** Avvio dell'ambiente
*** Starting controller
*** Adding hosts
*** Adding switches
*** Adding links
(10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms d
elay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay) (15.00Mbi
t 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay
) *** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
C1
*** Starting 4 switches
s1 (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay) s2 (10.00Mbit 2ms delay) (15.00Mbi
t 25ms delay) s3 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) s4 (10.00Mbit 2ms
delay) (15.00Mbit 25ms delay) ...(10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay) (10.
00Mbit 2ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms d
elay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay)
*** 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)
*** Avvio CLI
*** Starting CLI:
mininet>
```

Figure 13: Connettività della rete tramite `pingAll`

Attraverso il comando `ovs-ofctl dump-flows s1` è stato inoltre possibile verificare lo stato delle regole installate sullo switch `s1`, che in condizioni normali risultano assenti (figura 14).

```
friccio@friccio-virtual-machine: $ sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=15.565s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth3",dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=15.560s, table=0, n_packets=2, n_bytes=140, 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-eth3"
 cookie=0x0, duration=15.326s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth3",dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=15.320s, table=0, n_packets=2, n_bytes=140, 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-eth3"
 cookie=0x0, duration=15.195s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=15.189s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth1",dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04 actions=output:"s1-eth2"
 cookie=0x0, duration=14.759s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth3"
 cookie=0x0, duration=14.639s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth3",dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=output:"s1-eth2"
 cookie=0x0, duration=14.282s, table=0, n_packets=3, n_bytes=238, priority=1,in_port="s1-eth2",dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:03 actions=output:"s1-eth3"
 cookie=0x0, duration=14.168s, table=0, n_packets=2, n_bytes=140, priority=1,in_port="s1-eth3",dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:04 actions=output:"s1-eth2"
 cookie=0x0, duration=16.611s, table=0, n_packets=86, n_bytes=9515, priority=0 actions=CONTROLLER:65535
```

Figure 14: Dump flows in condizioni iniziali (assenza di regole)

Successivamente, mettendo gli host in ascolto sulle porte UDP e TCP, è stato possibile simulare il traffico applicativo. Nel nostro scenario il nodo `h1` agisce come attaccante, cercando di interrompere le comunicazioni verso il nodo `h3`.

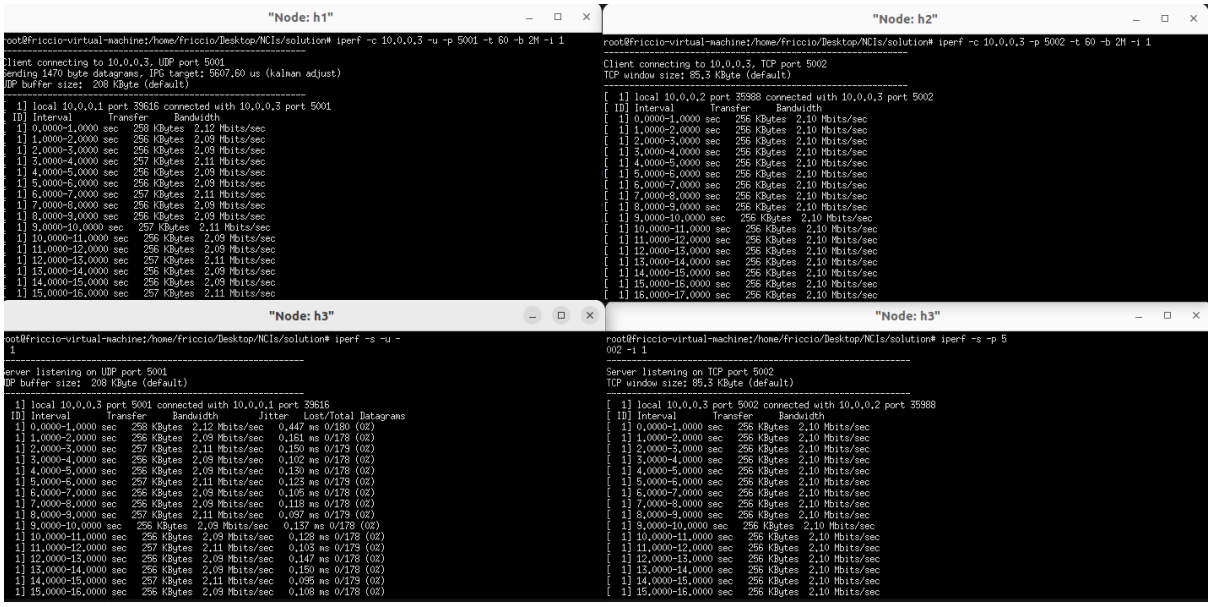

Figure 15: Generazione del traffico regolare da parte degli host

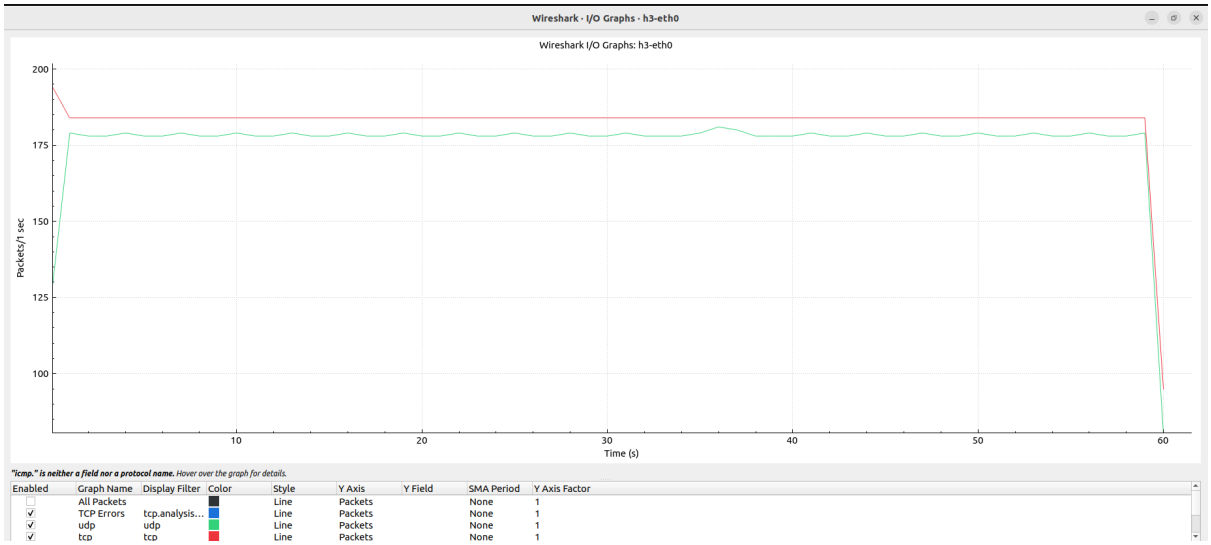

Figure 16: Andamento del traffico regolare

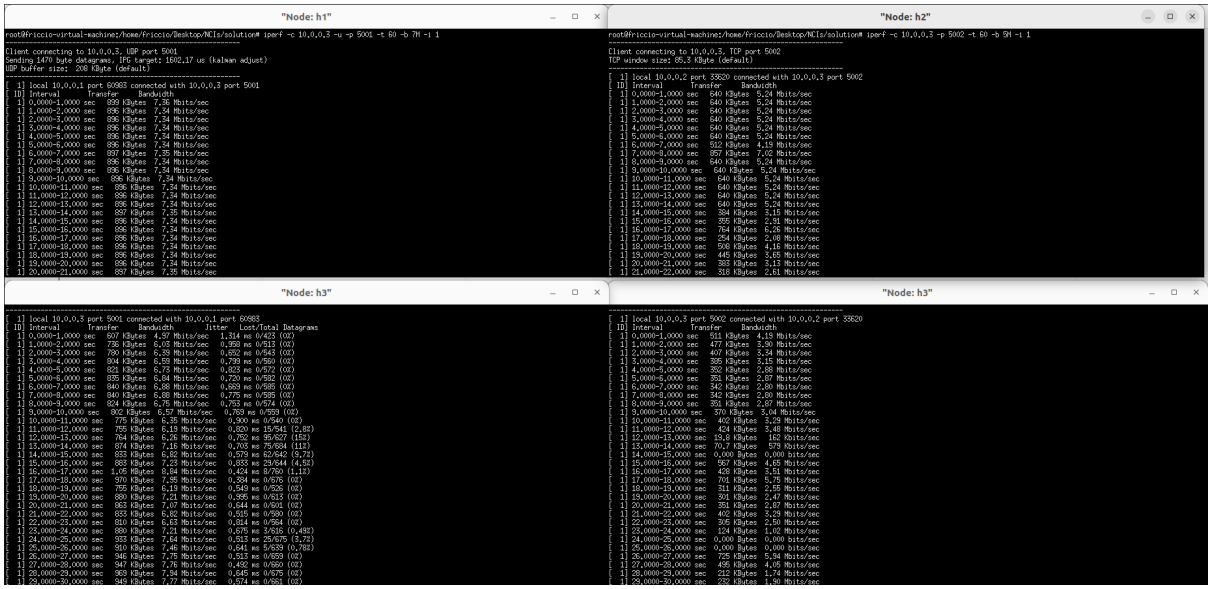

Figure 17: Generazione del traffico irregolare da parte degli host

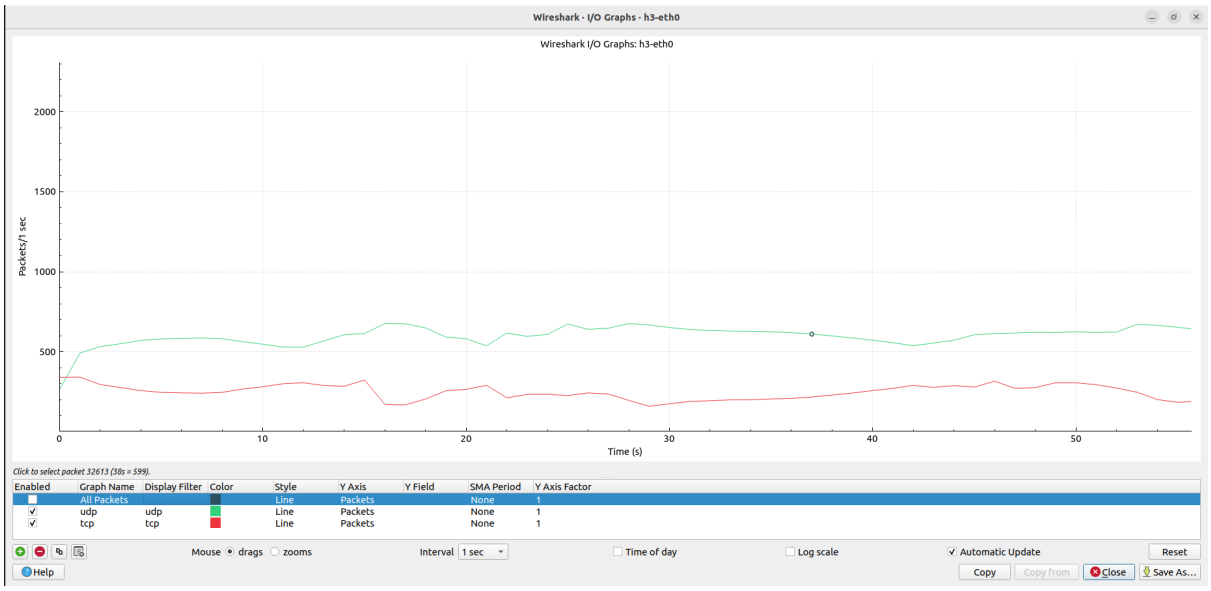

Figure 18: Andamento del traffico irregolare

In assenza del monitoraggio da parte del controller, è possibile osservare come, in presenza di traffico irregolare (figure 17 e 18), non vi sia alcun meccanismo di protezione della rete. In tali condizioni, qualsiasi host può facilmente saturare la larghezza di banda disponibile, causando perdita di pacchetti (*packet loss*) e, nei casi più gravi, l'interruzione completa delle comunicazioni.

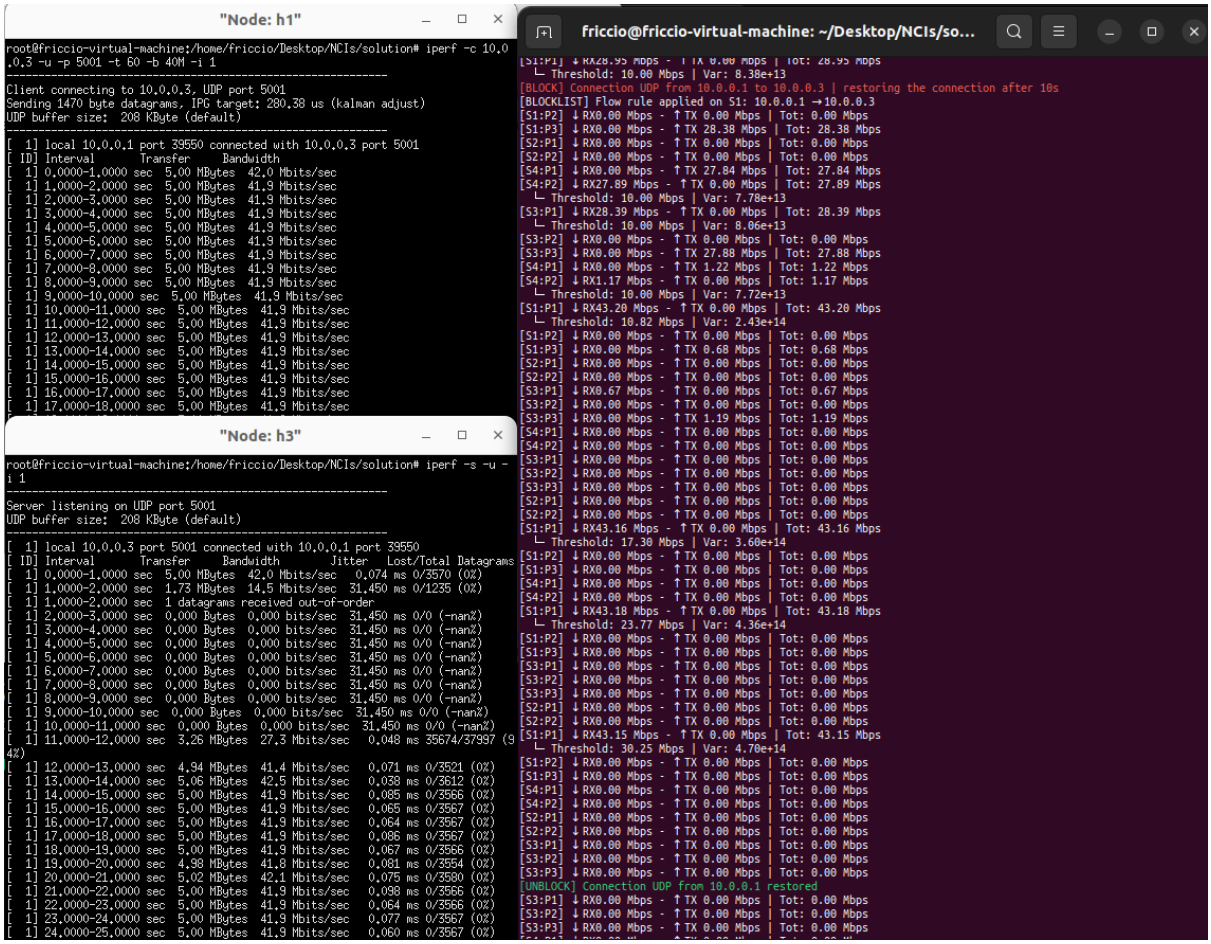

Figure 19: Blocco del traffico UDP

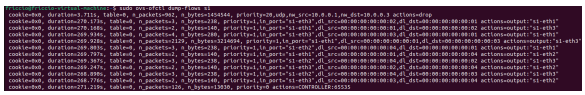

Figure 20: Dump flows post-bloccaggio

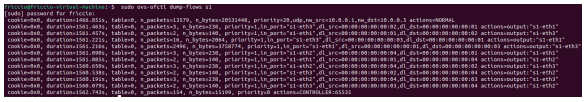

Figure 21: Dump flows post-rilascio

Con la presenza del controller, invece, la rete risulta protetta anche in caso di eventi sospetti o potenzialmente malevoli.

Dall'analisi dei log del controller e dei *dump flows* risulta evidente che, al momento del rilevamento di uno spike di traffico, viene installata sullo switch una regola di *drop* (figure 19 e 20). Successivamente, al termine del periodo di blocco, la regola viene sostituita con una regola *NORMAL* (figura 21).

In caso di comportamento malevolo persistente, il blocco può essere riapplicato in maniera incrementale, con una durata superiore rispetto all'occorrenza precedente. Un esempio di questo comportamento è mostrato nel caso della topologia allargata.

Figure 22: Traffico UDP analizzato con Wireshark

L'analisi tramite strumenti esterni, come Wireshark, conferma tali risultati. In figura 22 è riportato l'andamento delle comunicazioni: il traffico UDP viene interrotto nel momento in cui viene applicata la regola di *drop*, per poi riprendere non appena la regola viene sostituita con *NORMAL*.

4.2 Topologia allargata

Per valutare la scalabilità del controller è stata utilizzata una topologia più estesa, riportata in figura 23.

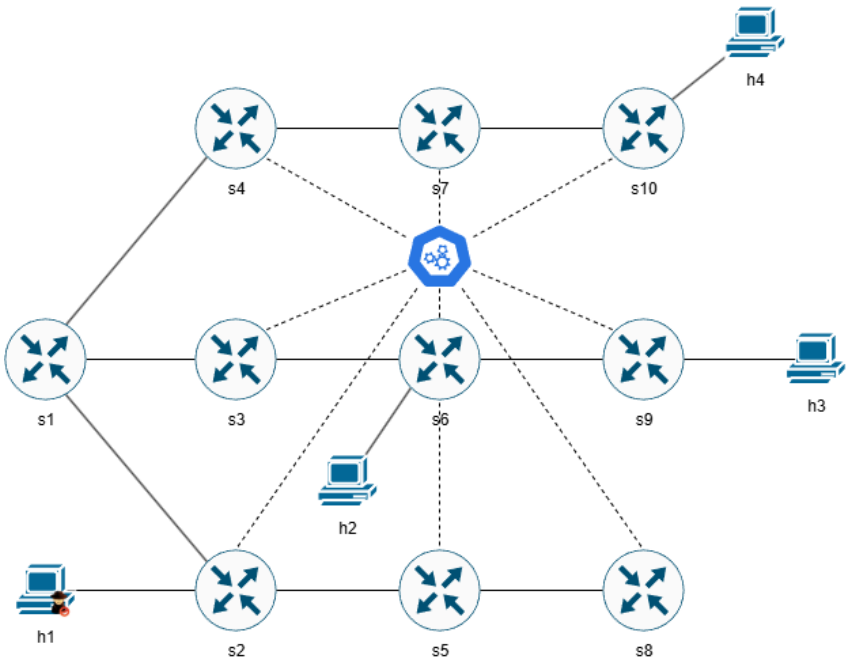

Figure 23: Topologia complessa utilizzata per i test di scalabilità

Come nel caso precedente, il comando `pingAll` ha confermato la raggiungibilità dei nodi (figura 24).

```
*** Avvio dell'ambiente complesso
*** Starting controller
*** Adding hosts
*** Adding switches
*** Adding links
(10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms d
elay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay) (15.00Mbi
t 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay
) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit
 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay)
 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) *** Starting network
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
C1
*** Starting 10 switches
s1 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) s2 (10.00Mbit 2ms delay) (15.00M
bit 25ms delay) (15.00Mbit 25ms delay) s3 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) s4 (15.00Mbit 25
ms delay) (15.00Mbit 25ms delay) s5 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) s6 (10.00Mbit 2ms dela
y) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) s7 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) s8 (15
.00Mbit 25ms delay) s9 (10.00Mbit 2ms delay) (15.00Mbit 25ms delay) s10 (10.00Mbit 2ms delay) (15.00Mbit
 25ms delay) ...(15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (10.00Mbit 2ms dela
y) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbi
t 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (10.00Mbit 2ms delay)
 (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit 25ms delay) (15.00Mbit
25ms delay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay) (10.00Mbit 2ms delay) (15.00Mbit 25ms delay)
*** 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)
*** Network topology summary:
*** Structure: Perfect Binary Tree (4 levels, NO CYCLES)
*** Level 0: s1 (root)
*** Level 1: s2, s3, s4
*** Level 2: s5, s6, s7
*** Level 3: s8, s9, s10
*** Legitimate hosts: h1(s2), h2(s6), h3(s9), h4(s10)
*** Avvio CLI
*** Starting CLI:
```

Figure 24: Connettività della topologia complessa tramite `pingAll`

In figura 25 sono riportati i terminali della topologia durante la fase di simulazione. Il caso di studio prevede tre host mittenti con traffico pari a 2 Mbps, 30 Mbps e 80 Mbps. In teoria, il primo flusso deve essere accettato senza problemi, il secondo deve consentire l'adattamento dinamico della soglia, mentre il terzo deve essere completamente bloccato, poiché saturerebbe la larghezza di banda dei link causando *packet loss* e degrado delle comunicazioni.

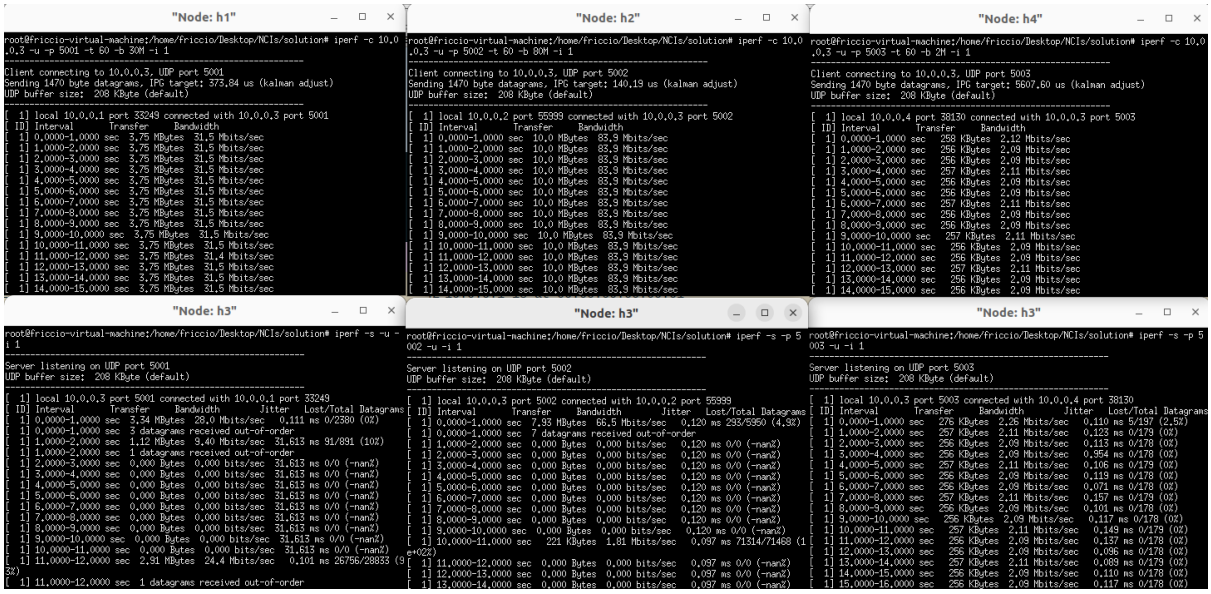

Figure 25: Terminali attivi nella topologia complessa

Analogamente al caso della topologia semplice, anche in questo scenario il controller è stato in grado di bloccare i flussi sospetti attraverso l'installazione di regole di *drop* sugli switch (figura 26).

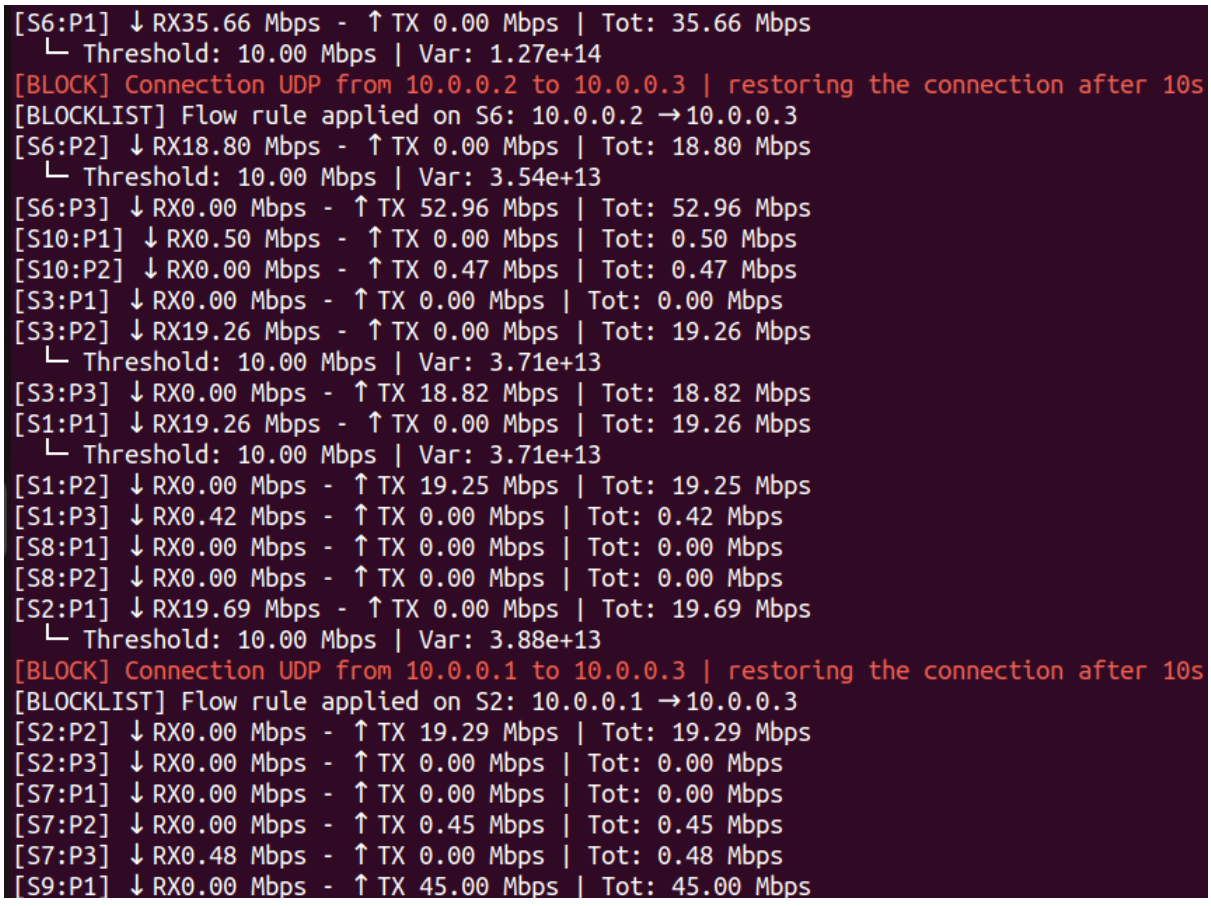

Figure 26: Primo blocco applicato nella topologia complessa

Come previsto, al primo segnale anomalo vengono bloccati sia l'host `h1` sia l'host `h4`. Successivamente, durante la fase di *recovery*, l'host `h1` viene sbloccato, mentre l'host `h4` viene ribloccato con una durata di penalità maggiore.

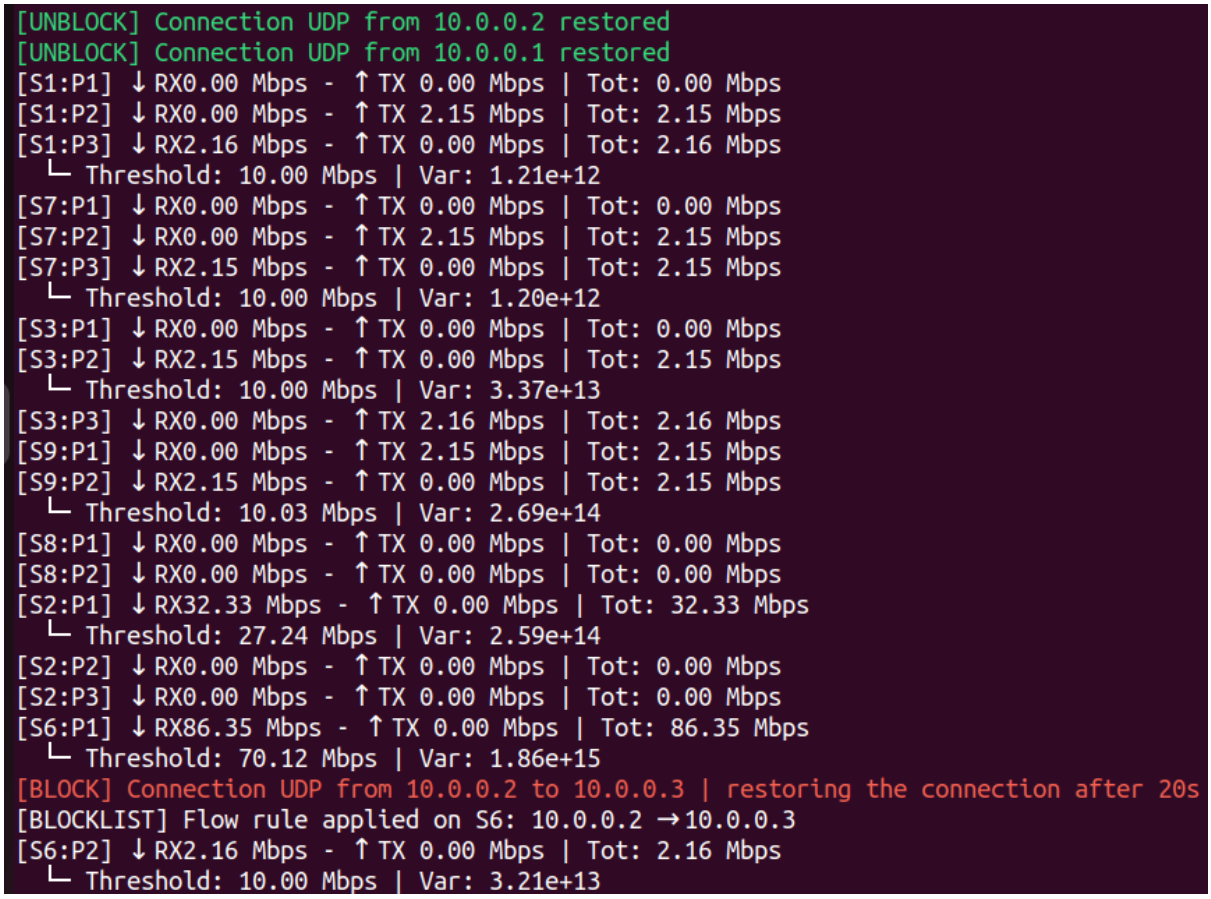

Figure 27: Secondo blocco e recovery nella topologia complessa

Infine, dall'analisi del traffico tramite Wireshark (figura 28) si può osservare come il comportamento rilevato sia coerente con quello ottenuto nella topologia semplice: i pacchetti UDP al di sotto della soglia viaggiano senza restrizioni (linea rossa), quelli che superano la soglia ma non l'*absolute threshold* subiscono un primo blocco temporaneo per consentire la stabilizzazione della soglia, mentre i pacchetti che oltrepassano la soglia assoluta vengono ribloccati a ogni nuovo tentativo.

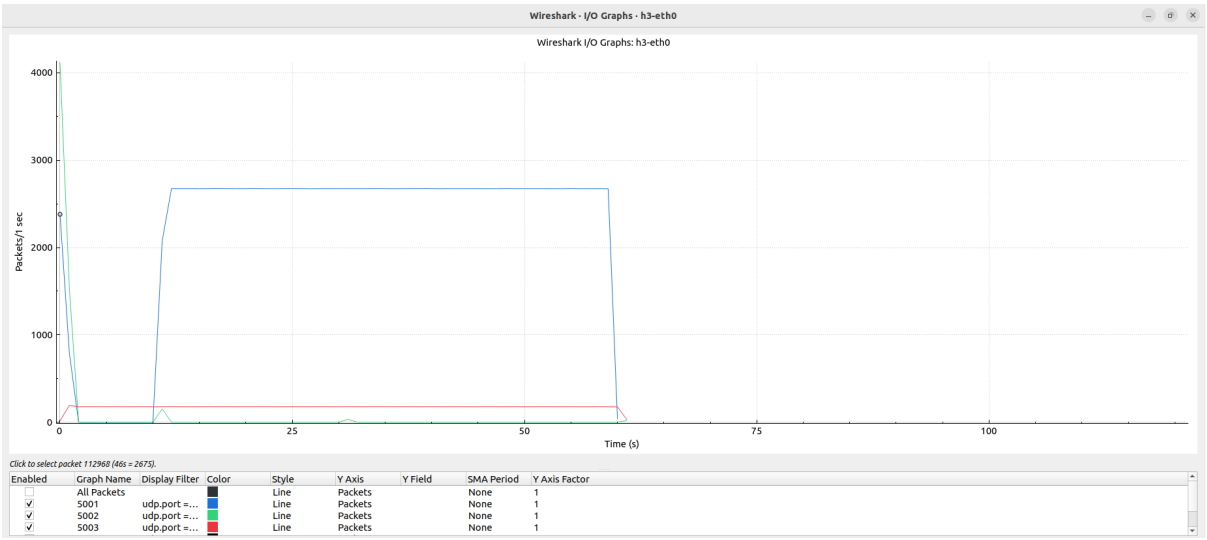

Figure 28: Analisi del traffico con Wireshark nella topologia complessa