UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato di Network and Cloud Infrastructure

Anno Accademico 2024-25

Prof. Giorgio Ventre

Prof. Roberto Canonico

Prof. Alessio Botta

Prof. Giovanni Stanco

Studente
Fabio Accurso M63001723

Contents

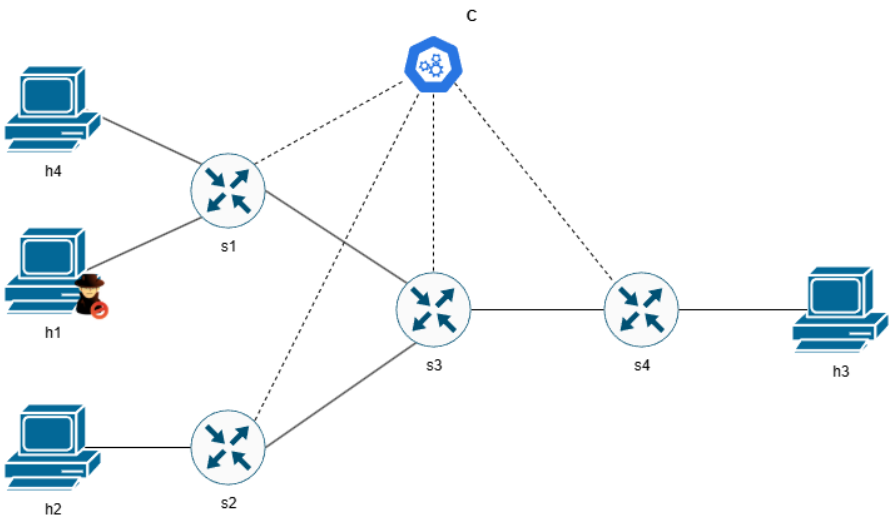
Chapter 1

Introduzione

Il progetto ha come obiettivo quello di risolvere alcune lacune di una rete basata su SDN, fornita dai docenti. La rete è stata virtualizzata tramite **mininet**, con il controller basato sul framework **ryu**, che gestisce gli switch tramite protocollo **OpenFlow 1.3** Vediamo quindi nel dettaglio il punto di partenza nei seguenti paragrafi.

1.1 Topologia

La topologia di partenza presenta 4 hosts (*h1,h2,h3,h4*) ciascuno con un indirizzo IP univoco all'interno della stessa sottorete, e 4 switch (*s1,s2,s3,s4*).



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, garantendo la comunicazione tra tutti gli host. I link tra gli host e i rispettivi switch d'accesso sono stati configurati con una larghezza di banda costante di 100 Mbps (ad eccezione del collo di bottiglia tra *h3* ed *s4* con larghezza di banda pari a 40) e un ritardo di 2 ms, allo stesso modo i collegamenti tra gli switch presentano una larghezza di banda costante, pari a 100 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 = 100, delay = '2ms')
self.net.addLink(self.h4, self.s1, bw = 100, delay = '2ms')
self.net.addLink(self.h2, self.s2, bw = 100, delay = '2ms')
self.net.addLink(self.h3, self.s4, bw = 40, delay = '2ms')  #<-- Collo di bottiglia

# Collegamenti tra switch
self.net.addLink(self.s1, self.s3, bw = 100, delay = '25ms')
self.net.addLink(self.s2, self.s3, bw = 100, delay = '25ms')
self.net.addLink(self.s4, self.s3, bw = 100, delay = '25ms')
```

1.2 Debolezze individuate

Le attuali debolezze individuate 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. In questo modo potrebbe bloccare più switch/hosts 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.

- **Topology sensitivity**: il codice deve essere reso scalabile e adattabile, in modo da non dover essere modificato ogni volta che cambia la topologia.

Chapter 2

Meccanismo adattivo di mitigazione del traffico

In questa sezione viene descritto il meccanismo di mitigazione adattiva implementato nel controller SDN. Il controller di partenza scelto è *SimpleSwitch13.py*, controller di default di Ryu. L'obiettivo non è applicare un blocco indiscriminato del traffico, ma adottare una strategia reattiva e proporzionale, capace di distinguere variazioni fisiologiche del traffico da comportamenti potenzialmente malevoli. Il sistema si basa su cinque componenti logiche:

1. Monitoraggio
2. Rilevamento delle anomalie
3. Calcolo della penalty

4. Applicazione del Blocco Selettivo

5. Recovery Adattivo

Questa suddivisione consente di separare raccolta dati, analisi e applicazione delle policy, migliorando la chiarezza architetturale e la manutenibilità del controller.

2.1 Monitoraggio

Il primo stadio del meccanismo consiste nella raccolta periodica delle statistiche di traffico dagli switch OpenFlow. Il controller invia richieste di tipo **OFPPortStatsRequest** ai datapath registrati, acquisendo informazioni relative ai byte trasmessi e ricevuti per ciascuna porta. Il campionamento avviene con intervallo temporale fisso (2 secondi), valore scelto come compromesso tra reattività e stabilità. I dati raccolti vengono memorizzati in una struttura condivisa, mantenendo uno storico temporale necessario per le successive elaborazioni statistiche. È importante sottolineare che questo livello non prende decisioni: esso svolge esclusivamente una funzione di osservazione continua dello stato della rete.

```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 2.1: Thread di osservazione periodica e richiesta delle statistiche di porta tramite OpenFlow

2.2 Rilevamento delle anomalie

Una volta raccolti i campioni di traffico, il sistema procede alla modellazione statistica del comportamento osservato. Per ciascun flusso monitorato viene mantenuta una finestra temporale contenente gli ultimi valori di throughput. A partire da tali campioni vengono calcolati:

- media aritmetica
- varianza

La media rappresenta il livello medio di traffico generato nel periodo di osservazione, mentre la varianza misura il grado di instabilità del comportamento. Sulla base di tali parametri viene definita una soglia dinamica (*dynamic threshold*), che si adatta automaticamente alle condizioni correnti della rete. Un'anomalia viene rilevata quando si verificano simultaneamente due condizioni:

- Il traffico corrente supera la soglia dinamica con uno scostamento

significativo.

- La variabilità del traffico indica la presenza di picchi irregolari.

In aggiunta alla soglia adattiva, è definito un limite massimo assoluto (*absolute threshold*), che garantisce protezione in caso di tentativi di saturazione della banda indipendentemente dal contesto statistico.

```python
class PolicyEngine:
    """
    IDENTICO all'originale, ma usa shared_data per comunicare
    """
    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):
        """IDENTICO all'originale"""
        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, 25e6)

        # Soglia assoluta per traffico molto alto
        ABSOLUTE_HIGH_THRESHOLD = 90e6  # 90 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 2.2: Calcolo della soglia dinamica e verifica delle condizioni di anomalia

2.3 Calcolo della penalty

L'identificazione di un'anomalia non comporta automaticamente un blocco statico e permanente. È stato implementato un meccanismo di escalation progressiva della penalità. Ogni nodo mantiene uno storico delle violazioni precedenti. La durata del blocco viene calcolata in funzione del numero di eventi anomali rilevati nel tempo. Questo approccio consente di distinguere tra:

- falsi positivi temporanei
- comportamenti aggressivi persistenti

Un singolo evento produce un blocco di breve durata; al contrario, un nodo che continua a generare traffico anomalo subisce penalità progressivamente più lunghe. Dal punto di vista sistemico, tale meccanismo introduce una memoria storica nel processo decisionale, rendendo la mitigazione proporzionale alla persistenza del comportamento.

```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 2.3: Meccanismo di incremento progressivo della durata del blocco

2.4 Applicazione del Blocco Selettivo

Quando viene classificato un comportamento come anomalo, il controller procede con l'applicazione della misura di mitigazione. L'azione consiste nell'installazione dinamica di una regola OpenFlow di tipo drop nella flow table dello switch interessato. L'operazione avviene tramite messaggi **FlowMod**, che modificano il comportamento dello switch senza interrompere il funzionamento globale della rete. Il nodo responsabile viene inoltre inserito in una blacklist interna, garantendo

coerenza tra stato logico e configurazione effettiva della rete. Questo intervento è mirato e non influisce sugli altri host legittimi.

```
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 2.4: Funzione block() della classe FlowEnforcer

```
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 2.5: Aggiunta di una regola tramite la funzione block udp flow()

2.5 Recovery Adattivo

Per evitare blocchi permanenti causati da condizioni temporanee, il sistema prevede una fase di ripristino automatico. Al termine dell'intervallo di penalità:

- la regola di drop viene rimossa
- viene ripristinata la regola di inoltro normale
- il nodo viene rimosso dalla blacklist

Questa fase garantisce equità tra i nodi e riduce l'impatto di eventuali falsi positivi. Nel caso in cui il comportamento anomalo si ripresenti, il meccanismo di escalation assicura un intervento più severo rispetto alla violazione precedente.

```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 2.6: Funzione unblock() della classe FlowEnforcer

Chapter 3

Miglioramenti Progettuali

In questo capitolo vengono descritte le modifiche apportate al progetto di partenza, con l'obiettivo di superare le principali limitazioni evidenziate, rendendo il controller modulare, decentralizzato e in grado di operare su topologie di rete complesse. Il sistema originale presentava diversi problemi strutturali:

1. Overblocking del traffico
2. Soglie statiche non adattive
3. Decisioni centralizzate esclusivamente nel controller
4. Mancanza di modularità
5. Sensibilità alla topologia

3.1 Overblocking

Il problema dell'overblocking è stato superato, come visto nel capitolo precedente, abbandonando l'approccio drastico che prevedeva il blocco totale del traffico su una specifica porta dello switch una volta superata la soglia di guardia. Come illustrato nel modulo di enforcement, la mitigazione avviene ora con una granularità a livello di flusso. L'installazione dinamica delle regole OpenFlow di drop intercetta in modo mirato il traffico anomalo, basandosi su parametri specifici come l'indirizzo IP sorgente. Questo intervento chirurgico garantisce che le comunicazioni legittime degli altri host attestati sul medesimo switch non subiscano alcuna interruzione.

3.2 Static Threshold

Come evidenziato nel capitolo precedente la rigidità introdotta dall'utilizzo di una Static Threshold è stata sostituita da un modello di rilevamento statistico. Il sistema non valuta più il throughput contro un valore limite preimpostato e immutabile , bensì calcola continuamente una soglia adattiva (*dynamic threshold*). Tale limite viene aggiornato in tempo reale in funzione della media aritmetica e della varianza calcolate su una finestra temporale di campioni recenti. Questo approccio permette al controller di adattarsi fisiologicamente alle normali fluttuazioni del traffico di rete, distinguendo in modo affidabile i picchi transitori

legittimi da tentativi di attacco sostenuti o di tipo bursty.

3.3 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

3.3.1 Shared Data Structure

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

```python
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 3.1: 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.

3.3.2 Multi-Source Blocking

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

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

Figure 3.2: Verifica unificata dello stato di blocco

3.3.3 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.

```python
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 3.3: API per l'integrazione di policy esterne

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

3.4 Modular Architecture Design

L'architettura di partenza 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

3.4.1 Traffic Monitor

Questo componente lavora in background all'interno di un thread indipendente. Il suo unico scopo è reperire periodicamente le metriche di traffico dagli switch, astenendosi da qualsiasi valutazione in merito ad eventuali anomalie. Avvalendosi della struttura *shared_ data.datapaths*, il processo di monitoraggio risulta completamente svincolato dal controller centrale, garantendo un'operatività asincrona e continua.

3.4.2 Policy Engine

Il motore delle policy costituisce il nucleo analitico del sistema. Si occupa di elaborare i dati grezzi raccolti dal monitor per distinguere i flussi anomali da quelli fisiologici tramite l'analisi statistica. Il principale vantaggio di questo isolamento logico è la manutenibilità: è infatti possibile aggiornare o perfezionare le euristiche di rilevamento degli attacchi senza rischiare di alterare le funzioni di raccolta dati o le procedure di mitigazione.

3.4.3 Flow Enforcer

Questo modulo rappresenta il braccio esecutivo dell'architettura. Ha il compito specifico e limitato di applicare o revocare le misure di mitigazione (come l'inserimento di regole di drop) nelle tabelle di flusso degli switch. Per tradurre le decisioni in azioni concrete, si interfaccia direttamente con il protocollo OpenFlow delegando la comunicazione

di basso livello al controller.

3.4.4 Decoupled Communication

Il perno che unisce questa architettura distribuita è il modulo shared_data, che funge da unico punto di scambio informativo tra le varie entità. Sostituendo le chiamate dirette tra componenti con uno stato condiviso, si scongiura la formazione di dipendenze circolari. Questo paradigma di progettazione non solo agevola lo sviluppo e il testing isolato di ciascun componente, ma assicura anche un'elevata scalabilità orizzontale. Il risultato finale è un ecosistema altamente flessibile, capace di adattarsi a nuove esigenze operative mantenendo inalterate le prestazioni di rete.

3.5 Topology Sensitivity

Uno dei limiti più stringenti dell'implementazione originale risiedeva nella sua rigida dipendenza dalla topologia di rete. Il sistema di partenza era stato concepito e calibrato per funzionare esclusivamente su una rete lineare e di piccole dimensioni, rendendolo inadatto a scenari reali in cui il numero di switch e i collegamenti possono variare dinamicamente.

La nuova architettura risolve questo problema alla radice rendendo il controller completamente agnostico rispetto alla conformazione della

rete. Questo risultato è stato ottenuto sfruttando la registrazione dinamica dei datapath (tramite gli eventi OpenFlow EventOFPStateChange) e la struttura centralizzata shared_data.datapaths. Invece di fare affidamento su percorsi o porte preimpostate, i moduli di monitoraggio (TrafficMonitor) e di analisi (PolicyEngine) iterano in tempo reale unicamente sugli switch effettivamente connessi e attivi.

Di conseguenza, il meccanismo di calcolo delle metriche, la valutazione delle soglie dinamiche e l'inserimento delle regole di drop o forwarding si adattano automaticamente a qualsiasi infrastruttura di base. Questa generalizzazione del codice non solo soddisfa il requisito di superare la sensibilità topologica, ma prepara il sistema a scalare senza alcuna modifica software. Come verrà ampiamente dimostrato nel seguente capitolo dedicato alle simulazioni, il controller è ora in grado di gestire in totale autonomia topologie ben più stratificate e complesse rispetto a quella di partenza.

Chapter 4

Simulazioni

4.1 Ambiente di Test e Strumenti

Per validare l'efficacia delle soluzioni implementate e descritte nei capitoli precedenti, è stato predisposto un ambiente di simulazione basato su Mininet per l'emulazione della rete e Ryu come framework per il controller SDN.

Gli strumenti e i comandi principali utilizzati per la generazione del traffico e la verifica delle policy includono:

- **Ping**: utilizzato per testare la connettività ICMP end-to-end e dimostrare che, a differenza dell'implementazione originale, il traffico legittimo non subisce interruzioni durante i blocchi selettivi.
- **iPerf**: impiegato per generare flussi di traffico UDP ad alta intensità, simulando attacchi DoS (Denial of Service) e verificando

la reattività della dynamic threshold.

Le simulazioni sono state condotte su due scenari topologici distinti per dimostrare l'adattabilità del controller:

- **Topologia di Base** (topology.py): Rappresenta l'infrastruttura di rete di partenza utilizzata per i primi tre scenari di test (risoluzione dell'overblocking, rilevamento adattivo e penalità progressiva). È composta da 4 switch e 4 host organizzati a stella intorno a un nodo di aggregazione centrale (s3). Gli host client h1 e h4 sono attestati sullo switch s1, h2 su s2, mentre il server vittima h3 è collegato a s4. Per consentire stress-test più realistici, le capacità dei link sono state modificate rispetto alla configurazione standard: i collegamenti core tra gli switch operano a 100 Mbps con 25 ms di ritardo, così come i link di accesso verso i client (100 Mbps, 2 ms). Una particolarità fondamentale di questa architettura è l'introduzione di un collo di bottiglia intenzionale: la capacità del link di accesso del server h3 è limitata a 40 Mbps. Questa specifica asimmetria di banda si è rivelata essenziale durante le simulazioni per dimostrare l'efficacia del meccanismo a soglia dinamica, permettendo di distinguere matematicamente un trasferimento TCP legittimo (che si autoregola sui 40 Mbps del collo di bottiglia) da un attacco volumetrico impulsivo in grado di saturare il link locale dell'attaccante a 100 Mbps.

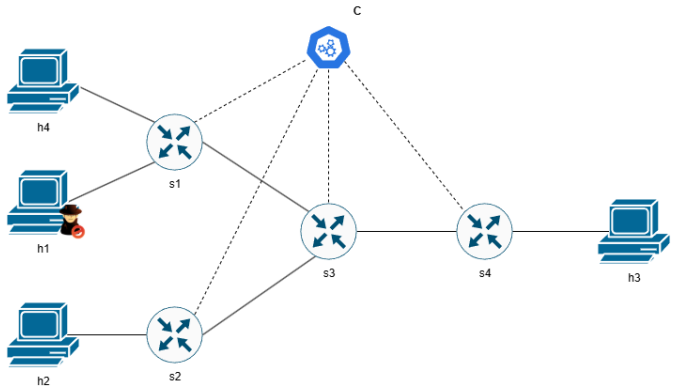


Figure 4.1: Topologia di partenza

- **Topologia Complessa** (complex_topology.py): Sviluppata ex-novo per dimostrare che il sistema ha superato il limite della topology sensitivity e per testare la mitigazione di attacchi distribuiti (multi-source blocking). Come illustrato nel diagramma di rete, l'infrastruttura è composta da 10 switch organizzati in tre diramazioni parallele principali che si originano da un nodo radice (s1). All'interno di questa rete operano 9 host, distribuiti strategicamente sui vari rami per creare un ecosistema realistico: 3 nodi fungono da attaccanti (attacker1 su s2, attacker2 su s6, attacker3 su s7), garantendo che le minacce provengano da percorsi disgiunti; 6 nodi costituiscono l'utenza legittima, con h3 (collegato a s9) che agisce come server vittima in ascolto, mentre gli host h1, h2, h4, h5 e h6 (attestati rispettivamente sugli switch s5, s8, s3, s4 e s10) rappresentano i client. I collegamenti core tra gli switch prevedono una larghezza di banda di 150 Mbps e un ritardo di 25 ms, mentre i link di accesso

verso gli host sono configurati a 100 Mbps con 2 ms di ritardo. Questo scenario stratificato e asimmetrico permette di verificare la tenuta del controller, dimostrando la sua capacità di mappare dinamicamente i datapath e orchestrare difese complesse contro attacchi simultanei in totale autonomia, senza fare affidamento su configurazioni topologiche inserite staticamente a codice.

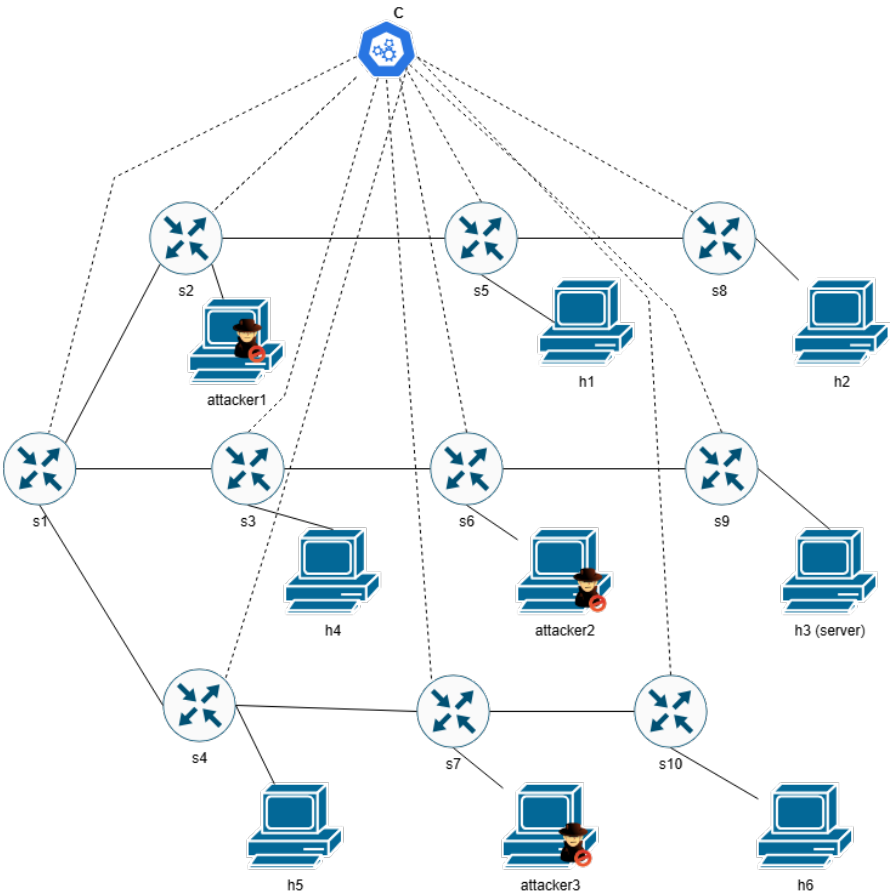


Figure 4.2: Topologia Complessa

4.2 Scenario 1: Risoluzione dell'Overblocking

Il primo scenario di simulazione è stato progettato per validare la risoluzione del problema dell'overblocking. Nel controller originale, il superamento della soglia di traffico comportava l'inserimento di una regola di blocco sull'intera porta dello switch, interrompendo indiscriminatamente le comunicazioni di tutti gli host attestati su di essa. Per dimostrare l'avvenuto passaggio a una mitigazione con granularità a livello di flusso , è stata utilizzata la topologia di base in cui gli host *h1* e *h4* condividono il medesimo switch di accesso (*s1*). La simulazione prevede che l'host *h1* avvii un attacco volumetrico di tipo UDP verso il server h3, mentre l'host *h4* intrattiene una comunicazione ICMP legittima sempre verso *h3*.

Come si evince dai log del controller, non appena il traffico anomalo generato da *h1* supera la soglia dinamica, il motore delle policy rileva la violazione e istruisce il modulo di enforcement. Quest'ultimo applica tempestivamente una regola di drop mirata esclusivamente all'indirizzo IP sorgente dell'attaccante (10.0.0.1) per il traffico UDP.

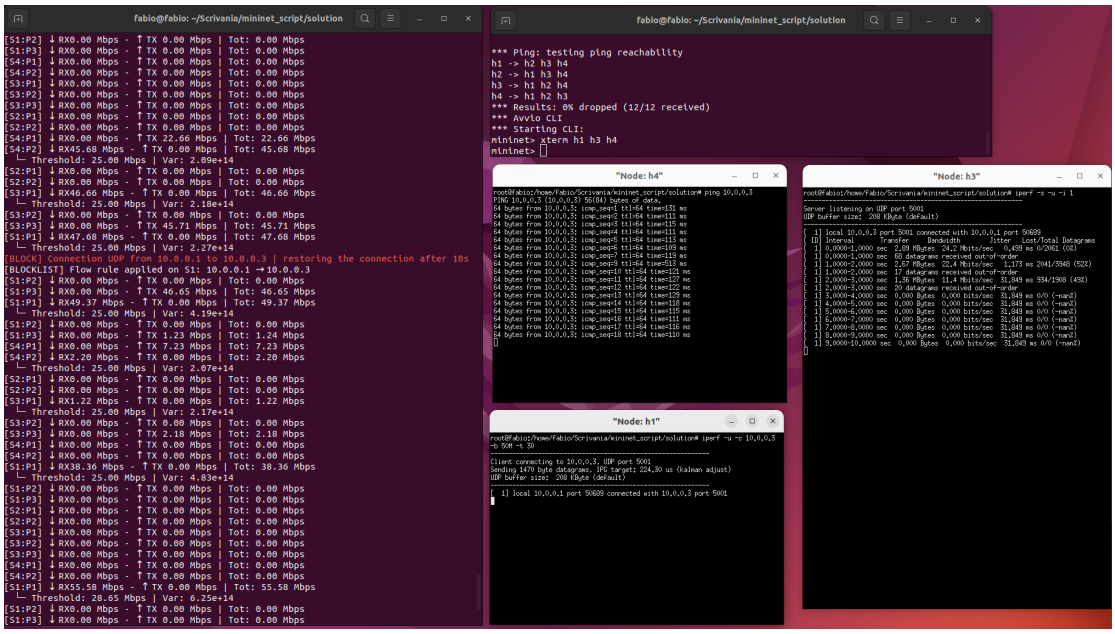


Figure 4.3: Scenario 1

Contestualmente, l'osservazione del traffico generato dall'host legittimo *h4* conferma l'assenza di disservizi. Le richieste ICMP (Ping) continuano a raggiungere la destinazione *h3* ricevendo risposta senza alcun calo di pacchetti o aumento significativo della latenza, a dimostrazione del fatto che la porta dello switch *s1* continua a inoltrare correttamente il traffico non malevolo.

Per fornire una validazione a livello di data plane e dimostrare empiricamente il funzionamento del meccanismo di isolamento, è stato ispezionato lo stato della tabella di flusso dello switch di accesso *s1* (tramite l'utility `ovs-ofctl dump-flows`) in tre fasi distinte del test:

- **Prima dell'attacco:** Lo switch presenta esclusivamente le re-

gole di base per l'inoltro di livello 2 (apprendimento dei MAC address) e la regola di default per l'invio dei pacchetti sconosciuti al controller (`priority=0, actions=CONTROLLER`).

- **Durante l'attacco (Mitigazione):** Non appena il *PolicyEngine* rileva l'anomalia, viene installata una nuova flow rule ad alta priorità (`priority=100`). Come visibile in Figura 4.4, la regola esegue un matching estremamente selettivo a livello di rete e di trasporto (`udp, nw_src=10.0.0.1, nw_dst=10.0.0.3`) e vi applica l'azione `drop`. Questa granularità è la ragione tecnica per cui il traffico ICMP dell'host *h4* non subisce interferenze.

- **Dopo il blocco (Recovery):** Allo scadere del timer di penalità, il *FlowEnforcer* interviene nuovamente sulla tabella di flusso. Come mostrato in Figura 4.5, la regola precedentemente restrittiva viene aggiornata con `actions=NORMAL`, rimuovendo il drop e delegando nuovamente l'inoltro alla normale pipeline hardware dello switch.

Figure 4.4: Scenario 1: Tabella di flusso di s1 durante l'attacco (Regola di Drop mirata)

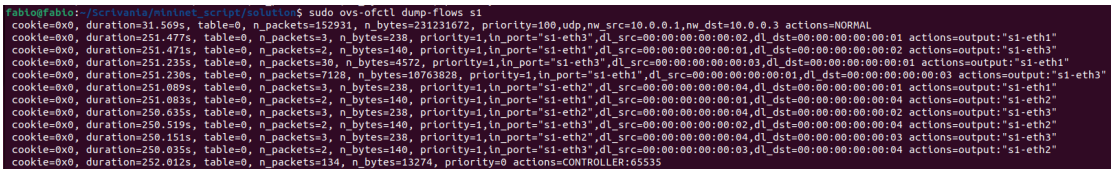


Figure 4.5: Scenario 1: Tabella di flusso di s1 al termine della penalità (Ripristino azioni)

4.3 Scenario 2: Rilevamento Adattivo (Traffico Legittimo vs Attacco Bursty)

Uno dei limiti più critici dell'architettura originale risiedeva nell'utilizzo di una soglia statica preimpostata. Questo approccio generava un duplice problema: da un lato risultava vulnerabile agli attacchi "bursty" (picchi improvvisi sotto-soglia), dall'altro rischiava di bloccare falsi positivi, come ad esempio host legittimi impegnati nel download di file pesanti che richiedevano un elevato throughput.

Per superare questa vulnerabilità, il modulo PolicyEngine calcola una dynamic threshold (soglia dinamica) basata sulla media mobile e sulla varianza dei campioni di traffico. L'obiettivo di questo scenario è dimostrare la capacità del controller di distinguere un picco di traffico legittimo (come un trasferimento TCP) da un attacco volumetrico impulsivo (UDP flood).

Durante questa simulazione, l'host *h1* ha inizialmente avviato un trasferimento TCP ad alta intensità verso il server. Come si evince dai

log, il meccanismo di controllo della congestione del TCP ha permesso un incremento graduale del traffico: la soglia dinamica del controller si è adattata in tempo reale all'aumento di banda, mantenendo la varianza sotto i livelli di allerta ed evitando l'overblocking della connessione legittima.

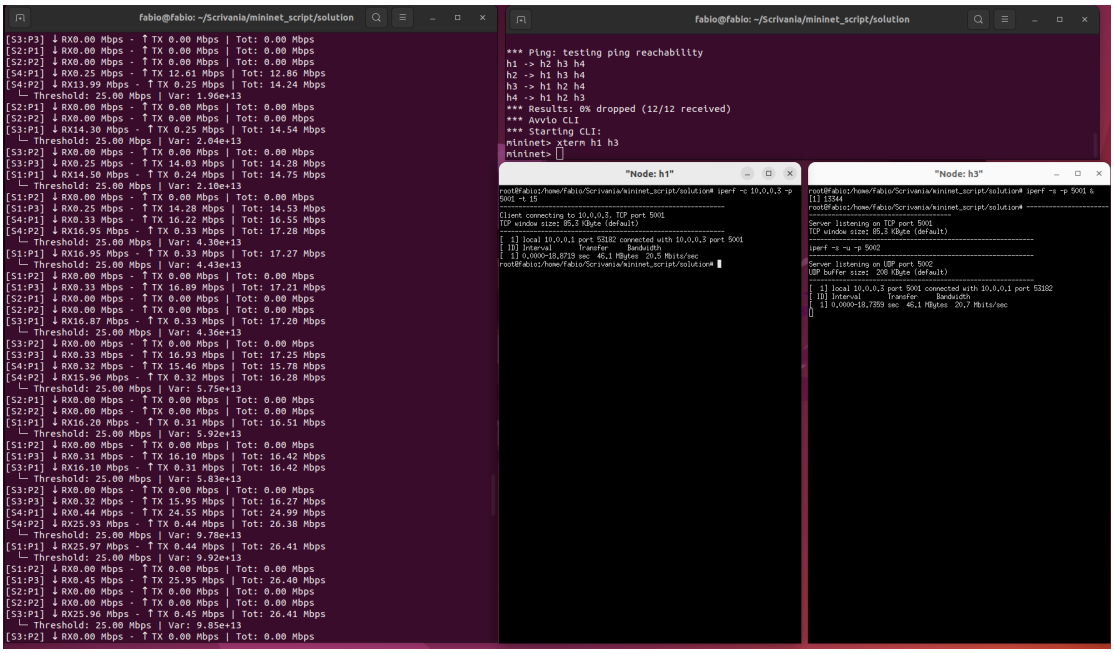


Figure 4.6: Scenario 2 - Prima Parte

Successivamente, lo stesso host ha interrotto il trasferimento e ha lanciato un attacco UDP di tipo bursty. In questo caso, l'assenza di controllo di flusso ha generato un picco istantaneo e sregolato. Il throughput ha superato di oltre il 20% la soglia dinamica correntemente calcolata e la varianza ha ecceduto il limite consentito (var_threshold). Il motore delle policy ha quindi classificato istantaneamente il flusso come sospetto, innescando l'applicazione della regola

di blocco.

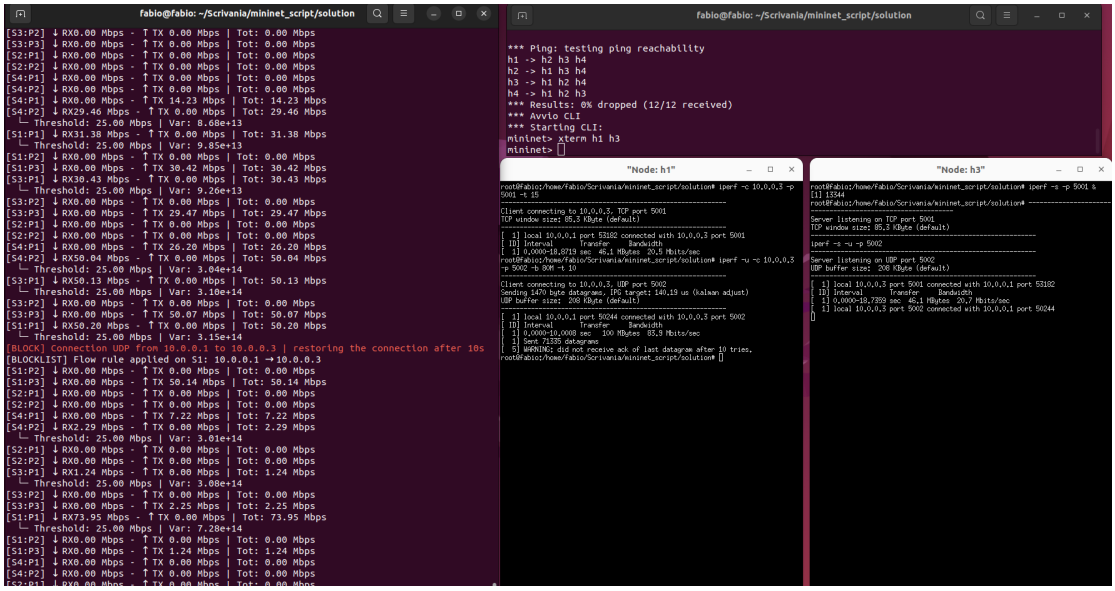


Figure 4.7: Scenario 2 - Seconda Parte

Questo dimostra come il rilevamento adattivo garantisca un bilanciamento ottimale tra sicurezza (reazione rapida agli attacchi) ed efficienza (tutela delle operazioni di rete ad alto carico).

Per fornire un'ulteriore e definitiva dimostrazione visiva delle capacità di discriminazione del modulo PolicyEngine, è stata eseguita una seconda simulazione in cui flussi di natura opposta coesistono simultaneamente sulla rete. In questo test, l'host *h2* ha avviato un flusso TCP legittimo verso il server, mentre in parallelo l'host *h1* ha iniettato un attacco UDP di tipo bursty.

L'analisi del traffico sull'interfaccia del server, catturata tramite Wireshark (Figura 4.3), illustra perfettamente il comportamento del sistema. La curva verde, rappresentante la connessione TCP, mostra il

fisiologico incremento iniziale per poi stabilizzarsi e continuare ininterrottamente per tutta la durata del test. Al contrario, la curva rossa, relativa al flood UDP, genera un picco improvviso e non negoziato. Nonostante i due flussi stiano competendo per le medesime risorse di rete, il controller isola l'instabilità statistica del traffico UDP, azzerandolo tempestivamente (intorno al secondo 3.5) tramite l'applicazione della regola di blocco, senza causare alcuna interruzione al trasferimento TCP in corso.

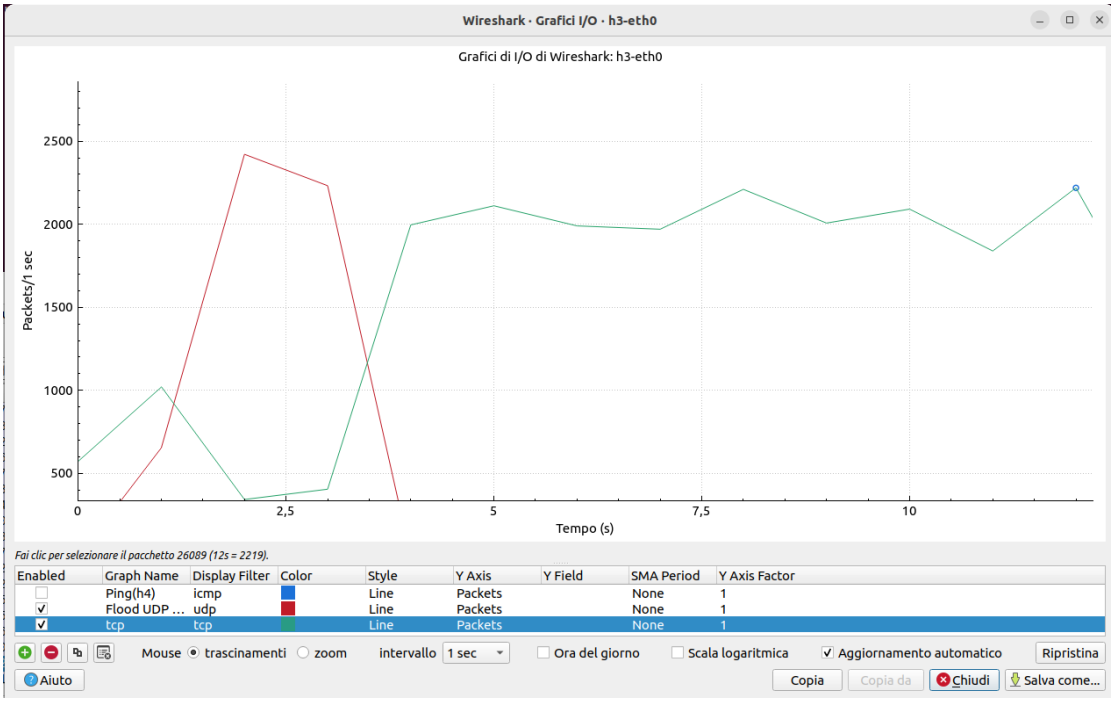


Figure 4.8: Scenario 2 - Traffico misto concorrente (Wireshark I/O Graph)

Questo risultato conferma in modo inequivocabile che la valutazione della varianza consente una mitigazione altamente selettiva, tutelando la continuità operativa dei flussi TCP legittimi anche durante scenari

di attacco in corso.

4.4 Scenario 3: Penalità Progressiva e Recovery Adattivo

Nell'implementazione originale, il meccanismo di mitigazione presentava una notevole rigidità: i blocchi risultavano inflessibili e non prevedevano una politica di ripristino intelligente. Questo approccio rischiava di penalizzare a tempo indeterminato host legittimi in caso di falsi positivi, oppure di consentire a veri attaccanti di riprendere le ostilità non appena rimossa manualmente la regola, per via della mancanza di uno storico delle violazioni.

Per risolvere questa criticità, il modulo *FlowEnforcer* implementa una logica di Recovery Adattivo unita a un meccanismo di penalità progressiva basato su un backoff esponenziale. La memoria storica delle violazioni per ogni singolo nodo viene mantenuta in tempo reale all'interno del dizionario condiviso *shared_ data.block_ counts*. L'obiettivo di questo scenario di test è dimostrare che il sistema applica blocchi di durata crescente agli host recidivi, garantendo al contempo una finestra di sblocco automatico in caso di violazioni isolate.

Durante la simulazione, l'host attaccante *h1* ha generato un primo flusso UDP anomalo. Il controller ha tempestivamente rilevato l'anomalia e ha bloccato l'host per una durata base calcolata di 10 secondi ($2^1 \times$

BASE_DELAY). Al termine di tale intervallo, come visibile dai log, il sistema ha rimosso automaticamente la regola di drop (messaggio di [UNBLOCK]), ripristinando il normale inoltro dei pacchetti e dimostrando la capacità di auto-guarigione della rete.

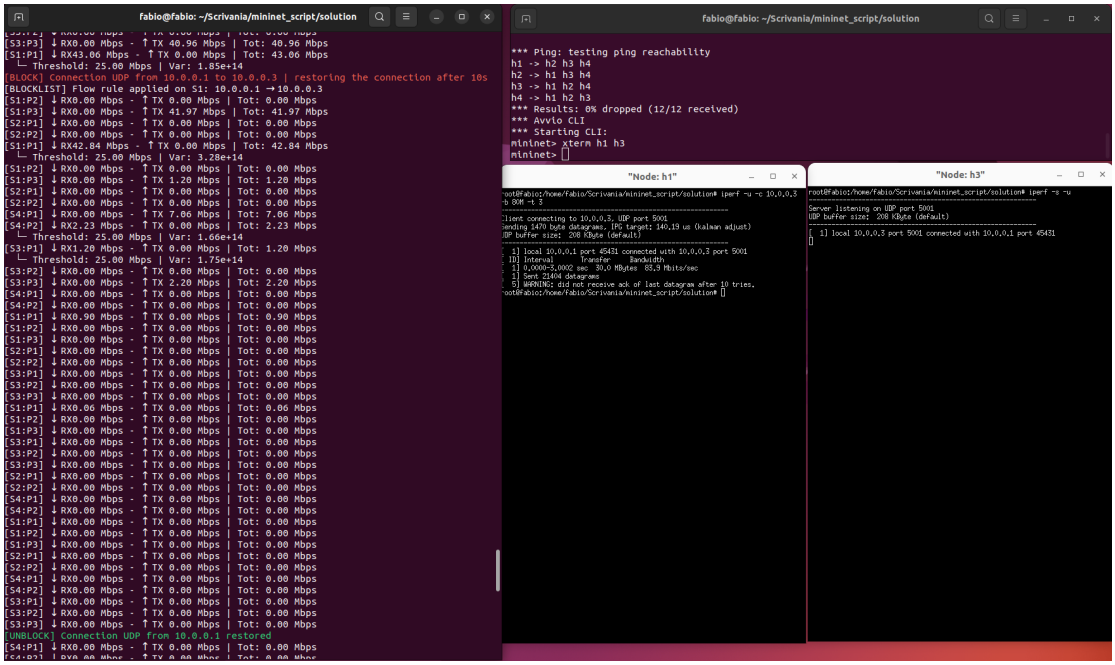


Figure 4.9: Scenario 3 - Prima Parte

Subito dopo lo sblocco, lo stesso host ha reiterato l'attacco. Questa volta, consultando lo storico delle violazioni, il sistema ha applicato un fattore di moltiplicazione esponenziale al calcolo della penalità, raddoppiando la durata del blocco a 20 secondi ($2^2 \times$ BASE_DELAY).

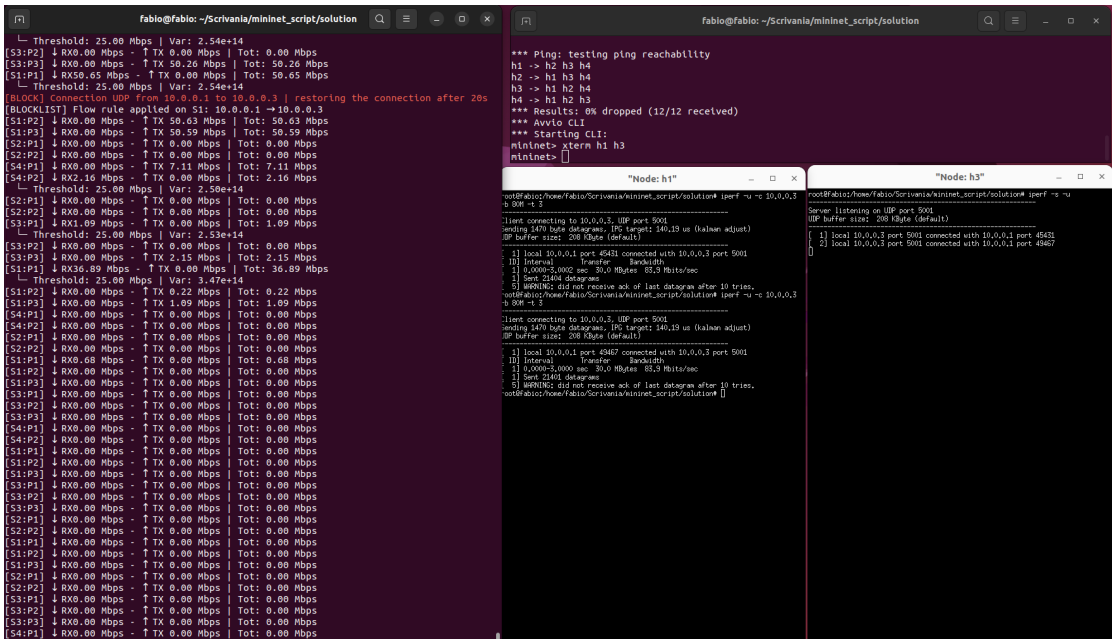


Figure 4.10: Scenario 3 - Seconda Parte

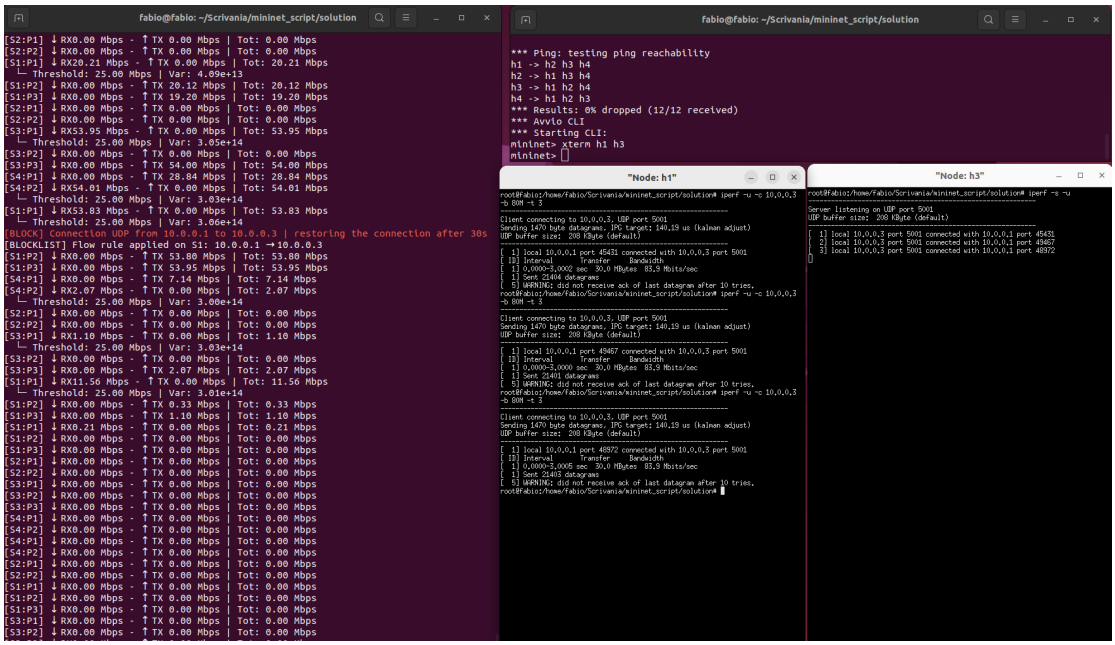


Figure 4.11: Scenario 3 - Terza Parte

Questo approccio flessibile garantisce equità, limitando i disservizi per i falsi positivi occasionali, ma opponendo una difesa proporzional-

mente sempre più severa e duratura contro gli attaccanti persistenti.

4.5 Scenario 4: Adattabilità Topologica e Mitigazione Multi-Sorgente

L'ultimo scenario di test è stato concepito per validare due tra i miglioramenti architetturali più significativi apportati al controller: il superamento della sensibilità topologica (topology sensitivity) e la capacità di gestire attacchi distribuiti provenienti da nodi differenti (multi-source blocking).

Il controller originale risultava strettamente accoppiato a una rete lineare predefinita e limitata. Grazie alla registrazione dinamica dei datapath e all'uso di strutture dati condivise (shared_data.datapaths), il nuovo sistema è in grado di mappare e proteggere reti di qualsiasi dimensione e forma in modo del tutto agnostico. Per dimostrarlo, è stata istanziata una topologia complessa sviluppata ex-novo (complex_top.py). Come si evince dal relativo diagramma (4.1), l'infrastruttura è composta da 10 switch organizzati in tre diramazioni parallele che si originano dal nodo radice (s1). All'interno della rete operano 9 host (6 legittimi e 3 malevoli) distribuiti su rami e switch differenti.

Per soddisfare il requisito di difesa contro minacce distribuite, la simulazione ha previsto un attacco volumetrico UDP simultaneo generato dai tre host attaccanti posti su rami disgiunti: *attacker1* (attestato

su s2), *attacker2* (attestato su s6) e *attacker3* (attestato su s7), tutti convergenti verso il server vittima h3 (collegato allo switch s9).

Come confermato dai log del controller, il PolicyEngine ha monitorato in parallelo tutte le porte attive dell'infrastruttura a 10 switch. Non appena i tre flussi anomali hanno saturato le rispettive soglie dinamiche locali, il sistema ha elaborato le violazioni in modo concorrente e indipendente. Il FlowEnforcer ha tempestivamente installato tre distinte regole di blocco mirate, intervenendo in modo decentralizzato esclusivamente sui tre switch di accesso periferici interessati.

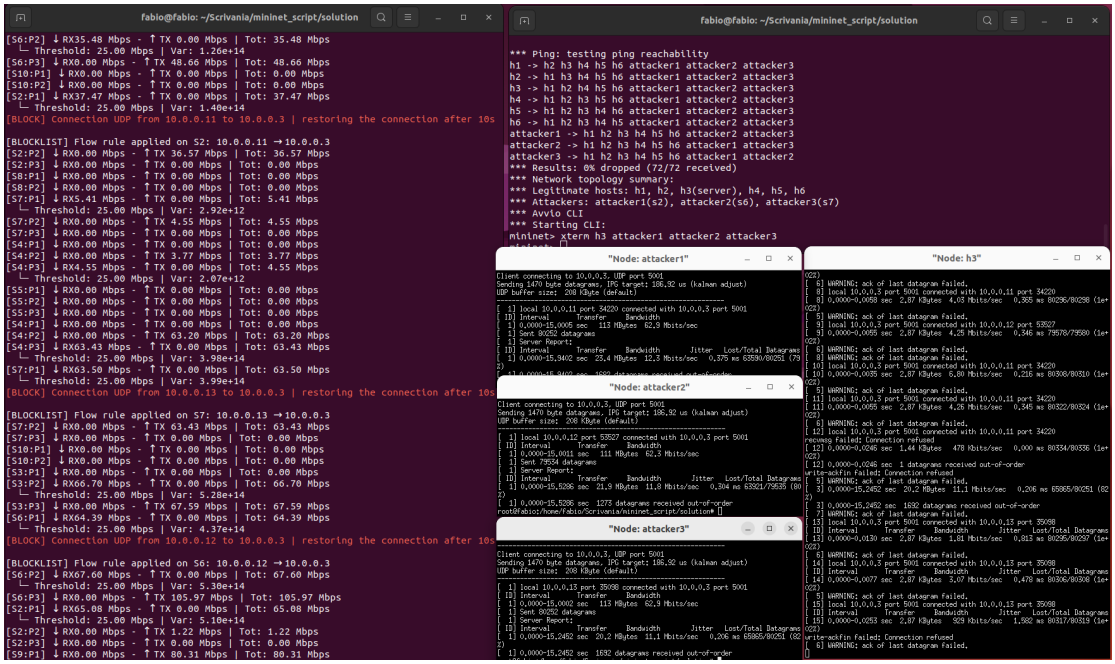


Figure 4.12: Scenario 4

Questa simulazione dimostra in modo inequivocabile la scalabilità orizzontale della soluzione proposta: il sistema orchestra difese complesse contro minacce distribuite e si adatta a infrastrutture di

grandi dimensioni senza richiedere alcuna modifica al codice sorgente o l'inserimento manuale di parametri topologici.