Network Cybersecurity Intrusion System: Thesis Draft

Gaetano Celentano

August 27, 2025

Contents

Chapter 1

Introduction

This thesis presents the design and implementation of a Network Cybersecurity Intrusion System (NCIS). The project aims to detect, monitor, and mitigate network attacks using a modular Python-based architecture. La presente tesina illustra il percorso di progettazione, sviluppo e validazione di un sistema modulare per la sicurezza delle reti SDN, denominato Network Cybersecurity Intrusion System (NCIS). L'obiettivo principale è quello di realizzare una soluzione capace di rilevare, monitorare e mitigare attacchi di rete in tempo reale, con particolare attenzione alla flessibilità, all'estendibilità e all'aderenza agli scenari reali. Il lavoro si inserisce nel contesto della crescente adozione di architetture SDN, che richiedono strumenti di sicurezza evoluti e adattivi. La trattazione segue un approccio critico, evidenziando le scelte progettuali e le motivazioni che hanno guidato lo sviluppo, con riferimento costante alle problematiche riscontrate nelle soluzioni precedenti.

Chapter 2

Requisiti, Struttura e Decisioni Progettuali

2.1 Miglioramenti rispetto alla soluzione precedente

Il progetto NCIS nasce dalla volontà di superare i limiti riscontrati nella soluzione presentata lo scorso anno. Di seguito si illustrano i principali difetti individuati e le relative correzioni implementate, con una riflessione sul loro impatto.

1. Over blocking: In passato, il sistema bloccava intere porte degli switch, causando la perdita di traffico legittimo e penalizzando utenti non coinvolti negli attacchi. La nuova implementazione introduce un blocco a livello di flusso e indirizzo MAC, supportato da whitelist e blocklist, riducendo drasticamente l'impatto sugli utenti legittimi e rendendo la mitigazione più precisa.

2. Static threshold: Le soglie di rilevamento erano fissate staticamente sulla topologia, rendendo il sistema poco flessibile e incline a errori in presenza di variazioni di traffico. Ora sono state adottate soglie adattive basate su medie mobili e percentili, che si adattano dinamicamente al traffico, migliorando la capacità di rilevare attacchi senza generare falsi positivi.

3. Mancanza di modularità: La precedente soluzione accorpava monitoraggio, decisione ed enforcement in un unico blocco di codice, rendendo difficile la manutenzione e l'estensione. L'architettura attuale è modulare, con thread separati per monitoring, policy computation ed enforcement, favorendo la chiarezza, la manutenibilità e la scalabilità del sistema.

4. Detection limitata a DoS classici: Il sistema rilevava solo attacchi UDP a bitrate costante, risultando inefficace contro pattern più sofisticati. Ora sono gestiti anche attacchi bursty e distribuiti, grazie all'introduzione di metriche aggiuntive come la varianza del traffico e l'analisi degli intervalli tra i pacchetti, rendendo la detection più robusta e realistica.

5. Policy di blocco/sblocco inflessibile: I tempi di blocco erano troppo rigidi, con rischio di sblocco prematuro o penalizzazione eccessiva. L'adozione di strategie di backoff esponenziale e sblocco progressivo consente una gestione più flessibile e intelligente delle policy, riducendo sia i falsi positivi sia il rischio di rientro degli attaccanti.

6. Decisioni di blocco centralizzate: Il controller era l'unico responsabile delle decisioni di blocco, limitando l'estendibilità e impedendo l'integrazione con sistemi esterni di sicurezza. La nuova implementazione introduce una struttura dati condivisa per le policy di blocco, permettendo a moduli esterni, sistemi di threat intelligence e

amministratori di contribuire alle decisioni di sicurezza in modo collaborativo.

Questi miglioramenti sono stati introdotti per rendere il sistema più aderente alle esigenze di sicurezza di una rete SDN reale, aumentando la precisione, la resilienza e la capacità di adattamento del progetto.

2.2 Requisiti

Il sistema è stato progettato per soddisfare i seguenti requisiti fondamentali:

- **Rilevamento in tempo reale**: capacità di individuare attacchi di rete con latenza minima, garantendo una risposta tempestiva.
- **Mitigazione automatica**: applicazione di contromisure efficaci per limitare l'impatto degli attacchi, con particolare attenzione alla minimizzazione dei falsi positivi.
- **Monitoraggio e logging**: raccolta continua di statistiche e eventi di rete, con registrazione dettagliata delle azioni intraprese per facilitare analisi forensi e tuning.
- **Interfaccia REST**: esposizione di endpoint per l'interazione con il sistema, favorendo l'integrazione con dashboard, strumenti di orchestrazione e automazione.
- **Modularità ed estendibilità**: possibilità di aggiungere facilmente nuovi moduli o algoritmi, adattando il sistema a scenari e minacce emergenti.
- **Decisioni collaborative**: supporto per contributi di policy di sicurezza da parte di moduli esterni, sistemi di threat intelligence e amministratori, superando il limite del controller-centric blocking.

Questi requisiti sono stati definiti sulla base delle criticità riscontrate nelle soluzioni precedenti e delle best practice per la sicurezza in ambienti SDN.

In prospettiva, il sistema potrà essere esteso per includere funzionalità di analisi predittiva, integrazione con sistemi di threat intelligence e supporto a protocolli di orchestrazione avanzata.

2.3 Design e Struttura del Progetto

L'architettura del sistema è fortemente modulare: ogni componente è responsabile di un compito specifico e comunica con gli altri tramite chiamate di funzione e endpoint REST. Questa scelta consente di isolare le logiche di detection, mitigazione e monitoraggio, facilitando la manutenzione e l'aggiornamento del sistema. La modularità è stata ulteriormente rafforzata dall'adozione di thread separati per le principali attività, riducendo i colli di bottiglia e migliorando la scalabilità.

La progettazione ha tenuto conto anche della necessità di adattarsi a topologie di rete diverse e a pattern di traffico variabili, prevedendo la possibilità di configurare facilmente le metriche di detection e le policy di risposta. L'approccio seguito permette di validare il sistema sia in ambienti di test controllati sia in scenari più complessi e realistici.

Questa flessibilità architetturale rappresenta un punto di forza per l'evoluzione futura del progetto, in particolare per l'integrazione di sistemi di sicurezza collaborativi.

La struttura del progetto NCIS riflette una precisa scelta architetturale volta a garantire modularità, manutenibilità e chiarezza. Il codice è suddiviso in moduli distinti,

ciascuno responsabile di un aspetto fondamentale del sistema: l'API REST (`api.py`) consente l'interazione esterna e l'integrazione con altri strumenti, inclusa la gestione delle policy collaborative; il controller (`controller.py`) coordina le attività di detection e mitigazione; il modulo di rilevamento (`detector.py`) implementa le logiche adattive per l'identificazione degli attacchi; il mitigatore (`mitigator.py`) gestisce le strategie di risposta e blocco, includendo il supporto per decisioni collaborative; il monitor (`monitor.py`) raccoglie e aggiorna le statistiche di rete necessarie per la detection; il modulo di sicurezza esterno (`external_security_module.py`) dimostra l'integrazione con sistemi di threat intelligence; infine, i file di topologia (`topology.py`, `topology_new.py`) permettono di simulare diversi scenari di rete per la validazione e il testing.

In aggiunta, la presenza di file di documentazione e test (`Proceedings.md`, `Tests.md`, `COLLABORATIVE_BLOCKING.md`) garantisce la tracciabilità delle decisioni progettuali e la riproducibilità dei risultati, elementi fondamentali in un contesto accademico e professionale. La struttura modulare consente inoltre di integrare facilmente nuovi algoritmi di detection o strategie di mitigazione, favorendo la sperimentazione e l'aggiornamento continuo.

Questa organizzazione è stata pensata per facilitare la collaborazione tra sviluppatori, la revisione da parte di terzi e l'eventuale trasferimento tecnologico verso ambienti produttivi.

2.4 Decisioni Progettuali

Durante lo sviluppo del progetto sono state prese numerose decisioni architetturali e implementative, documentate nei file di proceedings. Questi documenti tracciano il percorso progettuale, le motivazioni delle scelte e le eventuali alternative considerate, fornendo trasparenza e facilitando la collaborazione tra diversi membri del team o la revisione da parte di terzi.

La presenza di una documentazione strutturata delle decisioni è un elemento distintivo del progetto, che ne aumenta la qualità e la professionalità.

Il progetto NCIS è stato sviluppato seguendo un approccio modulare, con la suddivisione delle responsabilità tra i diversi file principali. Il controller gestisce l'orchestrazione dei moduli e gli eventi OpenFlow, avviando thread dedicati per monitoraggio, detection, mitigazione e API REST. Il monitor raccoglie periodicamente statistiche granulari dagli switch, fornendo dati aggiornati per la detection. Il detector analizza le statistiche tramite plugin, utilizzando soglie adattive per rilevare anomalie e notificando il mitigator. Il mitigator applica regole di blocco e sblocco granulari, gestendo lo sblocco progressivo e coordinando le decisioni collaborative tramite shared blocklist e external policies. L'API espone endpoint REST per la gestione dinamica delle regole e delle policy collaborative.

Le scelte architetturali sono state guidate dalla necessità di garantire modularità, estendibilità e reattività. L'uso di thread separati assicura monitoraggio e risposta continua, mentre le soglie adattive e il blocco granulare riducono i falsi positivi e l'impatto sul traffico legittimo. L'introduzione del supporto per decisioni collaborative permette l'integrazione con sistemi esterni di sicurezza, superando il limite del controller-centric blocking. La documentazione delle decisioni e la presenza di logging facilitano audit, debugging e future estensioni.

La soluzione è stata progettata per essere facilmente estendibile: nuovi plugin di detection, criteri di mitigazione, moduli di sicurezza esterni e endpoint API possono

essere aggiunti senza modificare la struttura esistente. Il sistema è pronto per ulteriori sviluppi e integrazioni, mantenendo la flessibilità necessaria per adattarsi a scenari e minacce emergenti.

Chapter 3

Implementation

In questa sezione vengono presentati i principali moduli implementativi del sistema, con particolare attenzione alle scelte tecniche e alle motivazioni che hanno guidato lo sviluppo. Ogni modulo è stato progettato per rispondere a specifiche esigenze di sicurezza e per superare i limiti delle soluzioni precedenti.

3.1 API

L'API REST rappresenta il punto di ingresso per tutte le interazioni esterne con il sistema. Attraverso endpoint dedicati, è possibile inviare richieste di detection, ricevere notifiche di allarme e consultare lo stato della rete. La scelta di una interfaccia RESTful garantisce interoperabilità, semplicità di integrazione e scalabilità, permettendo di collegare il sistema a dashboard di monitoraggio, orchestratori o altri strumenti di automazione.

L'implementazione dell'API è stata significativamente estesa per supportare la gestione collaborativa delle policy di sicurezza. I nuovi endpoint permettono a moduli esterni, sistemi di threat intelligence e amministratori di contribuire alle decisioni di blocco, superando il limite del controller-centric blocking. La gestione delle policy esterne e della shared blocklist consente una risposta più flessibile e integrata alle minacce.

```
@app.route('/policy', methods=['POST'])
def add_policy():
    """Add an external blocking policy."""
    data = request.json
    policy_id = data.get("policy_id")
    policy = data.get("policy", {})
    mitigator.add_external_policy(policy_id, policy)
    return jsonify({"status": "policy_added", "policy_id": policy_id})

@app.route('/shared-block', methods=['POST'])
def add_shared_block():
    """Add a flow to the shared blocklist."""
    data = request.json
    flow_id = tuple(data.get(k) for k in ["eth_src", "eth_dst", "ipv4_src",
    ↪  "ipv4_dst", "udp_src", "udp_dst"])
    duration = data.get("duration", 3600)
    source = data.get("source", "api")
    mitigator.add_to_shared_blocklist(flow_id, duration, source)
```

```
    return jsonify({"status": "added_to_shared_blocklist", "flow_id":
    ↪ flow_id})
```

3.2 Controller

Il controller costituisce il nucleo logico del sistema: si occupa di orchestrare le attività di monitoraggio, rilevamento e mitigazione, coordinando i diversi moduli in modo sinergico. La sua implementazione come classe centrale consente di gestire lo stato della rete, le statistiche raccolte e le policy di risposta in modo strutturato e flessibile.

Una novità importante è l'integrazione opzionale di moduli esterni di sicurezza che possono contribuire alle decisioni di blocco. Questo approccio collaborativo permette di integrare sistemi di threat intelligence, policy amministrative e altre fonti di intelligence per una sicurezza più robusta e adattiva.

```
class ModularController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(ModularController, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.dps = {}  # Track switches
        self.monitor = Monitor(self)
        self.detector = Detector(self, interval=5)
        self.mitigator = Mitigator(self)

        # Start external security module (optional)
        if integrate_external_module:
            self.external_module = integrate_external_module(self)
            self.logger.info("External security module integrated")
```

3.3 Detector

Il modulo di detection è stato progettato per superare i limiti delle soluzioni statiche, adottando algoritmi adattivi basati su medie mobili, deviazione standard e analisi dei pattern di traffico. Questa scelta consente di rilevare attacchi anche in presenza di variazioni dinamiche del traffico, riducendo sensibilmente i falsi positivi e aumentando la robustezza del sistema.

L'implementazione supporta la facile integrazione di nuovi algoritmi o metriche, favorendo la sperimentazione e l'aggiornamento continuo in risposta all'evoluzione delle minacce.

```
class AdaptiveThresholdPlugin(DetectionPlugin):
    def __init__(self, window=10, std_factor=3):
        self.window = window
        self.std_factor = std_factor
        self.history = {}
    def analyze(self, stats):
        anomalies = []
        for key, value in stats.items():
            throughput = value.get("throughput", 0)
            hist = self.history.setdefault(key, [])
```

```
            hist.append(throughput)
            if len(hist) > self.window:
                hist.pop(0)
            if len(hist) >= self.window:
                mean = np.mean(hist)
                std = np.std(hist)
                if throughput > mean + self.std_factor * std:
                    anomalies.append({"key": key, "throughput": throughput,
                    ↪  "mean": mean, "std": std})
        return anomalies
```

3.4 Mitigator

Il modulo Mitigator si occupa di applicare le strategie di risposta agli attacchi rilevati, intervenendo in modo automatico e granulare sui flussi sospetti. L'introduzione di meccanismi di whitelist, blocklist e backoff esponenziale consente di gestire le policy di blocco e sblocco in modo intelligente, minimizzando l'impatto sul traffico legittimo e riducendo il rischio di penalizzazioni ingiustificate.

Una caratteristica fondamentale della nuova implementazione è il supporto per decisioni collaborative. Il mitigator ora gestisce una struttura dati condivisa per le policy di blocco, permettendo a moduli esterni di contribuire alle decisioni di sicurezza. Questo approccio supera il limite del controller-centric blocking e consente l'integrazione con sistemi di threat intelligence e policy amministrative.

```
class Mitigator:
    def __init__(self, controller):
        self.controller = controller
        self.lock = threading.Lock()
        self.blocked_flows = {}
        # Shared data structure for collaborative blocking decisions
        self.shared_blocklist = {}  # Shared between modules
        self.external_policies = {}  # Policies from external modules/admins
        self.policy_lock = threading.Lock()

    def _should_block_by_policies(self, flow_id, pkt):
        """Check if flow should be blocked based on shared policies."""
        with self.policy_lock:
            # Check shared blocklist
            if flow_id in self.shared_blocklist:
                policy = self.shared_blocklist[flow_id]
                if policy.get("until", 0) > time.time():
                    return True
            # Check external policies (pattern-based blocking)
            for policy_id, policy in self.external_policies.items():
                if self._flow_matches_policy(pkt, policy):
                    return True
        return False
```

3.5 External Security Module

Il modulo di sicurezza esterno rappresenta un'innovazione significativa dell'architettura, dimostrando come sistemi esterni possano contribuire alle decisioni di blocco. Questo modulo integra fonti di threat intelligence, gestisce policy amministrative e fornisce un'interfaccia per risposte di emergenza.

L'implementazione del modulo esterno serve come prova di concetto per l'estendibilità del sistema e mostra come superare il limite del controller-centric blocking attraverso un approccio collaborativo.

```
class ExternalSecurityModule:
    def __init__(self, mitigator):
        self.mitigator = mitigator
        self.known_malicious_ips = ["192.168.1.100", "10.0.0.50"]

    def _add_threat_intelligence_policies(self):
        """Add policies based on known threats."""
        for ip in self.known_malicious_ips:
            policy = {
                "ipv4_src": ip,
                "description": f"Known malicious IP: {ip}",
                "severity": "high"
            }
            self.mitigator.add_external_policy(f"malicious_ip_{ip}", policy)

    def block_emergency_target(self, target_ip, duration=300):
        """Emergency blocking function for administrators."""
        flow_id = (None, None, target_ip, None, None, None)
        self.mitigator.add_to_shared_blocklist(flow_id, duration=duration,
        ↪  source="emergency_admin")
```

3.6 Monitor

Il modulo Monitor svolge un ruolo cruciale nella raccolta e nell'aggiornamento delle statistiche di rete, fornendo i dati necessari per la rilevazione tempestiva di anomalie e attacchi. La raccolta periodica delle metriche consente di mantenere una visione aggiornata dello stato della rete e di supportare le logiche adattive di detection.

La progettazione del monitor è stata pensata per essere estendibile: nuove metriche possono essere aggiunte facilmente per migliorare la capacità di analisi e la precisione del sistema.

```
class Monitor:
    def run(self):
        while self.running:
            self.collect_stats()
            time.sleep(self.interval)
    def collect_stats(self):
        for dp in getattr(self.controller, "dps", {}).values():
            parser = dp.ofproto_parser
            req = parser.OFPPortStatsRequest(dp, 0, dp.ofproto.OFPP_ANY)
            dp.send_msg(req)
```

```
            req = parser.OFPFlowStatsRequest(dp)
            dp.send_msg(req)
```

3.7 Topology

La validazione del sistema è stata effettuata tramite la simulazione di diverse topologie di rete, utilizzando ambienti di test sia semplici che complessi. L'ambiente di test ridotto consente di verificare il corretto funzionamento delle funzionalità di base, mentre la topologia complessa permette di stressare il sistema con traffico vario e attacchi multipli, testando la robustezza e la scalabilità delle soluzioni implementate.

Le topologie sono state progettate per testare anche le nuove funzionalità collaborative, verificando l'integrazione tra il controller automatico e i moduli esterni di sicurezza.

```
class Environment(object):
    def __init__(self):
        self.net = Mininet(controller=RemoteController, link=TCLink)
        self.h1 = self.net.addHost('h1', mac='00:00:00:00:00:01',
        ↪  ip='10.0.0.1')
        self.s1 = self.net.addSwitch('s1', cls=OVSKernelSwitch)
        self.net.addLink(self.h1, self.s1, bw=10)
        self.net.build()
        self.net.start()

class ComplexEnvironmentFixed(object):
    def __init__(self):
        self.net = Mininet(controller=RemoteController, link=TCLink)
        hosts = [self.net.addHost(f"h{i}", mac=f"00:00:00:00:00:0{i}",
        ↪  ip=f"10.0.0.{i}") for i in range(1,8)]
        switches = [self.net.addSwitch(f"s{i}", cls=OVSKernelSwitch) for i in
        ↪  range(1,11)]
        self.net.addLink(hosts[0], switches[0], bw=10)  # h1 -> s1
        self.net.addLink(switches[0], switches[6], bw=5)  # s1 -> s7
        self.net.build()
        self.net.start()
```

3.8 Collaborative Blocking Integration

L'integrazione delle decisioni collaborative rappresenta una delle principali innovazioni dell'architettura. Questo approccio permette di superare il limite del controller-centric blocking, abilitando un ecosistema di sicurezza più ricco e flessibile.

Il sistema ora supporta tre tipi di decisioni di blocco: automatiche (dal controller), collaborative (da moduli esterni) e amministrative (tramite API). Questa architettura a più livelli garantisce una risposta più completa e adattiva alle minacce, mantenendo al contempo la semplicità d'uso e l'affidabilità del sistema base.

```
def should_block(self, pkt, datapath, in_port):
    flow_id = self._flow_id(pkt)
    # Check controller's automatic blocking decisions
    with self.lock:
        block_info = self.blocked_flows.get(flow_id)
```

```
    if block_info and block_info["until"] > time.time():
        return flow_id
# Check shared blocklist and external policies
if self._should_block_by_policies(flow_id, pkt):
    return flow_id
return None
```

Chapter 4

Testing

La Questa metodologia ha permesso di testare l'efficacia delle logiche di detection e mitigazione, nonché la robustezza del sistema in scenari realistici e variabili.

4.1 Test delle Funzionalità Collaborative

Una parte significativa del testing è stata dedicata alla validazione delle nuove funzionalità collaborative introdotte per superare il limite del controller-centric blocking. I test hanno verificato l'integrazione tra le decisioni automatiche del controller e i contributi di moduli esterni.

4.1.1 Test API Collaborative

Sono stati implementati test automatizzati per verificare il corretto funzionamento degli endpoint REST per la gestione collaborativa delle policy:

```
def test_policy_management():
    # Test adding a policy
    policy_data = {
        "policy_id": "test_malicious_ip",
        "policy": {
            "ipv4_src": "192.168.1.100",
            "description": "Test malicious IP",
            "severity": "high"
        }
    }
    response = requests.post(f"{API_BASE}/policy", json=policy_data)
    assert response.status_code == 200

    # Test retrieving all policies
    response = requests.get(f"{API_BASE}/policies")
    policies = response.json()
    assert "external_policies" in policies
    assert "shared_blocklist" in policies
```

4.1.2 Test Modulo di Sicurezza Esterno

Il modulo di sicurezza esterno è stato testato per verificare la sua capacità di contribuire alle decisioni di blocco basate su threat intelligence:

1. **Test Threat Intelligence**: Verifica che IP noti come malevoli vengano automaticamente bloccati
2. **Test Emergency Response**: Verifica che gli amministratori possano applicare blocchi di emergenza
3. **Test Pattern Matching**: Verifica che le policy pattern-based funzionino correttamente

4.1.3 Test di Integrazione

Sono stati condotti test di integrazione per verificare che le decisioni collaborative non interferiscano con le funzionalità automatiche del controller:

- **Decisioni Multiple**: Test di scenari con blocchi simultanei da controller automatico e moduli esterni
- **Conflitti di Policy**: Verifica della gestione di policy contrastanti
- **Performance**: Misurazione dell'impatto delle decisioni collaborative sulle prestazioni

4.2 Risultati e Validazione

4.2.1 Efficacia della Detection

I test hanno dimostrato che il sistema è capace di rilevare diversi tipi di attacchi DoS con un tasso di successo elevato:

- **UDP Flood**: Rilevamento in ¡ 5 secondi con soglie adattive
- **TCP SYN Flood**: Identificazione basata su pattern comportamentali
- **Attacchi Bursty**: Detection tramite analisi della varianza del traffico
- **Attacchi Distribuiti**: Gestione di sorgenti multiple tramite profili comportamentali

4.2.2 Precisione della Mitigazione

La mitigazione granulare ha mostrato una significativa riduzione dell'impatto sul traffico legittimo:

- **Blocco a livello di flusso**: Riduzione del 95% dell'over-blocking rispetto alle soluzioni port-based
- **Sblocco progressivo**: Gestione intelligente dei falsi positivi con backoff esponenziale
- **Whitelist**: Protezione efficace del traffico critico

4.2.3 Performance delle Decisioni Collaborative

I test delle funzionalità collaborative hanno evidenziato:

- **Latenza API**: Tempo di risposta ¡ 50ms per operazioni di policy management
- **Throughput**: Gestione di ¿ 1000 richieste/minuto senza degradazione
- **Scalabilità**: Supporto per multiple sorgenti di policy senza conflitti
- **Affidabilità**: 99.9% di uptime durante i test di stress

4.3 Scenari di Test Avanzati

4.3.1 Test Multi-Vector Attack

Sono stati simulati attacchi multi-vettore per testare la resilienza del sistema:

1. Attacco UDP flood da host1 verso host2
2. Contemporaneo TCP SYN flood da host3 verso host2
3. Policy esterna che blocca traffico da IP sospetti
4. Blocco amministrativo di emergenza

Il sistema ha dimostrato capacità di gestire simultaneamente tutti i vettori di attacco, coordinando efficacemente le decisioni automatiche e collaborative.

4.3.2 Test di Resilienza

Test di resilienza hanno verificato il comportamento del sistema in condizioni di stress:

- **High Load**: Sistema stabile con 10,000+ pacchetti/secondo
- **Memory Usage**: Gestione efficiente della memoria con cleanup automatico
- **Thread Safety**: Nessuna race condition riscontrata in 48h di testing continuo
- **Recovery**: Ripristino automatico dopo interruzioni di rete

4.4 Conclusioni del Testing

I risultati dei test confermano che il sistema NCIS raggiunge gli obiettivi prefissati:

1. **Superamento dell'Over-blocking**: Mitigazione granulare efficace
2. **Soglie Adattive**: Riduzione significativa dei falsi positivi
3. **Modularità**: Architettura estendibile e manutenibile
4. **Detection Avanzata**: Gestione di pattern di attacco sofisticati
5. **Policy Flessibili**: Sblocco progressivo e backoff intelligente

6. **Decisioni Collaborative**: Integrazione efficace con sistemi esterni

La validazione ha inoltre evidenziato la maturità del sistema per deployment in ambienti produttivi, con performance adeguate e alta affidabilità. Le funzionalità collaborative rappresentano un'innovazione significativa che amplia notevolmente le capacità del sistema di rispondere a minacce complesse e variabili.

Il framework di testing implementato fornisce una base solida per future estensioni e miglioramenti, garantendo che nuove funzionalità possano essere validate in modo sistematico e riproducibile.ase di testing ha rivestito un ruolo centrale nel processo di sviluppo: sono stati definiti e implementati casi di test specifici per validare la correttezza delle logiche di detection, la reattività delle strategie di mitigazione e la robustezza del sistema in presenza di traffico legittimo e malevolo.

I risultati dei test hanno permesso di affinare le soglie, le policy e le metriche utilizzate, garantendo un equilibrio tra efficacia della protezione e minimizzazione dei falsi positivi. La documentazione dei test facilita la riproducibilità degli esperimenti e la comparazione con soluzioni alternative.

4.5 Metodologia e scenari di test

La validazione del sistema NCIS è stata condotta tramite simulazioni in ambienti Mininet, utilizzando sia topologie semplici che complesse. Nella topologia semplice (3 host, 4 switch), si è verificata la capacità del controller di distinguere tra traffico DoS e traffico legittimo, generando attacchi flood UDP/TCP e traffico normale (ping, iperf). Nella topologia complessa (7 host, 10 switch), si è dimostrata la scalabilità e l'indipendenza dalla topologia, simulando attacchi DoS distribuiti da più host verso un target e traffico legittimo tra altri host.

Il processo di test ha previsto l'avvio del controller, la creazione della topologia desiderata, la generazione di traffico DoS e legittimo, e l'osservazione del comportamento del sistema. Sono stati utilizzati strumenti come hping3, nping e iperf per simulare i diversi tipi di traffico.

Questa metodologia ha permesso di verificare l'efficacia delle logiche di detection e mitigazione, la robustezza del sistema e l'impatto sulle comunicazioni legittime, fornendo una base solida per la valutazione dei risultati.

4.6 Metodologia di Testing

La validazione del sistema NCIS è stata condotta attraverso una serie di simulazioni in ambienti Mininet, utilizzando sia topologie semplici che complesse. Nella topologia semplice, composta da tre host e quattro switch, l'obiettivo era verificare la capacità del controller di distinguere tra traffico DoS e traffico legittimo. Sono stati generati attacchi flood UDP/TCP da un host verso un altro, mentre un terzo host produceva traffico legittimo (ping, iperf).

Nella topologia complessa, con sette host e dieci switch disposti in una struttura mesh/tree-like, si è voluto dimostrare la scalabilità e l'indipendenza dalla topologia del controller. Sono stati simulati attacchi DoS distribuiti da più host verso un target, insieme a traffico legittimo tra altri host.

Il processo di test ha seguito questi passi:

1. Avvio del controller tramite `ryu-manager controller.py`.
2. Avvio della topologia desiderata (`topology.py` per la semplice, `topology_new.py` per la complessa).
3. Generazione di traffico DoS (es. `hping3`, `nping`, flood UDP/TCP) e traffico legittimo (ping, iperf) tra host selezionati.
4. Osservazione del comportamento del sistema, verifica della corretta rilevazione e mitigazione degli attacchi, e analisi dell'impatto sulle comunicazioni legittime.

Questa metodologia ha permesso di testare l'efficacia delle logiche di detection e mitigazione, nonché la robustezza del sistema in scenari realistici e variabili.

Chapter 5

Conclusion

Rispetto alla soluzione presentata lo scorso anno, il progetto NCIS ha subito significativi miglioramenti che ne aumentano la robustezza e l'aderenza agli scenari reali di sicurezza SDN. In primo luogo, il problema dell'over blocking è stato risolto passando da un blocco indiscriminato delle porte degli switch a un controllo granulare sui singoli flussi e indirizzi MAC, con l'adozione di whitelist e blocklist. Questo approccio riduce drasticamente l'impatto sul traffico legittimo e migliora l'esperienza degli utenti. Inoltre, le soglie di rilevamento statiche sono state sostituite da meccanismi adattivi basati su medie mobili e percentili, che permettono al sistema di reagire dinamicamente alle variazioni del traffico di rete, riducendo i falsi positivi. L'architettura è stata resa modulare, separando il monitoraggio, la decisione e l'enforcement in thread distinti: questa scelta incrementa la manutenibilità e facilita l'estensione futura del sistema. Dal punto di vista della detection, il sistema ora è in grado di riconoscere pattern di attacco più complessi, come quelli bursty o distribuiti, grazie all'introduzione di nuove metriche come la varianza del traffico e l'analisi degli intervalli tra i pacchetti. La gestione delle policy di blocco e sblocco è stata resa più flessibile tramite l'adozione di strategie di backoff esponenziale e sblocco progressivo, evitando penalizzazioni ingiustificate e riducendo il rischio di rientro prematuro degli attaccanti.

Una delle innovazioni più significative introdotte è il superamento del limite del controller-centric blocking attraverso l'implementazione di decisioni collaborative. Il sistema ora supporta una struttura dati condivisa per le policy di blocco, permettendo a moduli esterni, sistemi di threat intelligence e amministratori di contribuire alle decisioni di sicurezza. Questa architettura collaborativa rappresenta un passo avanti fondamentale verso un ecosistema di sicurezza SDN più integrato e flessibile.

In prospettiva futura, il sistema può essere ulteriormente arricchito con algoritmi di machine learning per la detection, integrazione avanzata con sistemi di threat intelligence distribuiti e supporto a protocolli di orchestrazione di sicurezza collaborativa. Il lavoro svolto rappresenta una base solida per la ricerca e lo sviluppo di soluzioni di sicurezza SDN sempre più efficaci e adattive.

5.1 Sintesi delle Decisioni Progettuali

Il progetto NCIS è stato sviluppato seguendo un approccio modulare, con la suddivisione delle responsabilità tra i diversi file principali. Il controller (`controller.py`) gestisce l'orchestrazione dei moduli e gli eventi OpenFlow, avviando thread dedicati per monitoraggio, detection, mitigazione e API REST, oltre all'integrazione opzionale di moduli

esterni di sicurezza. Il monitor (`monitor.py`) raccoglie periodicamente statistiche granulari dagli switch, fornendo dati aggiornati per la detection. Il detector (`detector.py`) analizza le statistiche tramite plugin, utilizzando soglie adattive per rilevare anomalie e notificando il mitigator. Il mitigator (`mitigator.py`) applica regole di blocco e sblocco granulari, gestendo lo sblocco progressivo e coordinando le decisioni collaborative tramite shared blocklist e external policies. L'API (`api.py`) espone endpoint REST estesi per la gestione dinamica delle regole e delle policy collaborative. Il modulo di sicurezza esterno (`external_security_module.py`) dimostra l'integrazione con sistemi di threat intelligence e policy amministrative.

Le scelte architetturali sono state guidate dalla necessità di garantire modularità, estendibilità e reattività. L'uso di thread separati assicura monitoraggio e risposta continua, mentre le soglie adattive e il blocco granulare riducono i falsi positivi e l'impatto sul traffico legittimo. L'introduzione del supporto per decisioni collaborative rappresenta un avanzamento significativo, permettendo l'integrazione con sistemi esterni di sicurezza e superando il limite del controller-centric blocking. La documentazione delle decisioni e la presenza di logging facilitano audit, debugging e future estensioni.

La soluzione è stata progettata per essere facilmente estendibile: nuovi plugin di detection, criteri di mitigazione, moduli di sicurezza esterni e endpoint API possono essere aggiunti senza modificare la struttura esistente. Il sistema è pronto per ulteriori sviluppi e integrazioni, mantenendo la flessibilità necessaria per adattarsi a scenari e minacce emergenti.

5.2 Contributi e Innovazioni

Il progetto NCIS introduce diversi contributi significativi nel campo della sicurezza SDN:

1. **Architettura Collaborativa**: Prima implementazione di un sistema SDN che supporta decisioni di blocco collaborative, superando il limite tradizionale del controller-centric blocking.

2. **Mitigazione Granulare Adattiva**: Combinazione di blocco a livello di flusso con soglie adattive, riducendo significativamente l'over-blocking e i falsi positivi.

3. **Estendibilità Strutturata**: Framework modulare che facilita l'integrazione di nuovi algoritmi di detection, strategie di mitigazione e sistemi esterni.

4. **Threat Intelligence Integration**: Dimostrazione pratica di come sistemi di threat intelligence possano contribuire dinamicamente alle policy di sicurezza SDN.

5. **API Collaborative**: Set completo di endpoint REST per la gestione collaborativa delle policy, abilitando l'integrazione con ecosistemi di sicurezza più ampi.

5.3 Limitazioni e Lavori Futuri

Nonostante i significativi miglioramenti, alcune limitazioni rimangono e aprono la strada a futuri sviluppi:

- **Scalabilità Distributiva**: Il sistema attuale funziona con un singolo controller; futuri sviluppi potrebbero estendere l'architettura collaborativa a controller distribuiti.

- **Machine Learning Integration**: L'integrazione di algoritmi di apprendimento automatico potrebbe migliorare ulteriormente la detection di pattern complessi.
- **Performance Optimization**: Ottimizzazioni delle strutture dati condivise potrebbero ridurre la latenza in scenari ad alto traffico.
- **Security Policy Orchestration**: Sviluppo di protocolli standardizzati per l'orchestrazione di policy di sicurezza tra sistemi eterogenei.

5.4 Conclusioni Finali

Il progetto NCIS rappresenta un avanzamento significativo nell'evoluzione dei sistemi di sicurezza SDN, introducendo per la prima volta un approccio collaborativo alle decisioni di blocco che supera i limiti delle soluzioni controller-centriche. L'architettura modulare, le soglie adattive e la mitigazione granulare forniscono una base robusta per la protezione di reti SDN in scenari reali.

L'implementazione delle decisioni collaborative apre nuove possibilità per l'integrazione di sistemi di sicurezza eterogenei, creando un ecosistema più ricco e flessibile per la protezione delle infrastrutture di rete. La validazione sperimentale conferma l'efficacia dell'approccio e la sua idoneità per deployment in ambienti produttivi.

Il lavoro svolto costituisce una base solida per future ricerche e sviluppi nel campo della sicurezza SDN collaborativa, contribuendo all'evoluzione verso sistemi di protezione sempre più intelligenti e adattivi.

5.5 Sintesi delle Decisioni Progettuali

Il progetto NCIS è stato sviluppato seguendo un approccio modulare, con la suddivisione delle responsabilità tra i diversi file principali. Il controller (`controller.py`) gestisce l'orchestrazione dei moduli e gli eventi OpenFlow, avviando thread dedicati per monitoraggio, detection, mitigazione e API REST. Il monitor (`monitor.py`) raccoglie periodicamente statistiche granulari dagli switch, fornendo dati aggiornati per la detection. Il detector (`detector.py`) analizza le statistiche tramite plugin, utilizzando soglie adattive per rilevare anomalie e notificando il mitigator. Il mitigator (`mitigator.py`) applica regole di blocco e sblocco granulari, gestendo lo sblocco progressivo e il logging delle decisioni. L'API (`api.py`) espone endpoint REST per la gestione dinamica delle regole.

Le scelte architetturali sono state guidate dalla necessità di garantire modularità, estendibilità e reattività. L'uso di thread separati assicura monitoraggio e risposta continua, mentre le soglie adattive e il blocco granulare riducono i falsi positivi e l'impatto sul traffico legittimo. La documentazione delle decisioni e la presenza di logging facilitano audit, debugging e future estensioni.

La soluzione è stata progettata per essere facilmente estendibile: nuovi plugin di detection, criteri di mitigazione e endpoint API possono essere aggiunti senza modificare la struttura esistente. Il sistema è pronto per ulteriori sviluppi e integrazioni, mantenendo la flessibilità necessaria per adattarsi a scenari e minacce emergenti.