UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Informatica

AI-Driven Security Automation: Generazione Dinamica di Regole Firewall con Large Language Model e RAG

Corso di Penetration Testing and Ethical Hacking

Studente:
Marco Santoriello

ANNO ACCADEMICO 2025/2026

Contents

1 Introduzione

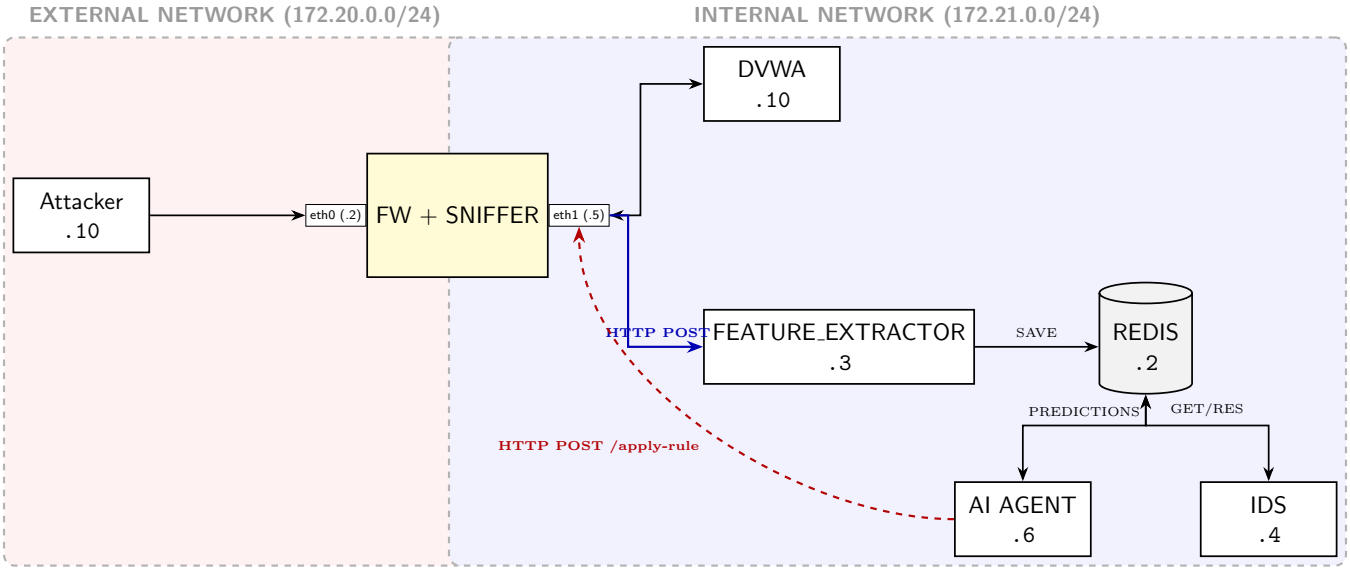
I sistemi di Intrusion Detection (IDS) rappresentano una componente fondamentale delle architetture di sicurezza moderne, consentendo il rilevamento automatico di attività malevole attraverso l'analisi del traffico di rete. Gli approcci tradizionali si dividono principalmente in due categorie: *signature-based*, che identificano pattern noti di attacco, e *anomaly-based*, che rilevano deviazioni dal comportamento normale mediante tecniche di Machine Learning.
Tuttavia, il rilevamento della minaccia costituisce solo il primo stadio di una strategia difensiva efficace. La capacità di rispondere in tempo reale agli attacchi identificati, generando automaticamente policy di sicurezza contestuali e applicandole dinamicamente ai sistemi di enforcement (firewall, IPS), rappresenta un'evoluzione critica verso architetture di sicurezza adattive.
L'avvento dei Large Language Model (LLM) apre nuove possibilità in questo dominio.
Questo progetto, sviluppato per l'esame di *Penetration Testing and Ethical Hacking*, si propone di investigare l'utilizzo di un LLM, integrato con tecniche di Retrieval Augmented Generation (RAG), per generare automaticamente regole di enforcement per un firewall, in risposta ad attacchi di rete rilevati da un sistema di Intrusion Detection. Verrà analizzato il livello di accuratezza che è possibile raggiungere e i limiti di questo approccio.
Il progetto si articola nei seguenti obiettivi specifici:

- Progettare ed addestrare un modello di ML per l'analisi del traffico di rete, capace di identificare attività malevole o attacchi.
- Costruire un framework che comprenda un agente AI in grado di interpretare l'output dell'IDS e generare regole di sicurezza in modo automatico e dinamico.
- Implementare un meccanismo RAG che fornisca all'agente esempi validati di regole per specifiche tipologie di attacco, con l'obiettivo di mitigare le allucinazioni e fornire maggiore contesto all'agente.
- Sviluppare un prototipo di architettura funzionale in ambiente containerizzato che integri tutti i componenti del sistema.
- Implementare un framework di valutazione per misurare le performance dell'agente
- Identificare i limiti di questo approccio ed eventuali sviluppi futuri

Architettura. Di seguito è riportata una descrizione sintetica dell'architettura realizzata

- Container Metasploit che simula l'attaccante, situato nella rete esterna
- Firewall (con interfaccia `eth0` connessa alla rete esterna, interfaccia `eth1` connessa alla rete interna)
- Sniffer che cattura il traffico diretto alla rete interna sull'interfaccia `eth0` del firewall
- Modulo di feature extraction che analizza ed estrae le features dal traffico, utilizzando NTLFlowLyzer [7]
- Container Redis
- Modulo IDS anomaly-based.

- AI Agent che genera regole di enforcement.
- Container DVWA (Damn Vulnerable Web Application): asset vulnerabile



2 Come replicare

Requisiti. Il sistema richiede i seguenti componenti:

- Docker Engine (versione utilizzata v29.1.5)
- Docker Compose (versione utilizzata: v5.0.1)
- Ollama con modello Llama 3 installato sull'host
- Git

Clone del Repository. Il codice sorgente completo è disponibile su GitHub:

```
git clone https://github.com/marcosantoriello/PTEH-Smart-Security-Agent.git
cd PTEH-Smart-Security-Agent
```

Setup di Ollama Prima di avviare l'infrastruttura, è necessario installare Ollama (`https://ollama.com/`), scaricare il modello `Llama 3` e avviare il server:

```
ollama pull llama3

ollama serve
```

Setup NTLFlowLyzer Innanzitutto, bisogna posizionarsi sul tag v0.1.0:

```
# dalla root della repo
git submodule update --init --recursive

cd architecture/feature-extractor/libs/NTLFlowLyzer
git fetch --tags
git checkout v0.1.0
```

Prima del deployment, è necessario correggere un typo nella libreria NTLFlowLyzer. Modificare il file:

```
architecture/feature-extractor/libs/NTLFlowLyzer/NTLFlowLyzer/__init__.py
```

Sostituire la linea 5 con:

```
from .network_flow_analyzer import NTLFlowLyzer
```

Per farlo automaticamente, eseguire:

```
sed -i 's/NLFlowLyzer/NTLFlowLyzer/g' NTLFlowLyzer/__init__.py

cd ../../../../
```

Deployment Una volta completati i passaggi precedenti, avviare l'infrastruttura:

```
docker compose up --build
```

Esecuzione degli Attacchi Per accedere al container dell'attaccante ed eseguire attacchi simulati:

```
docker exec -it attacker bash

# da qui vi sono varie opzioni, tra cui:
# avvio msfconsole
./msfconsole

# port scan con nmap
nmap -T4 172.21.0.10

# Denial of Service
python3 dos.py 172.21.0.10
```

Shutdown Per arrestare l'infrastruttura:

```
docker compose down

# Per rimuovere anche i volumi
docker compose down -v
```

Nota: al primo avvio, il container dell'agente impiegherà

3 IDS

Di seguito viene dettagliato il processo di sviluppo, addestramento e validazione del modulo IDS (Intrusion Detection System). L'obiettivo primario di questa fase è stato quello di superare i limiti di un semplice classificatore binario (realizzato in una prima versione del progetto, visualizzabile nel file */notebook/ids_v1.ipynb*), implementando un modello multiclasse capace di distinguere con precisione 14 diverse tipologie di attacco, oltre al traffico benigno. Questa granularità è fondamentale per permettere all'Agente Intelligente di generare regole di firewall "context-aware" e specifiche per la minaccia rilevata.

3.1 Dataset Il dataset scelto per questo progetto è il *BCCC-CIC-IDS2017*[1]. Sebbene esista una versione più aggiornata, generata con una versione più recente di NTLFlowLyzer, la decisione è ricaduta su questo dataset per ovvie limitazioni di hardware. Infatti, il *BCCC-CIC-IDS2018* è una versione estremamente più grande e pesante, che avrebbe reso l'addestramento impraticabile, considerata la configurazione hardware a disposizione.

3.2 Data Analyisis Il dataset è composto da vari file, ciascuno contenente pacchetti appartenenti ad uno specifico attacco:

- *botnet_ares.csv*
- *ddos_loit.csv*
- *dos_hulkdos_golden_eye.csv*
- *dos_hulk.csv*
- *dos_slowhttptest.csv*
- *dos_slowloris.csv*
- *friday_benign.csv*
- *ftp_patator.csv*
- *heartbleed.csv*
- *monday_benign.csv*
- *portscan.csv*
- *ssh_patator-new.csv*
- *thursday_benign.csv*
- *tuesday_benign.csv*
- *web_brute_force.csv*
- *web_sql_injection.csv*
- *web_xss.csv*

[1] Url: `https://www.unb.ca/cic/datasets/ids-2017.html`

- *wednesday_benign.csv*

Per comodità, sono stati uniti tutti i file per creare un unico dataset.
Dopo aver caricato il dataset, è stata effettuata un'analisi esplorativa dei dati (EDA) per comprendere la struttura e le caratteristiche del dataset.
La variabile target è rappresentata dalla colonna *Label*, che indica se il traffico è normale, oppure appartiene ad una specifica tipologia di attacco. I valori univoci presenti nella colonna *Label* sono i seguenti:

- *Benign*
- *Port_Scan*
- *SSH-Patator*
- *Web_SQL_Injection*
- *Web_Brute_Force*
- *Web_XSS*
- *Botnet_ARES*
- *DoS_GoldenEye*
- *Heartbleed*
- *DDoS_LOIT*
- *FTP-Patator*
- *DoS_Slowhttptest*
- *DoS_Hulk*
- *DoS_Slowloris*

3.2.1 Analisi della Distribuzione delle Classi L'analisi esplorativa dei dati (EDA) ha evidenziato un forte sbilanciamento delle classi. Come mostrato nella Figura 1, la classe *Benign* domina il dataset, seguita da *DoS Hulk* e *PortScan*. Alcuni attacchi, come *Heartbleed* o *Web SQL Injection*, sono rappresentati da un numero estremamente esiguo di campioni (nell'ordine delle decine), il che pone una sfida significativa per l'algoritmo di apprendimento.

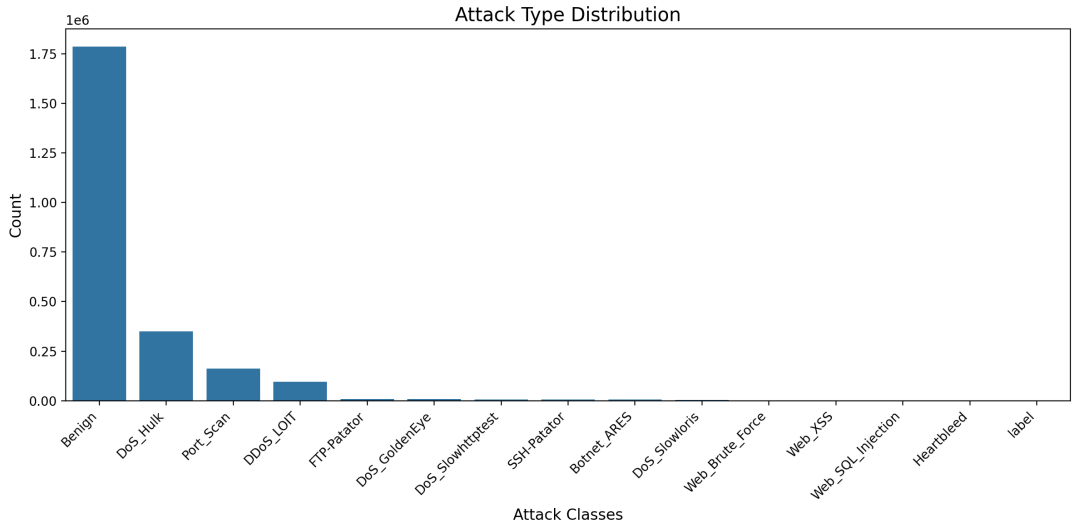


Figure 1: Distribuzione delle tipologie di attacco nel dataset CIC-IDS2017.

Al fine di visualizzare meglio le proporzioni, la Figura 2 mostra la ripartizione percentuale delle classi. Nonostante lo sbilanciamento, si è deciso di utilizzare uno *Stratified Split* per il training, garantendo che la proporzione delle classi fosse mantenuta sia nel training set che nel test set.

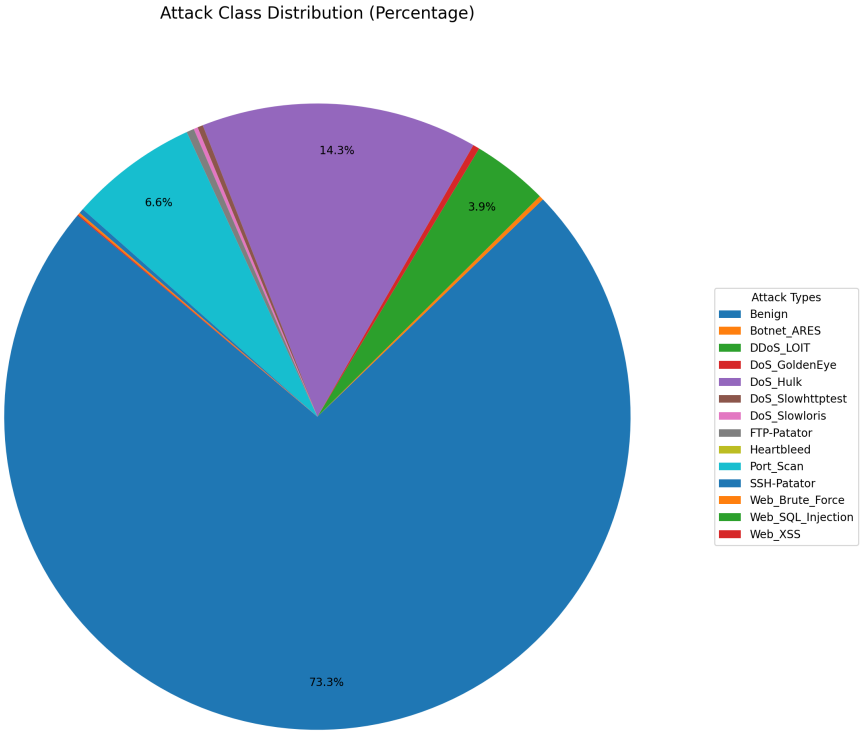


Figure 2: Ripartizione percentuale delle classi di traffico.

3.3 Data Cleaning In una fase preliminare di pulizia, il dataset è stato sottoposto a deduplicazione, alla sostituzione dei valori indefiniti o infiniti con *NaN* e alla successiva rimozione degli stessi. Contestualmente, la variabile target è stata separata dal resto delle variabili predittive, così da consentire un flusso di preprocessing metodologicamente corretto e conforme alle buone pratiche di apprendimento supervisionato.
Infine, sono state rimosse alcune features non rilevanti per il processo di apprendimento, riportate di seguito:

- *flow_id*
- *src_ip*
- *dst_ip*
- *src_port*
- *timestamp*

Di seguito è riportato un confronto tra il numero di record presenti nel dataset prima e dopo la fase di data cleaning.

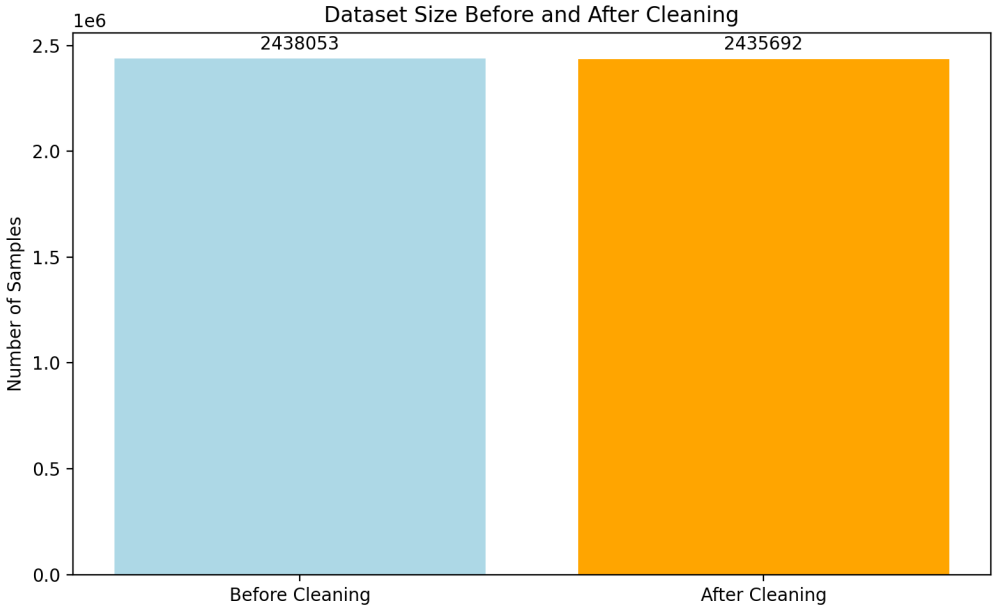


Figure 3: Confronto tra il numero di record prima e dopo la fase di data cleaning

3.4 Feature Engineering e Selection Il dataset originale comprende oltre 80 feature estratte dai flussi di rete. Per migliorare l'efficienza computazionale del modello e ridurne il rischio di overfitting, è stata applicata una rigorosa pipeline di selezione delle feature.

3.4.1 Gestione della Multicollinearità Una delle problematiche principali nei dati di rete è l'alta correlazione tra feature derivate (es. *Flow Duration* vs *Fwd IAT Total*). Tramite l'analisi della matrice di correlazione (Figura 4), sono state identificate e rimosse le feature con un coefficiente di correlazione di Pearson superiore a 0.95. Questo passaggio ha permesso di eliminare informazioni ridondanti.

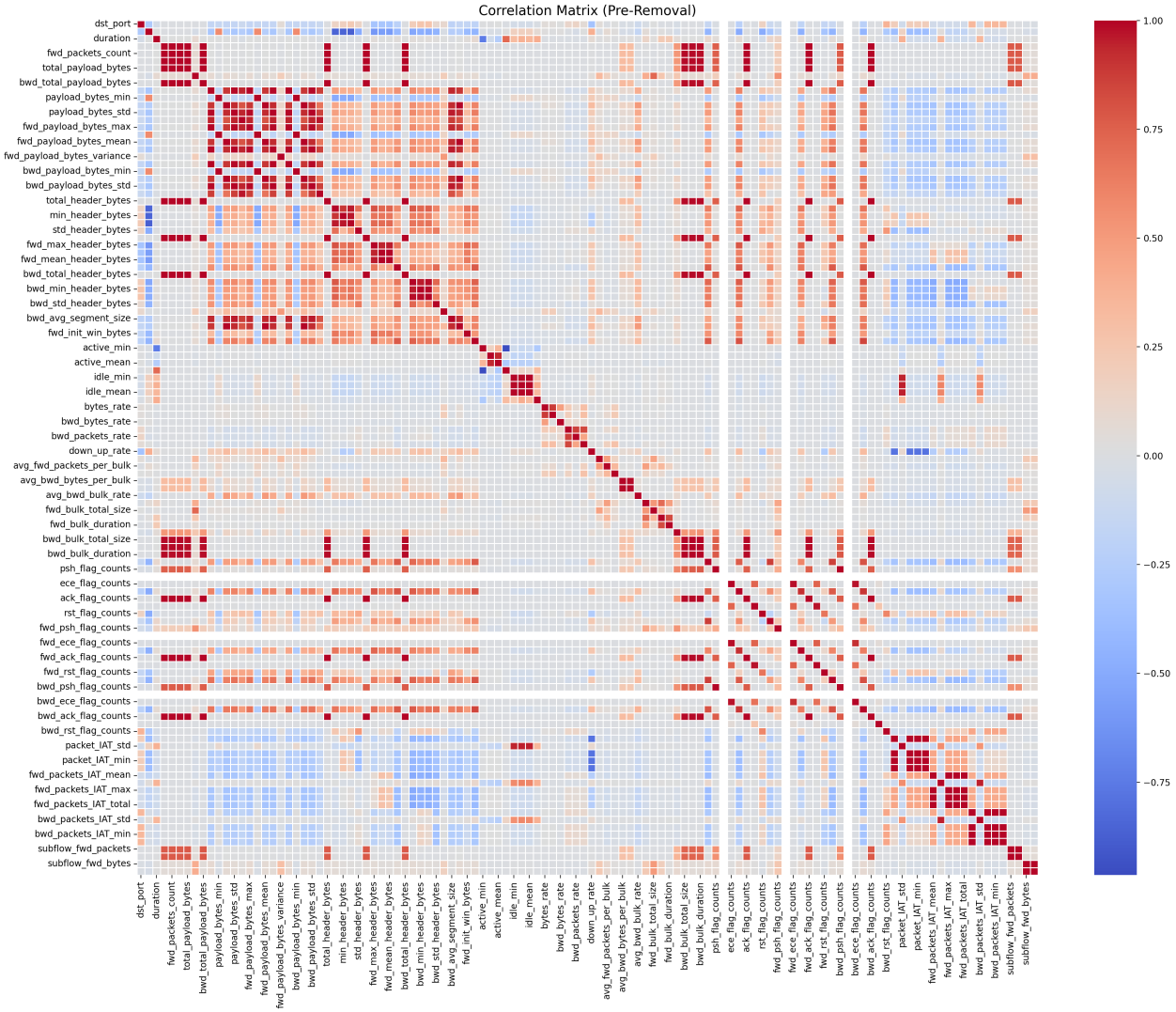


Figure 4: Matrice di correlazione delle feature prima della rimozione delle ridondanze.

3.4.2 Selezione tramite Random Forest Importance Dopo la rimozione delle feature a bassa varianza e di quelle correlate, è stato addestrato un Random Forest preliminare per calcolare la *Feature Importance* (Gini Importance). Sono state selezionate le top 40 feature più discriminanti. Questa selezione ha permesso di scartare attributi poco significativi, riducendo la dimensione del vettore di input senza compromettere l'accuratezza. Successivamente, i dati sono stati normalizzati utilizzando uno *StandardScaler*, portando tutte le feature a media 0 e varianza 1.

3.5 Addestramento e Ottimizzazione del Modello

L'algoritmo scelto per la classificazione è il **Random Forest Classifier**. Questo modello ensemble è stato preferito per la sua intrinseca capacità di gestire dati non lineari e per la sua robustezza agli outlier.
Per individuare la configurazione ottimale, è stata eseguita una ricerca degli iperparametri tramite **RandomizedSearchCV** con *3-fold cross-validation*. Per limitare i tempi di calcolo, l'ottimizzazione è stata effettuata su un sottoinsieme stratificato del dataset (50.000 campioni).
I migliori parametri identificati e successivamente utilizzati per il training sul dataset completo sono:

- **n_estimators**: 100
- **max_depth**: 20

- **min_samples_split**: 2
- **min_samples_leaf**: 1
- **max_features**: 'log2'

3.6 Valutazione delle Performance Il modello finale è stato valutato sul test set (20% del dataset totale, circa 480.000 campioni). I risultati sono stati eccellenti, con un'**accuratezza globale del 99.86%** e un **F1-Score (weighted) del 99.85%**.
La matrice di confusione (Figura 5) evidenzia come il modello riesca a classificare correttamente la quasi totalità del traffico benigno e degli attacchi volumetrici (DoS).

Figure 5: Matrice di Confusione del modello Random Forest ottimizzato sul Test Set.

È importante notare che, per classi estremamente minoritarie come *Heartbleed* o *Web SQL Injection* (che contano pochissimi campioni nel test set), il modello potrebbe mostrare performance inferiori (recall più bassa), un limite intrinseco dovuto alla scarsità di esempi di training. Tuttavia, per gli attacchi più comuni e critici in ambienti enterprise (Botnet, DDoS, PortScan), il rilevamento risulta affidabile.

3.7 Esportazione e Produzione Viene, infine, eseguito il processo di training, al termine del quale il sistema esporta automaticamente tutti gli artefatti necessari per l'inferenza in produzione. Utilizzando la libreria `joblib`, viene creato un singolo pacchetto (`ids_model.joblib`) che incapsula:

1. Il modello *Random Forest* addestrato.
2. L'oggetto *StandardScaler* fittato (per normalizzare i nuovi dati con gli stessi parametri).
3. Il *LabelEncoder* per decodificare le predizioni numeriche in etichette testuali
4. L'encoder per il protocollo.
5. La lista delle 40 feature selezionate, per garantire che l'input in fase di predizione abbia la stessa struttura del training.

4 Network Architecture

Nel seguente capitolo, verranno illustrate e dettagliate singolarmente le varie componenti dell'architettura di rete.

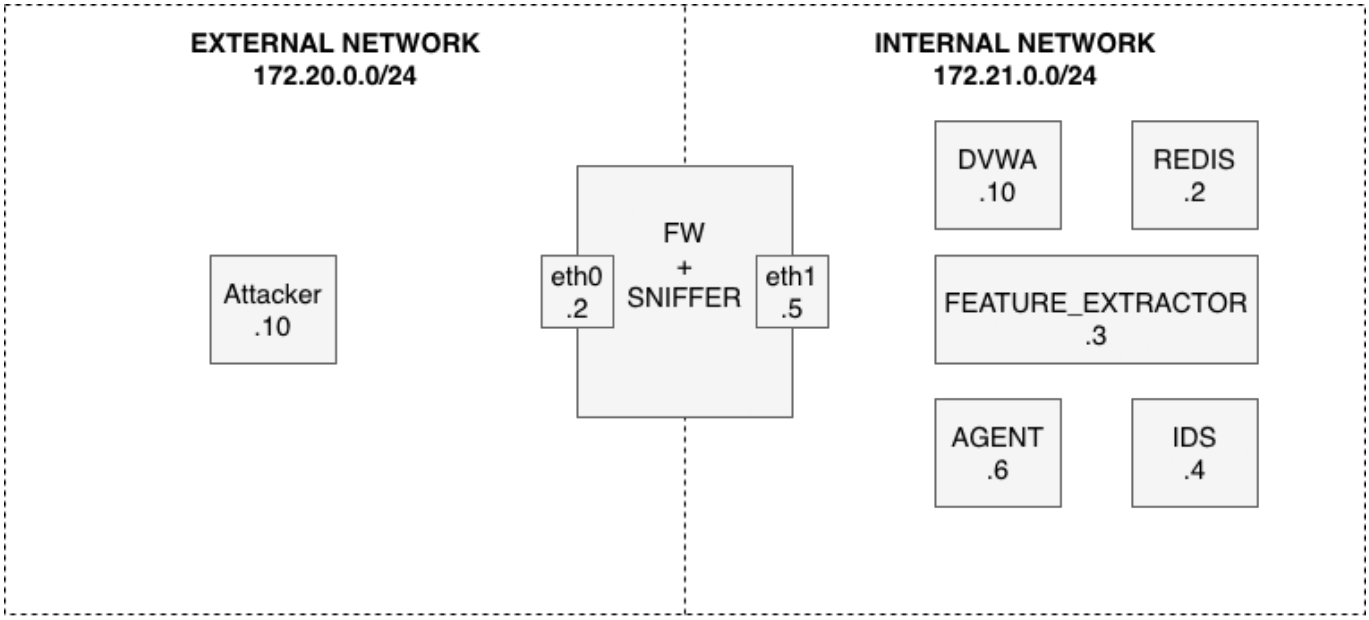


Figure 6: Architettura di rete

4.1 Traffic Sniffer. Il modulo **Traffic Sniffer** è responsabile della cattura dei pacchetti di rete in transito sull'interfaccia di rete configurata. Tale modulo è incorporato all'interno del modulo firewall e cattura pacchetti sull'interfaccia *eth0*, ovvero l'interfaccia connessa alla rete esterna. Pertanto, l'esecuzione del modulo è avviata dal firewall. Tuttavia, in questo documento, verrà considerato logicamente come un'entità separata. Per la sua implementazione è stata utilizzata la libreria Python *Scapy*. In particolare, tale modulo cattura i pacchetti e li inserisce in un buffer temporaneo finché la dimensione di tale buffer non raggiunge una soglia specificata (*batch size*): al raggiungimento di tale soglia, i pacchetti presenti nel buffer vengono salvati in un file *.pcap* e il buffer viene svuotato. L'introduzione di tale soglia consente di ottimizzare le operazioni di *I/O*. I file generati dallo sniffer vengono salvati, per garantire unicità e chiarezza, utilizzando la seguente convenzione di naming: `capture_YYYYMMDD_HHMMSS.pcap`.
Ogni volta che viene generato un nuovo file PCAP, il modulo effettua una chiamata HTTP POST all'API del modulo di Feature Extraction per notificare che è disponibile un nuovo file da analizzare. In tale richiesta non viene passato direttamente il file PCAP, ma viene inviato il path assoluto del file. La motivazione alla base di questa scelta è che consente di evitare eventuali sovraccarichi di memoria e di rete, migliorando l'efficienza rispetto all'invio diretto del file. Inoltre, considerando le finalità del progetto, garantisce un approccio meno complesso, sfruttando un volume condiviso Docker. Infine, il modulo implementa un robusto sistema di logging, tracciando tutte le fasi operative ed eventuali errori/anomalie.

4.2 Feature Extractor. Il modulo di *feature extraction* ha il compito di processare i file contenenti il traffico catturato, generati dal Traffic Sniffer. Tale modulo espone un endpoint REST (`/new_pcap`), a cui è possibile inviare, come appena discusso, una richiesta HTTP POST contenente il path assoluto al file PCAP contenente il traffico catturato. Per l'effettiva estrazione del traffico, è

stato utilizzato il tool **NTLFlowLyzer**[7], eseguito su un thread separato per non bloccare, eventualmente, richieste REST in arrivo. Inoltre, il file JSON di configurazione `config.json`, viene caricato in memoria e modificato dinamicamente per aggiornare il path del file PCAP di input ed eventualmente quello di output, generando un file temporaneo utilizzato da NTLFlowLyzer.
Una volta terminata l'estrazione delle features dal file di input, il Feature Extractor salva il risultato in un file CSV il cui nome rispetta la convenzione `"features:capture_YYYYMMDD_HHMMSS.csv"`. Il contenuto del file viene quindi salvato in Redis utilizzando come chiave il pattern `features:capture_YYYYMMDD_HHMMSS.csv`. Contestualmente, viene estrato il timestamp dal nome del file, convertito in formato Unix timestamp e viene eseguita la seguente operazione di indicizzazione: la chiave Redis viene aggiunta a un Sorted Set, denominato `features_index`, utilizzando il timestamp Unix come score (indice).
Quest'ultima operazione consente al modulo IDS di interrogare Redis in modo efficiente per recuperare solo i file processati dopo un determinato timestamp, come descritto nella sezione successiva di questo documento.
Anche questo modulo è dotato di un meccanismo di logging robusto.

4.3 Interazione tra Feature Extractor e Traffic Sniffer.

Di seguito è illustrata più in dettaglio l'interazione tra i due moduli:

1. Traffic Sniffer: cattura pacchetti in arrivo e li salva in batch in file PCAP nella directory condivisa `/shared/pcap`.

2. Traffic Sniffer: dopo ogni salvataggio, invia una richiesta HTTP POST al Feature Extractor con un payload JSON contenente il path e il nome del PCAP appena salvato.

3. Feature Extractor: riceve la notifica e avvia in background, su un thread separato, la procedura di estrazione delle features sul suddetto PCAP.

4. Feature Extractor: al termine dell'extraction, in assenza di errori, salva il file ottenuto in Redis.

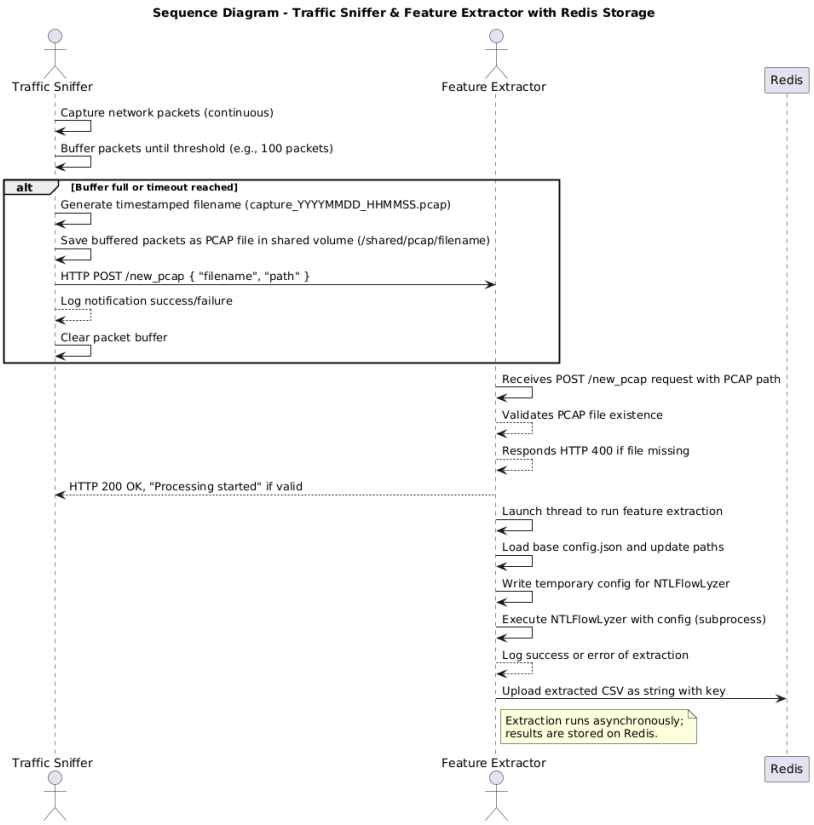


Figure 7: Sequence Diagram che mostra in dettaglio il flusso appena descritto

4.4 IDS Module

Il modulo **IDS** rappresenta il nucleo del sistema di rilevamento delle anomalie. Questo componente è responsabile della classificazione del traffico di rete in tempo reale, distinguendo tra flussi benigni e malevoli attraverso un modello di Machine Learning pre-addestrato. A differenza dei moduli precedenti che operano in modalità event-driven (reagendo a richieste HTTP), l'IDS adotta un approccio *polling-based*: interroga periodicamente Redis alla ricerca di nuovi dati da analizzare, senza richiedere notifiche esterne.[2]

4.4.1 Inizializzazione e Setup

Durante la fase di inizializzazione, il modulo IDS esegue una serie di operazioni preliminari, fondamentali per il suo corretto funzionamento. Come illustrato nel Sequence Diagram 8, il processo di setup si articola in tre fasi principali:

1. **Connessione a Redis e recupero dello stato**: il modulo si connette all'istanza Redis e interroga il Sorted Set `features_index` tramite il comando `ZREVRANGE`, che restituisce gli elementi ordinati per score (timestamp) in ordine decrescente. Questo consente di recuperare il timestamp dell'ultimo file processato (*last_processed_timestamp*). Se presente, tale valore viene salvato in memoria per consentire al modulo di riprendere l'elaborazione dal punto in cui si era interrotta, evitando così il riprocessamento di dati già analizzati. In caso contrario, il modulo assume di trovarsi alla prima esecuzione e processerà tutti i file disponibili.

2. **Caricamento del modello pre-addestrato**: viene caricato dal disco il *model package* (file `.joblib` serializzato con *joblib*), contenente quattro componenti essenziali:

[2]Con il termine *polling* si intende un meccanismo in cui un processo verifica attivamente, a intervalli regolari, la disponibilità di nuove risorse o dati, piuttosto che attendere passivamente notifiche asincrone.

- `model`: il classificatore Random Forest addestrato sul dataset CIC-IDS2017.
- `scaler`: lo scaler MinMaxScaler utilizzato per normalizzare le features nel range [0,1], fittato sui dati di training.
- `feature_names`: lista ordinata delle 50 features selezionate durante il training, necessaria per garantire la corrispondenza tra le colonne del DataFrame in input e quelle attese dal modello.
- `label_encoder`: encoder utilizzato per decodificare le predizioni numeriche in etichette testuali.
- `protocol_encoder`: encoder specifico per convertire i valori categorici della colonna `protocol` in valori numerici.

3. **Avvio del loop di polling**: al termine del setup, il modulo entra nel ciclo principale di elaborazione, che viene eseguito indefinitamente con intervalli di attesa configurabili tramite la variabile d'ambiente `POLLING_INTERVAL` (valore di default: 10 secondi).

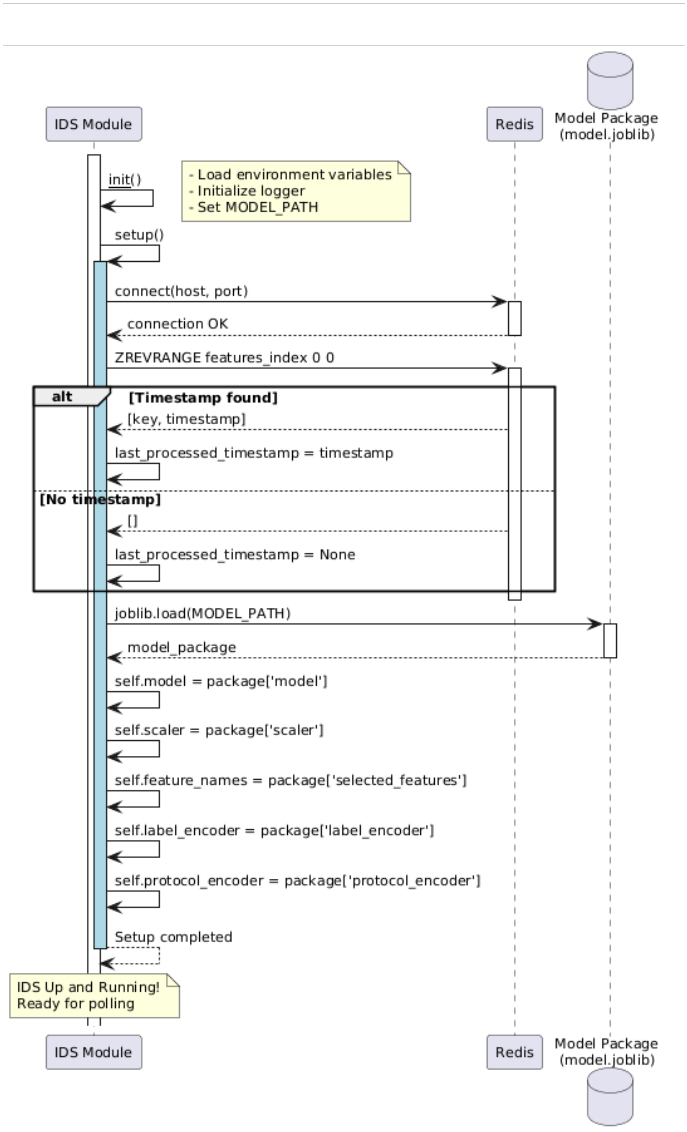


Figure 8: Sequence Diagram: fase di setup del modulo IDS

4.4.2 Loop Principale: Polling, Preprocessing e Predizione Il cuore del modulo IDS è rappresentato dal loop principale, illustrato nel Sequence Diagram 9, che orchestra l'intero flusso di elaborazione. Tale ciclo si ripete indefinitamente e si compone delle seguenti fasi:

1. **Recupero delle chiavi dei nuovi file**: il modulo interroga Redis utilizzando il comando `ZRANGEBYSCORE` sul Sorted Set `features_index`. Tale comando restituisce tutte le chiavi il cui score (timestamp Unix) è strettamente maggiore di `last_processed_timestamp`. Questo meccanismo di indicizzazione temporale, introdotto dal Feature Extractor, consente di recuperare in modo efficiente solo i file generati successivamente all'ultima elaborazione, evitando il riprocessamento di dati già analizzati. Se non vengono trovati nuovi file, il modulo attende per `POLLING_INTERVAL` secondi prima di riprovare.

2. **Caricamento e aggregazione dei dati**: per ogni chiave recuperata (es. `features:capture_20260114_143052.csv`), il modulo esegue un comando `GET` per ottenere il contenuto CSV memorizzato in Redis. I dati vengono deserializzati in DataFrame Pandas e, nel caso di più file, vengono concatenati in un unico DataFrame aggregato. Questa fase garantisce la gestione efficiente di batch multipli di traffico.

3. **Preprocessing dei dati**: il DataFrame aggregato viene sottoposto a una serie di trasformazioni, necessarie per renderlo compatibile con il modello:

   - *Gestione dei valori infiniti*: eventuali valori $\pm\infty$ vengono sostituiti con `NaN` e successivamente rimossi o imputati.
   - *Encoding della colonna protocol*: i valori categorici della colonna `protocol` (es. "TCP", "UDP", "ICMP") vengono convertiti in valori numerici utilizzando il `protocol_encoder` caricato dal model package. Nel caso di protocolli sconosciuti (non presenti nel training set), viene assegnato il valore di default -1.
   - *Selezione delle features*: vengono estratte esclusivamente le 50 features utilizzate durante il training, garantendo la corrispondenza esatta con l'ordine memorizzato in `feature_names`.
   - *Scaling*: le features selezionate vengono normalizzate tramite il metodo `transform()` dello `scaler`, che applica la stessa trasformazione MinMax utilizzata in fase di training.

   Il risultato di questa fase è una matrice `X_scaled` contenente le features preprocessate, pronta per essere fornita in input al modello.

4. **Predizione**: la matrice `X_scaled` viene passata al classificatore Random Forest, che restituisce:

   - `predictions`: array di valori binari (0 = benign, 1 = attack) che indicano la classe predetta per ogni flusso di rete.
   - `prediction_proba`: matrice contenente le probabilità di appartenenza a ciascuna classe. Da questa viene estratta la confidence (probabilità massima) per ogni predizione.

   Contestualmente alle predizioni, vengono estratti dal DataFrame originale i metadati necessari per l'applicazione delle regole di sicurezza: `src_ip`, `src_port`, `dst_ip`, `dst_port`, `protocol`. Questi campi, insieme alle predizioni e alle confidence, vengono combinati in un DataFrame strutturato che verrà salvato in Redis per l'utilizzo da parte dell'AI Agent.

5. **Salvataggio delle predizioni**: il DataFrame contenente le predizioni e i metadati viene salvato in Redis in due modalità distinte:

- *Tutte le predizioni*: vengono serializzate in formato JSON e salvate con chiave `predictions:{timestamp}`, dove {`timestamp`} rappresenta il timestamp Unix corrente. La chiave viene aggiunta al Sorted Set `predictions_index` tramite il comando `ZADD`, utilizzando il timestamp come score per consentire interrogazioni temporali efficienti.
- *Solo gli attacchi rilevati*: i flussi classificati come malevoli (prediction = 1) vengono filtrati e salvati separatamente con chiave `attacks:{timestamp}`, indicizzati nel Sorted Set `attacks_index`. Questa separazione consente all'AI Agent di recuperare rapidamente solo i flussi che richiedono un'azione di mitigazione immediata, senza dover analizzare l'intero dataset di predizioni.

6. **Aggiornamento del timestamp di elaborazione**: al termine del processamento, il modulo aggiorna il valore di `last_processed_timestamp` per garantire che nella prossima iterazione del polling vengano recuperati solo i file generati successivamente. Questo viene fatto interrogando Redis con il comando `ZMSCORE`, che restituisce gli score (timestamp) associati alle chiavi processate. Il massimo tra questi valori viene salvato come nuovo `last_processed_timestamp`. Questo meccanismo è fondamentale per garantire l'idempotenza del sistema: anche in caso di riavvio del modulo, l'elaborazione riprende esattamente dal punto in cui si era interrotta, senza perdita o duplicazione di dati.

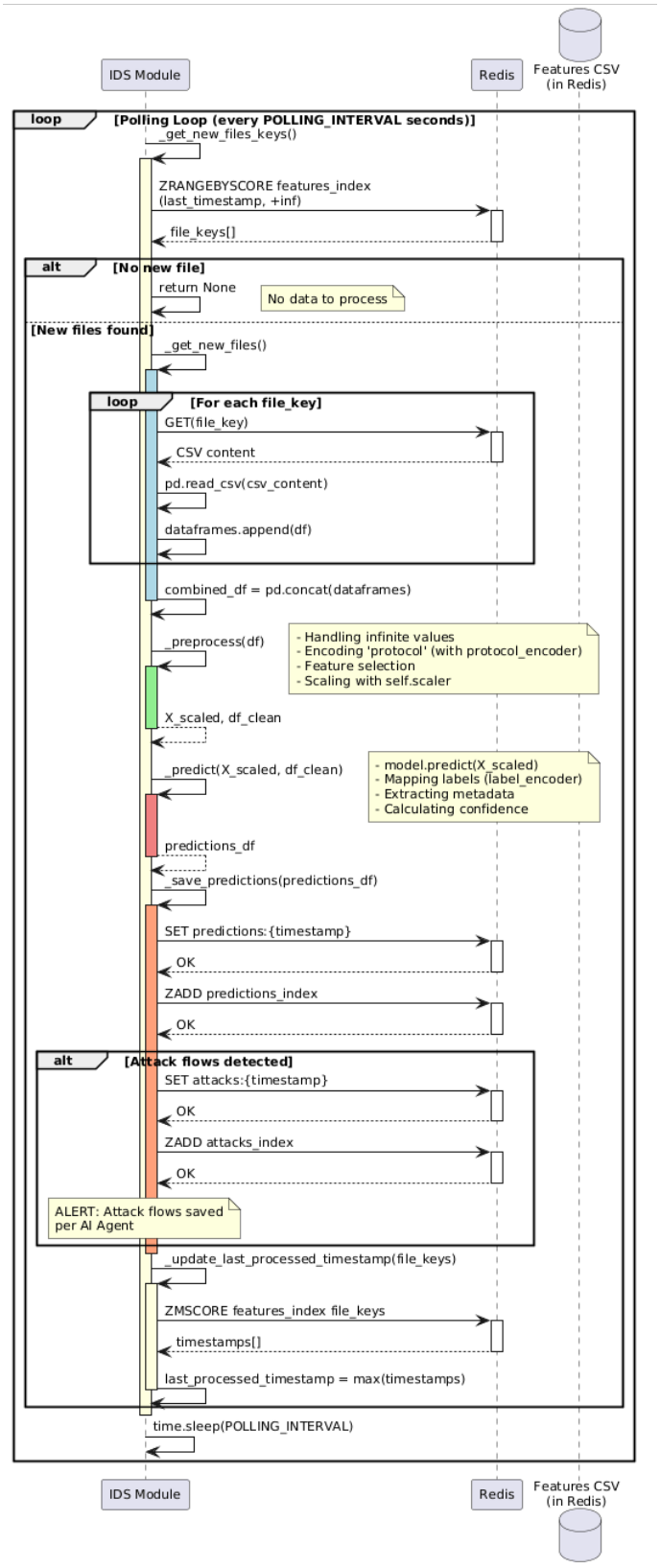


Figure 9: Sequence Diagram: loop principale del modulo IDS, che include polling, preprocessing, predizione e salvataggio

4.4.3 Struttura dei Dati Salvati in Redis I dati salvati dal modulo IDS seguono una struttura JSON rigorosa, progettata per facilitarne l'utilizzo da parte dell'AI Agent. Ogni predizione è rappresentata da un oggetto JSON contenente i seguenti campi:

- `src_ip`, `src_port`: indirizzo IP sorgente e porta sorgente del flusso di rete.
- `dst_ip`, `dst_port`: indirizzo IP di destinazione e porta di destinazione.
- `protocol`: protocollo utilizzato (TCP, UDP, ICMP, etc.).
- `prediction`: label testuale della classe predetta (es. "Benign", "DoS Hulk").
- `prediction_id`: identificativo numerico della classe predetta (es. 0, 4).
- `confidence`: livello di confidenza della predizione, espresso come valore reale nell'intervallo [0, 1].
- `timestamp`: timestamp ISO 8601 del momento in cui è stata effettuata la predizione.

Questa struttura fornisce all'AI Agent tutte le informazioni necessarie per generare regole firewall specifiche e contestualizzate, consentendo un'azione di mitigazione mirata (es. blocco di uno specifico IP sorgente su una specifica porta).

4.4.4 Gestione degli Errori e Logging Il modulo implementa un sistema di logging dettagliato, che traccia ogni fase del processo di elaborazione.
In caso di errore durante l'elaborazione di un singolo file, il modulo non interrompe l'esecuzione, ma registra l'errore e prosegue con i file successivi, garantendo la robustezza del sistema anche in presenza di dati corrotti o incompleti. Inoltre, il meccanismo di aggiornamento del timestamp è atomico: il `last_processed_timestamp` viene aggiornato solo dopo il completamento con successo di tutte le fasi di elaborazione, evitando inconsistenze in caso di crash.

4.5 Firewall.

Il modulo firewall costituisce il principale componente di enforcement delle policy di sicurezza fungendo, inolte, da gateway tra la rete esterna e la rete interna. Il firewall è configurato come un container Docker *multi-homed*, ovvero connesso simultaneamente a due reti distinte:

- **external_net** (172.20.0.0/24): rappresenta la rete esterna in cui si trova l'attaccante. Il firewall è connesso a questa rete tramite l'interfaccia *eth0*, con indirizzo IP `172.20.0.2`.
- **internal_net** (172.21.0.0/24): rappresenta la rete interna, contenente il servizio da proteggere (il target dell'attaccante) e i moduli di back-end di cui sopra. Il firewall è connesso a questa rete tramite l'interfaccia *eth1*, con indirizzo IP `172.21.0.5`.

Tale segmentazione logica, implementata tramite reti **Docker bridge**, garantisce isolamento tra le due zone, assicurando che tutto il traffico tra le due reti passi attraverso il firewall.

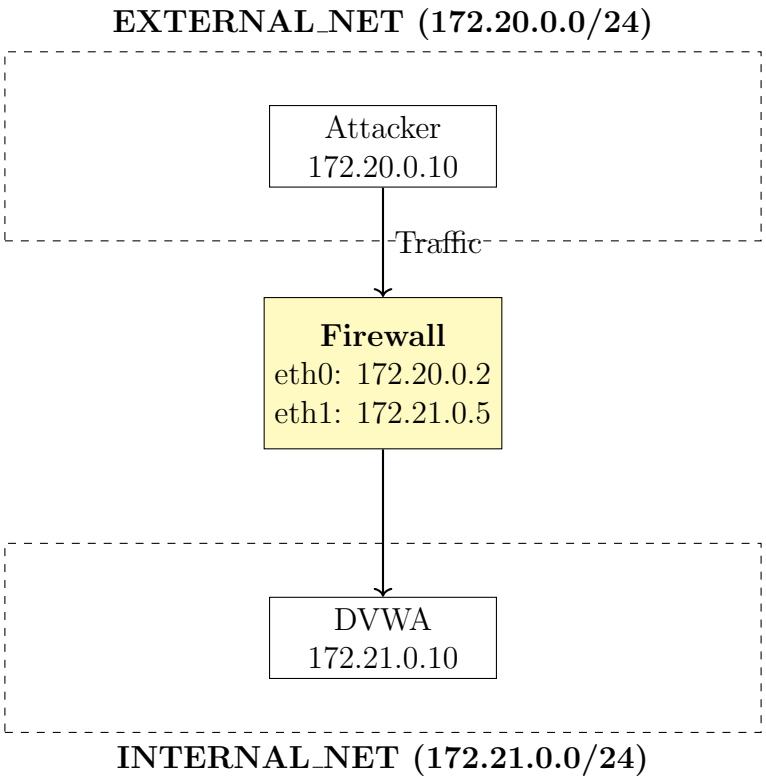


Figure 10: Topologia di rete semplificata con firewall multi-homed

Nell'architettura implementata, il modulo di traffic sniffing è integrato nel container del firewall. Come già brevemente anticipato, il firewall è l'unico punto di transito obbligatorio tra le due reti. Posizionando lo sniffer al suo interno, viene garantita la cattura di tutto il traffico in ingresso, prima che venga processato dal firewall effettivo. Infatti, Scapy, utilizzando raw socket, intercetta i pacchetti a livello kernel, prima del layer di filtraggio, assicurando visibilità completa anche di traffico che potrebbe essere successivamente bloccato. Il modulo *Traffic Sniffer*, implementato in una versione iniziale del progetto, prima dell'aggiunta del firewall nell'archiettura, viene montato come volume.

4.5.1 Firewall Enforcer All'interno del firewall, è presente un submodule, chiamato Firewall Enforcer, che funge da interfaccia sicura tra l'agente intelligente e le effettive regole del firewall del sistema. Tale livello implementa un livello di validazione critico, che garantisce integrità e sicurezza dell'infrastruttura. Tale sottomodulo, espone una REST API, implementata utilizzando Flask, che riceve richieste contenenti regole da applicare, le valida e, in caso di validazione positiva, le applica al sistema mediante *iptables*.
Il processo di validazione è stato implementato prevedendo tre fasi distinte:

1. **Pattern Matching:** in questa fase, viene utilizzata un'espressione regolare per verificare che la regola rispetti la sintassi base di iptables. Infatti, viene controllato che il comando inizi con `iptables`, seguito da uno dei flag validi, ovvero `-A` (append), `-I` (insert), `-D` (delete), `-L` (list) e da una chain valida. Input malformati, pertanto, verranno scartati.

2. **Sanitizzazione dell'input:** l'input dell'agente intelligente, considerato un'entità non fidata e potenziale target di attacchi, viene sanitizzato tramite una blacklist. Input che contengono caratteri pericolosi (e.g. `&&`, `$()`, ecc.) vengono rigettati. Infatti, oltre ad essere un potenziale

target di attacchi, vi è il rischio che un LLM generi output non deterministici, oppure che abbia allucinazioni, generando comandi pericolosi.

3. **Validazione nativa del kernel:** la validazione finale utilizza il comando *iptables-restore* con il flag `--test` per validare la sintassi della regola attraverso il parser nativo del kernel Linux. Tale approccio garantisce che la regola sia sintatticamente corretta, secondo le specifiche esatte di *iptables*, includendo la validazione di tutti i moduli e delle loro opzioni.[6] Fondamentalmente, dunque, tramite il flag `--test`, il comando non applica effettivamente le regole al sistema, bensì si limita a fare parsing e a costruire il ruleset per verificarne la correttezza sintattica, senza eseguire il commit finale. L'esecuzione avviene tramite `subprocess.run`, con `shell=False`, prevenendo l'interpolazione di caratteri speciali da parte della shell e garantendo protezione contro attacchi di injection[3].

Relativamente alla gestione degli errori, qualsiasi fallimento durante la validazione o l'applicazione di una regola comporta il rifiuto dell'intera operazione, restituendo all'AI Agent un messaggio di errore dettagliato.

4.5.2 Endpoint REST Il modulo espone due endpoint principali:

- `/apply-rule` (POST): endpoint che riceve in input un payload JSON contenente i seguenti campi:
  - `rule`: la regola iptables da applicare
  - `reasoning`: motivazione per cui la regola viene applicata
  - `attack_data`: dizionario contenente i metadati dell'attacco rilevato dall'IDS (classificazione, confidence, flusso di rete)

  L'endpoint esegue la pipeline di validazione descritta precedentemente e, in caso di successo, applica la regola tramite `subprocess.run` con un timeout di 10 secondi. Ogni tentativo di applicazione, sia riuscito che fallito, viene registrato in un log strutturato contenente timestamp, regola, reasoning, attack data, esito e messaggio di errore (se presente).

- `/list_rules` (GET): endpoint di utilità che restituisce l'elenco completo delle regole iptables correntemente attive nel sistema, eseguendo il comando `iptables -L -n -v`. Tale endpoint è utilizzato per debugging e monitoring dello stato del firewall.

4.5.3 Deployment e Configurazione Il Firewall Enforcer è deployato all'interno del container Docker del firewall, condividendo lo stesso network namespace. Il Dockerfile installa i pacchetti necessari (`iptables`, `iproute2`) e il container viene eseguito con la capability `NET_ADMIN`, necessaria per modificare le regole di filtraggio del kernel. Il modulo è avviato automaticamente dallo script `entrypoint.sh` e rimane in ascolto sulla porta 5002, accessibile solo dalla rete interna (*internal_net*).

4.6 AI Security Agent. Tale modulo rappresenta il componente sperimentale dell'architettura, rappresentando l'obiettivo primario del progetto: valutare l'efficacia di un Large Language Model (LLM) nel contesto della risposta automatica alle minacce di rete. Il modulo agisce come un intermediario intelligente tra il sistema di detection (ovvero l'IDS) e il sistema di enforcement, traducendo classificazioni di attacco in regole firewall sintatticamente corrette e semanticamente appropriate.

L'agente adotta, analogamente al modulo IDS, un'architettura *polling-based*. Questa scelta progettuale è stata effettuata per mantenere coerenza architetturale e semplicità implementativa. L'intervallo di polling è configurabile tramite la variabile d'ambiente `POLLING_INTERVAL`, con un valore di default di 10 secondi.

4.6.1 Inizializzazione e configurazione Durante la fase di inizializzazione, il modulo esegue varie operazioni. Innanzitutto, si connette a Redis e tenta di recuperare il timestamp dell'ultimo batch di attacchi processato, interrogando il *Sorted Set* `agent_processed_index`. Viene, poi, istanziato un client **Ollama** che si connette al server LLM in esecuzione sull'host fisico. La connessione avviene tramite l'indirizzo `"host.docker.internal"`, che Docker risolve automaticamente all'IP dell'host. Il modello da utilizzare è configurabile tramite la variabile d'ambiente `OLLAMA_MODEL`, con valore di default "llama3:latest".

4.6.2 Loop Principale Di seguito verrà descritto con maggiore dettaglio il loop di esecuzione del modulo:

Fase 1: Recupero degli Attacchi Viene interrogato Redis per recuperare tutte le chiavi il cui score è strettamente maggiore dell'ultimo timestamp processato. Per ogni chiave recuperata, viene richiesto il payload JSON che contiene l'array degli attacchi rilevati dall'IDS. Per prevenire sovraccarichi del modulo e del firewall enforcer in caso di attacchi a più alta intensità, è stato scelto di includere 10 attacchi per batch.

Fase 2: Generazione delle Regole Firewall Per ogni attacco recuperato, viene, innanzitutto, creato un prompt che incapsula tutte le informazioni contestuali necessarie per la generazione di una regola appropriata (quali, ad esempio, tipo di attacco, grado di confidence della predizione, IP sorgente, IP di destinazione, ecc.). Vengono, inoltre, specificati requisiti espliciti (e.g. *generare esclusivamente il comando iptables, senza spiegazioni aggiuntive*, ecc.) e un esempio di formato atteso, in modo tale da indurre il modello a generare un output strutturato.
Il prompt viene inviato al modello tramite il client Ollama. Inoltre, poiché vi è la possibilità che l'LLM generi testo aggiuntivo (nonostante le istruzioni esplicite), è stata implementata una fase di post-processing in cui viene estratta esclusivamente la porzione dell'output rappresentante il comando, scartando eventuali contenuti aggiuntivi.
L'output di quesa fase è un dizionario contenente: la regola iptables generata, il razionale associato, i dati dell'attacco originale, il grado di confidence dell'IDS e un timestamp (ISO 8601).

Fase 3: Validazione Semantica e Applicazione In quest'ultima fase, per aumentare la robustezza, viene effettuata una verifica locale della semantica della regola, per evitare che quest'ultima faccia riferimento a un indirizzo IP errato. Se la validazione ha seuccesso, la regola viene inviata tramite HTTP POST al Firewall (endpoint */apply-rule*). Indipendentemente dall'esito, l'azione viene registrata in Redis in un log strutturato (chiave `agent_actions:<timestamp>`).
Infine, il timestamp viene aggiornato al timestamp relativo all'ultimo attacco processato.

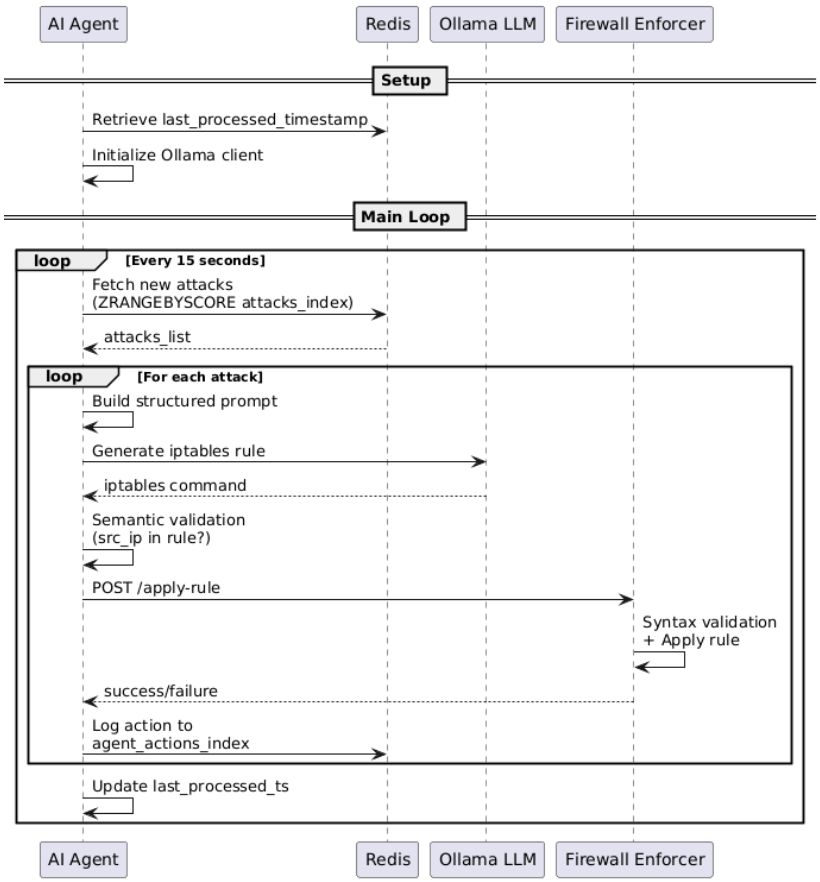


Figure 11: Sequence Diagram che mostra il flusso (semplificato) dell'agente

4.6.3 Osservazioni Durante la fase di testing del sistema, allo stato appena descritto, è emersa una limitazione in particolare, intrinseca all'approccio basato su prompt engineering puro, che motiva l'introduzione di un meccanismo di **RAG** (Retrieval-Augmented Generation). Considerando, ad esempio, attacchi di tipo *Port Scan*, l'LLM, operando esclusivamente sul contesto fornito dal singolo flusso di rete rilevato dall'IDS, tende a generare regole specifiche per la porta target invece di bloccare tutto il traffico (almeno temporaneamente) proveniente dall'indirizzo sorgente. Nell'esempio di seguito illustrato, l'attaccante esegue un Port Scan con *nmap*:

```
marcus@MacBook-Pro-di-Marco-2 architecture % docker exec -it attacker bash
5e98685c141f:/usr/src/metasploit-framework# nmap -T4 172.21.0.10
Starting Nmap 7.93 ( https://nmap.org ) at 2026-01-22 17:34 UTC
Nmap scan report for 172.21.0.10
Host is up (0.0000070s latency).
Not shown: 999 closed tcp ports (reset)
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.20 seconds
5e98685c141f:/usr/src/metasploit-framework#
```

Figure 12: Attacker avvia lo scan con nmap

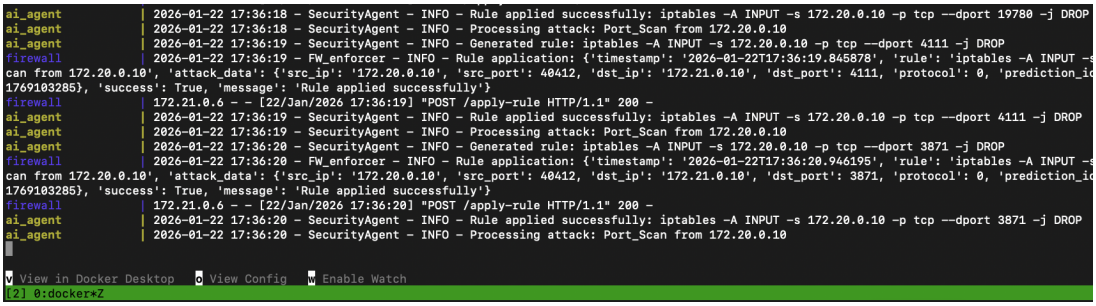


Figure 13: I logs mostrano le regole generate dall'agent

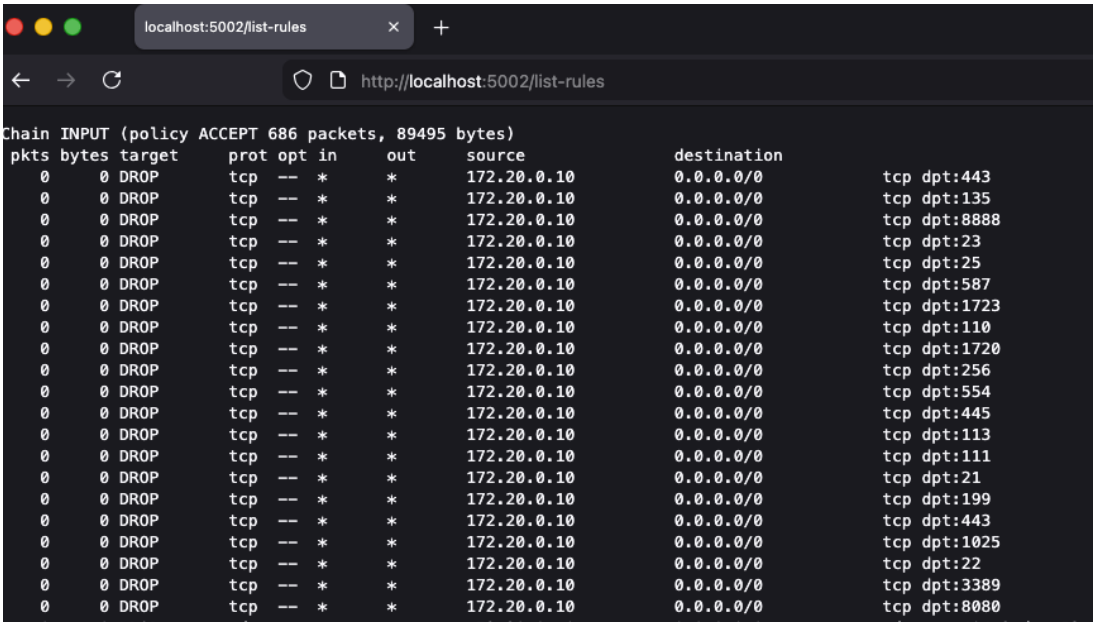


Figure 14: L'endpoint /list-rules mostra chiaramente che è stata generata una regola per ogni porta interessata dall'attacco

Questo comportamento, pur essendo semanticamente coerente rispetto al prompt fornito, nonché sintatticamente corretto, è estremamente inefficiente poiché, in primo luogo, non blocca l'attacco ma risulta anche nella degradazione delle performance del firewall poiché vengono generate centinaia di regole ridondanti (che differiscono soltanto per il numero di porta). In conclusione, questo evidenzia una mancanza di conoscenza del dominio da parte dell'LLM.
Tale osservazione empirica evidenzia la necessità di fornire al modello esempi validati di regole per specifiche tipologie di attacco, in modo da aumentare la domain knowledge dell'agente e ridurre, inoltre, la probabilità di allucinazioni del modello. Pertanto, è stato implementato un sistema RAG, che verrà descritto di seguito. [8]

4.6.4 Retrieval Augmented Generation (RAG). In base a quanto osservato nel precedente paragrafo, è stato ritenuto necessario implementare un meccanismo **RAG** (Retrieval-Augmented Generation) per migliorare la qualità e la consistenza delle regole generate dall'agente.

Cos'è il paradigma RAG. Il Retrieval-Augmented Generation è una tecnica che combina il recupero di informazioni da una knowledge base esterna con la generazione di testo tramite Large Language Model [5]. A differenza dei modelli puramente generativi, il RAG introduce un passaggio

intermedio di *retrieval* che consente al modello di accedere a conoscenza specifica del dominio durante la fase di generazione.
Il paradigma RAG si articola in tre fasi fondamentali [4]:

1. **Retrieve**: data una query, il sistema recupera documenti o esempi rilevanti da una knowledge base mediante ricerca semantica.

2. **Augment**: il contesto recuperato viene inserito nel prompt fornito al LLM, arricchendo l'input con informazione contestuale.

3. **Generate**: l'LLM genera l'output basandosi sia sulla query originale che sul contesto recuperato.

Questo approccio mitiga il problema delle *hallucinations* [8] e consente di aggiornare la knowledge base del modello senza richiedere costosi re-training. Nel contesto di questo progetto, il RAG permette all'agente di generare regole firewall coerenti con le best practice validate, fornendo esempi specifici per ogni tipologia di attacco.

Integrazione nell'architettura. Innanzitutto, è stata creata una *Knowledge Base* in formato JSON, (`iptables_rules.json`) contenente 14 entry, una per ciascuna tipologia di attacco supportata dal modello IDS. Ogni entry comprende il nome della tipologia di attacco (che la identifica), una descrizione del pattern di attacco, un template di esempio della regola di mitigazione e una spiegazione della strategia di mitigazione adottata.
Per il retrieval semantico è stato utilizzato **ChromaDB** [1], un database vettoriale open-source ottimizzato per applicazioni di *embedding* e *similarity search* . All'avvio dell'agente, la base di conoscenza viene indicizzata in una collection ChromaDB denominata `iptables_rules`.
Per ogni entry, viene generato un *document* testuale strutturato come:

```
"<attack_type>: <description>"
```

Questo testo viene convertito in un embedding vettoriale tramite un modello sentence-transformer predefinito. I metadati completi (rule, reasoning, description) vengono salvati come metadata associati, consentendo il loro recupero post-retrieval senza necessità di ricostruire il contesto.

Flusso di Generazione con RAG. Il processo di generazione delle regole è stato modificato per integrare il retrieval semantico. La Figura 15 illustra il flusso completo.

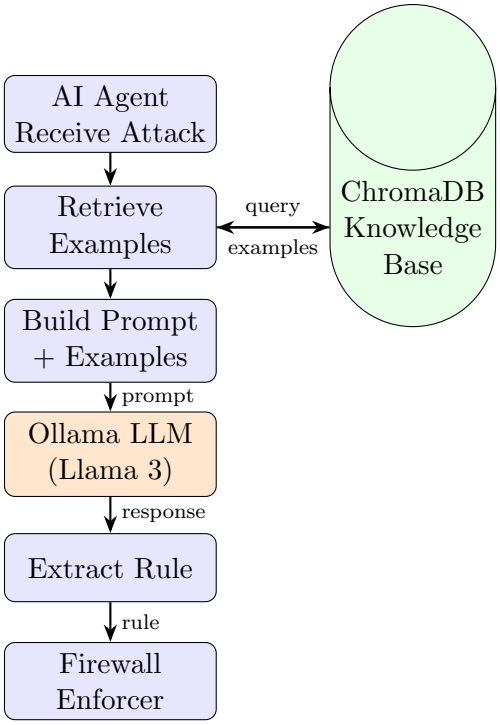


Figure 15: Flusso del sistema RAG nell'architettura dell'AI Agent.

Fondamentalmente, l'integrazione del paradigma RAG ha portato i seguenti vantaggi:

- per attacchi della stessa tipologia, il modello genera regole semanticamente equivalenti, eliminando la variabilità casuale osservata nel prompt engineering puro.
- il modello acquisisce maggiore conoscenza del dominio, potendo fare riferimento ad esempi validati
- è stata incrementata la scalabitlià, poiché la knowledge base può essere estesa con nuove tipologie di attacco senza richiedere re-training del LLM o modifica del codice.
- è stata migliorata l'efficienza, eliminando regole ridondanti (es. centinaia di regole port-specific per Port Scan), e sono state migliorate le performance del firewall, riducendo il carico computazionale.

Nota sulla deduplicazione. Non è stato implementato alcun sistema di deduplicazione delle regole. Allo stato attuale, infatti, un attacco continuo, come un DoS, che produce traffico malevolo per un intervallo di tempo prolungato, comporta la generazione ridondante della stessa regola di mitigazione da parte dell'agent.
Questo accade perché la cattura del traffico, da parte del traffic sniffer, che arriva all'IDS e, conseguentemente, all'agent (sottoforma di predizione), avviene *prima* che vengano applicate le regole di filtraggio. Di conseguenza, anche dopo che l'agent ha generato e applicato una regola di blocco, l'IDS continua a ricevere e classificare flussi provenienti dallo stesso attaccante, generando predizioni duplicate che innescano nuove generazioni di regole identiche.
In un ambiente di production, ovviamente, tale approccio risulterebbe impraticabile. Tuttavia, nell'ambito di questo progetto non costituisce una problema, bensì un vantaggio nell'analisi delle performance dell'agent. Infatti, trattando ogni predizione come un evento indipendente (l'agent non fa riferimento alla sua esperienza, ovvero a regole precedentemente generate), è possibile misurare il

grado di consistenza e **correttezza** delle regole generate dall'LLM su attacchi dello stesso tipo, raccogliendo multipli datapoint statistici da un singolo attaccante. Questo elimina la necessità di configurare molteplici macchine attaccanti con indirizzi IP distinti per ottenere un numero sufficiente di campioni per la valutazione.

4.7 Target Vulnerabile. Per rappresentare il target dell'attacco, è stato scelto di utilizzare *Damn Vulnerable Web Application* (**DVWA**), deployata come container Docker. DVWA è un'applicazione web PHP/MySQL deliberatamente vulnerabile, progettata per scopi didattici e di testing nel campo della cybersecurity [2].

4.8 Attaccante. Per simulare scenari di attacco realistici, è stato utilizzato *Metasploit Framework*, deployato come container Docker sulla rete esterna (*external_net*). Metasploit è il framework open-source più diffuso per penetration testing. All'avvio, viene configurata una route statica per permettere all'attaccante di raggiungere il target, passando per il firewall. Il framework è utilizzato per generare traffico malevolo di diverse categorie, coerenti con il dataset CIC-IDS2017 su cui è addestrato il modello ML. È stato anche realizzato uno script che simula accuratamente un attacco di tipo DDoS, caricato nel container sotto il nome di `dos.py`.

5 Caso d'uso

Verrà, ora, illustrato un caso d'uso in cui verrà effettuato un attacco di tipo Port Scan.

1. Deployment dell'infrastruttura Il sistema viene deployato mediante Docker Compose, che orchestra l'avvio di tutti i container e la configurazione delle reti.

```
marcus@MacBook-Pro-di-Marco-2 architecture % docker compose up
WARN[0000] No services to build
[+] up 7/9
[+] up 10/10rchitecture_internal_net Created
 ✔ Network architecture_internal_net Created
 ✔ Network architecture_external_net Created
 ✔ Container dvwa                    Created
 ✔ Container attacker                Created
 ✔ Container redis                   Created
es not match the detected host platform (linux/arm64/v8) and no specific plat
 ✔ Container feature_extractor       Created
 ✔ Container firewall                Created
 ✔ Container ids                     Created
 ✔ Container ai_agent                Created
Attaching to ai_agent, attacker, dvwa, feature_extractor, firewall, ids, redi
redis  | 1:C 27 Jan 2026 11:01:00.332 * oO0OoO0OoO0Oo Redis is starting oO0Oo
redis  | 1:C 27 Jan 2026 11:01:00.332 * Redis version=7.4.7, bits=64, commit=
[10] 0:[tmux]*Z
```

Figure 16: Deployment dell'infrastruttura

```
ids              | 2026-01-27 11:01:51 - IDS - INFO - Fetching files after timestamp 1769450921.0...
ids              | 2026-01-27 11:01:51 - IDS - INFO - Found 0 new file(s)
ids              | 2026-01-27 11:01:51 - IDS - INFO - No new files to retrieve
ids              | 2026-01-27 11:01:51 - IDS - INFO - No data to preprocess.
ids              | 2026-01-27 11:01:51 - IDS - INFO - Sleeping for 5 seconds...
ids              | 2026-01-27 11:01:56 - IDS - INFO - Fetching files after timestamp 1769450921.0...
ids              | 2026-01-27 11:01:56 - IDS - INFO - Found 0 new file(s)
ids              | 2026-01-27 11:01:56 - IDS - INFO - No new files to retrieve
ids              | 2026-01-27 11:01:56 - IDS - INFO - No data to preprocess.
ids              | 2026-01-27 11:01:56 - IDS - INFO - Sleeping for 5 seconds...
ids              | 2026-01-27 11:02:01 - IDS - INFO - Fetching files after timestamp 1769450921.0...
ids              | 2026-01-27 11:02:01 - IDS - INFO - Found 0 new file(s)
ids              | 2026-01-27 11:02:01 - IDS - INFO - No new files to retrieve
ids              | 2026-01-27 11:02:01 - IDS - INFO - No data to preprocess.
ids              | 2026-01-27 11:02:01 - IDS - INFO - Sleeping for 5 seconds...
ids              | 2026-01-27 11:02:06 - IDS - INFO - Fetching files after timestamp 1769450921.0...
ids              | 2026-01-27 11:02:06 - IDS - INFO - Found 0 new file(s)
ids              | 2026-01-27 11:02:06 - IDS - INFO - No new files to retrieve
ids              | 2026-01-27 11:02:06 - IDS - INFO - No data to preprocess.
ids              | 2026-01-27 11:02:06 - IDS - INFO - Sleeping for 5 seconds...
```

Figure 17: Logs del modulo di IDS

2. Avvio di nmap L'attaccante avvia il port scan con nmap.

```
marcus@MacBook-Pro-di-Marco-2 architecture % docker exec -it attacker nmap 172.21.0.10
Starting Nmap 7.93 ( https://nmap.org ) at 2026-01-27 11:06 UTC
Nmap scan report for 172.21.0.10
Host is up (0.0000060s latency).
Not shown: 999 closed tcp ports (reset)
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.18 seconds
marcus@MacBook-Pro-di-Marco-2 architecture %
```

Figure 18: Port scan con nmap

Logs Traffic Sniffer e Feature Extractor I pacchetti vengono catturati dal traffic sniffer che, dopo aver raccolto un numero di pacchetti pari alla batch size, in questo caso impostata a 100, invia l'output risultante al feature extractor.

```
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [99] TCP | 172.20.0.10:40346 -> 172.21.0.10:873 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [100] TCP | 172.20.0.10:40346 -> 172.21.0.10:6156 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [101] TCP | 172.20.0.10:40346 -> 172.21.0.10:49157 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [102] TCP | 172.20.0.10:40346 -> 172.21.0.10:2607 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [103] TCP | 172.20.0.10:40346 -> 172.21.0.10:5009 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [104] TCP | 172.20.0.10:40346 -> 172.21.0.10:9917 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [105] TCP | 172.20.0.10:40346 -> 172.21.0.10:5432 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [106] TCP | 172.20.0.10:40346 -> 172.21.0.10:7200 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [107] TCP | 172.20.0.10:40346 -> 172.21.0.10:1233 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [108] TCP | 172.20.0.10:40346 -> 172.21.0.10:5901 | Len: 58
firewall         | 2026-01-27 11:06:20 - PacketSniffer - INFO - [109] TCP | 172.20.0.10:40346 -> 172.21.0.10:3013 | Len: 58
feature_extractor | 2026-01-27 11:06:20 - feature_extractor - INFO - Received new PCAP for processing: /shared/pcap/capture_20260127_110620.pcap
feature_extractor | 172.21.0.5 - - [27/Jan/2026 11:06:20] "POST /new_pcap HTTP/1.1" 200 -
```

Figure 19: Packet sniffer cattura pacchetti e li invia, tramite HTTP POST al feature extractor

3. IDS rileva attacco L'IDS analizza il traffico e rileva l'attacco.

```
ids | 2026-01-27 11:06:21 - IDS - INFO - Found 1 new file(s)
ids | 2026-01-27 11:06:21 - IDS - INFO - Loaded 87 rows from features:capture_20260127_110620.csv
ids | 2026-01-27 11:06:21 - IDS - INFO - Successfully merged 1 file(s) into DataFrame with 87 rows
ids | 2026-01-27 11:06:21 - IDS - INFO - === STARTING PREPROCESS ===
ids | 2026-01-27 11:06:21 - IDS - INFO - Starting preprocessing of 87 rows...
ids | 2026-01-27 11:06:21 - IDS - INFO - Encoding 'protocol' column using protocol_encoder...
ids | 2026-01-27 11:06:21 - IDS - INFO - Preprocessing completed: (87, 40)
ids | 2026-01-27 11:06:21 - IDS - INFO - === STARTING PREDICTION ===
ids | 2026-01-27 11:06:21 - IDS - INFO - Starting prediction for 87 flows...
ids | 2026-01-27 11:06:21 - IDS - INFO - Prediction completed: 85 attacks, 2 benign flows
ids | 2026-01-27 11:06:21 - IDS - INFO - Average confidence: 0.746
ids | 2026-01-27 11:06:21 - IDS - WARNING - Detected 85 ATTACK flows - saved to attacks:1769511981
ids | 2026-01-27 11:06:21 - IDS - INFO - Saved 87 predictions to Redis: predictions:1769511981
ids | 2026-01-27 11:06:21 - IDS - INFO - Updated last processed timestamp to 1769511980.0
ids | 2026-01-27 11:06:21 - IDS - INFO - Sleeping for 5 seconds...
```

Figure 20: IDS rileva l'attacco e salva le prediction su Redis

4. L'AI Agent genera le regole L'AI agent, sulla base della prediction effettuata dall'IDS, recupera gli esempi da ChromaDB, genera una regola e la applica, inviando una HTTP POST al modulo di enforcement del firewall, che la applica correttamente.

```
ai_agent | 2026-01-27 11:20:03 - SecurityAgent - INFO - Rule applied successfully: iptables -A FORWARD -s 172.20.0.10 -p tcp --dport 1024 -j DROP
ai_agent | 2026-01-27 11:20:03 - SecurityAgent - INFO - Processing attack: Port_Scan from 172.20.0.10
ai_agent | 2026-01-27 11:20:03 - SecurityAgent - INFO - Retrieving examples for: Port_Scan
ai_agent | 2026-01-27 11:20:04 - SecurityAgent - INFO - Retrieved 1 examples
ai_agent | 2026-01-27 11:20:04 - SecurityAgent - INFO - RAG Example Retrieved:
ai_agent | 2026-01-27 11:20:04 - SecurityAgent - INFO -   - Attack Type: Port_Scan
ai_agent | 2026-01-27 11:20:04 - SecurityAgent - INFO -   - Rule Template: iptables -A FORWARD -s {src_ip} -j DROP
ai_agent | 2026-01-27 11:20:04 - SecurityAgent - INFO -   - Reasoning: Blocks ALL traffic from scanner IP, not individual ports. Prevents hundreds of redundant rules.
```

Figure 21: Agent recupera esempi

```
ai_agent | 2026-01-27 11:20:05 - SecurityAgent - INFO - Generated rule: iptables -A FORWARD -p 0 -s 172.20.0.10 -j DROP
firewall | 2026-01-27 11:20:05 - FW_enforcer - INFO - Rule application: {'timestamp': '2026-01-27T11:20:05.228351', 'rule': 'iptables -A FORWARD -p 0 -s 172.20.0.10 -j DROP', 'reasoning': 'Mitigating Port_Scan from 172.20.0.10', 'attack_data': {'src_ip': '172.20.0.10', 'src_port': 40346, 'dst_ip': '172.21.0.10', 'dst_port': 1864, 'protocol': 0, 'prediction_id': 9, 'prediction': 'Port_Scan', 'confidence': 0.6897142857, 'timestamp': 1769512782}, 'success': True, 'message': 'Rule applied successfully'}
firewall | 172.21.0.6 - - [27/Jan/2026 11:20:05] "POST /apply-rule HTTP/1.1" 200 -
ai_agent | 2026-01-27 11:20:05 - SecurityAgent - INFO - Rule applied successfully: iptables -A FORWARD -p 0 -s 172.20.0.10 -j DROP
```

Figure 22: Agent genera la regola

Regole correttamente applicate L'indirizzo IP dell'attaccante è stato bloccato.

```
marcus@MacBook-Pro-di-Marco-2 architecture % ping -c 10 172.21.0.10
PING 172.21.0.10 (172.21.0.10): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
Request timeout for icmp_seq 3
Request timeout for icmp_seq 4
Request timeout for icmp_seq 5
Request timeout for icmp_seq 6
Request timeout for icmp_seq 7
Request timeout for icmp_seq 8

--- 172.21.0.10 ping statistics ---
10 packets transmitted, 0 packets received, 100.0% packet loss
```

Figure 23: ping mostra che l'attaccante non può raggiungere il target

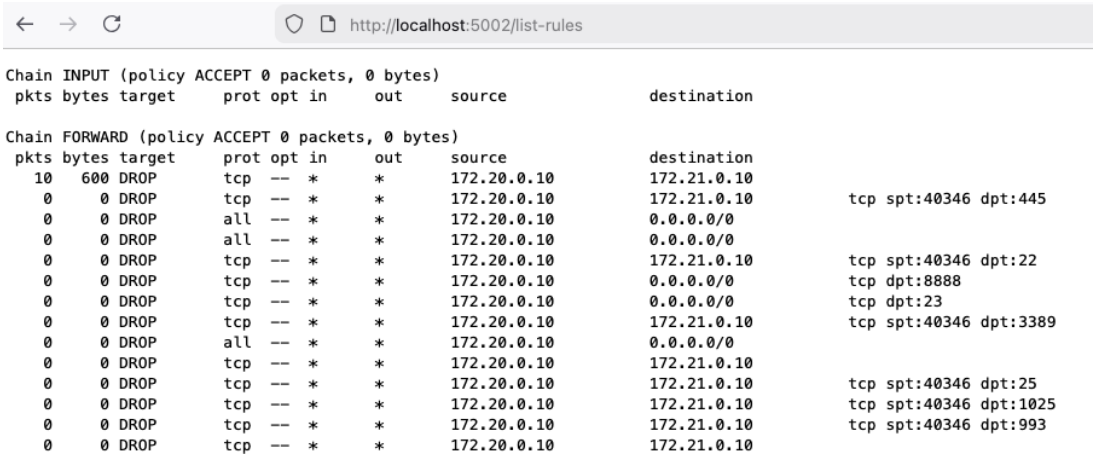


Figure 24: Alcune delle regole che sono state applicate. Si notano alcune regole semanticamente errate

6 Framework di valutazione

La metodologia di valutazione scelta si concentra sulla qualità delle risposte dell'agente nell'ambiente simulato appena descritto.
Il framework di valutazione, implementato nello script `evaluator.py`, adotta un approccio rule-based per verificare la correttezza delle regole generate. Infatti, per ogni tipologia di attacco supportata, è stata definita una specifica ground truth, che riporta: **must_have**, ovvero elementi sintattici che devono essere obbligatoriamente presenti nella regola, **must_not_have**, ovvero elementi sintattici che non devono comparire nella regola e una **description**, cioè la motivazione della strategia di mitigazione attesa.
Tale approccio consente una valutazione deterministica e automatica.

6.1 Processo di valutazione

Il processo di valutazione si articola su più fasi.

Fase 1 - Recupero delle azioni dell'agente. Innanzitutto, l'evaluator recupera da Redis tutte le azioni registrate dall'agente durante la fase di testing, memorizzate con la chiave `agent_actions:{ts}`.

Fase 2 - Estrazione dei metadati. Per ogni azione, vengono estratti:

- Tipologia di attacco (`prediction`)
- La regola iptables generata (`rule`)
- L'indirizzo IP sorgente (`src_ip`)
- Lo stato di successo dell'applicazione della regola (`success`)

Fase 3 - Validazione semantica. Viene, poi, verificato che la regola contenga tutti gli elementi obbligatori definiti nella metrica per l'attacco riportato nella prediction, che la regola non contenga pattern vietati e che faccia riferimento all'indirizzo IP sorgente corretto.

Fase 4 - Aggregazione dei risultati. Infine, vengono calcolate le metriche di accuracy per ogni tipologia di attacco e le metriche di accuracy complessive. Si noti che vengono considerate solo le regole applicate con successo, ovvero quelle con `success=True`.

6.2 Risultati

La valutazione è stata condotta su un totale di **2578** regole, generate dall'agent in risposta agli attacchi simulati. La tabella 1 riporta i risultati aggregati per tipologia di attacco.

Table 1: Risultati della valutazione dell'AI Agent per tipologia di attacco

Attack Type	Correct	Wrong	Total	Accuracy
Port Scan	934	89	1023	91.30%
DDoS LOIT	1554	0	1554	100.00%
DoS Hulk	1	0	1	100.00%
OVERALL	**2489**	**89**	**2578**	**96.55%**

Il sistema ha raggiunto un'accuracy complessiva del **96.55%**, con 2489 regole corrette su 2578 generate.
Per gli attacchi di tipo Denial of Service, l'agente ha dimostrato performance perfette. La strategia di mitigazione adottata dal modello è consistente e corretta: blocco totale del traffico proveniente dall'indirizzo IP sorgente mediante regole del tipo:

```
iptables -A INPUT -s <attacker_ip> -j DROP
```

Relativamente agli attacchi di tipo Port Scan, l'agente ha generato 89 regole errate su 1023. Sebbene l'integrazione del RAG abbia migliorato le performance dell'agent, il tasso di errore è relativamente elevato, considerando la criticità del contesto di applicazione di tale sistema. Tuttavia, va anche considerato che le regole considerate errate non sono sintatticamente sbagliate, ma risultano semanticamente subottimali (poiché, bloccando singolarmente ogni porta su cui viene rilevato l'attacco, come spiegato precedentemente, non si riesce a bloccare un attacco in corso).

6.3 Limitazioni e sviluppi futuri

È stato possibile valutare i risultati principalmente su due tipologie di attacchi: Port Scan e DoS (Denial of Service), questo a causa della migliore capacità del modello di Intrusion Detection (descritto nella sezione 3) di rilevare tali tipologie: la maggior parte dei campioni, infatti, fa riferimento, ovviamente, a traffico legittimo, Port Scan e alcune tipologie di DoS. Purtroppo, per questo progetto, non è stato possibile addestrare il modello utilizzando un dataset più grande e aggiornato, come il *BCCC-CIC-IDS2018*, a causa di limitazioni hardware che avrebbero reso impraticabile l'addestramento.
Inoltre, l'attuale implementazione del RAG si limita a recuperare esempi statici dalla knowledge base iniziale, senza sfruttare le regole generate con successo come feedback per l'apprendimento incrementale dell'agente. Tale feature potrebbe sicuramente essere aggiunta in futuro.
Altra limitazione, particolarmente critica in un contesto di sicurezza come quello in esame, è intrinseca agli LLM ed è costituita dal loro comportamento stocastico, il quale introduce variabilità nelle risposte, anche con un RAG attivo. Tale fattore di rischio, non deve necessariamente significare l'inadeguatezza di un tale sistema in contesti critici, tuttavia suggerisce, ad esempio, l'integrazione del paradigma *human-in-the-loop*, in cui è l'uomo a dare l'approvazione finale per l'applicazione della regola generata dall'agente.
Come già discusso in precedenza, non è stato implementato alcun meccanismo di deduplicazione: l'agente genera regole duplicate per attacchi continui. Questo, pur consentendo la raccolta di multipli

campioni statistici per la valutazione, è impraticabile in produzione e richiederebbe un meccanismo di state management.
Infine, l'approccio rule-based adottato classifica le regole come *corrette* o *errate* in modo binario. Non viene considerato il grado di efficacia relativa (e.g. una regola port-specific per Port Scan è errata ma comunque parzialmente efficace). Pertanto, metriche più sofisticate potrebbero fornire informazioni più dettagliate.
Alla luce di quanto detto, emergono diverse possbilità di miglioramento di questo paradigma, di seguito brevemente illustrate:

- Fine-tuning del Large Language Model, per addestrare un modello specializzato e ridurre la variabilità.
- Ensemble di validazione: introdurre, ad esempio, un secondo LLM che agisca da validator, verificando la regola generata prima di applicarla e richiedendo una rigenerazione in caso di errore.
- Retrieval dinamico con feedback: implementare un meccanismo di reinforcement learning che aggiorni i pesi del retrieval in base al feedback degli amministratori.
- Espansione della knowledge base con esempi multipli per ciascuna tipologia di attacco, edge cases, ecc.
- Human-in-the-Loop: implementare un workflow di approvazione semi-automatico in cui le regole generate dall'agente AI vengono presentate a un amministratore di sistema prima dell'applicazione effettiva.

7 Conclusioni

In conclusione, i risultati ottenuti dimostrano che l'integrazione di un agente AI basato su Large Language Model e Retrieval-Augmented Generation per la generazione automatica di regole firewall è tecnicamente fattibile e può raggiungere livelli di accuracy elevati (96.55%). Tuttavia, la transizione da un ambiente sperimentale controllato a un ambiente di production comporta il superamento di sfide ancora aperte, in quanto un tale sistema presenta limitazioni intrinseche legate alla natura probabilistica dei Large Language Models, che introducono un elemento di non determinismo, incompatibile con contesti mission-critical, dove la prevedibilità assoluta è requisito fondamentale. In tali scenari, l'agente AI potrebbe essere utilizzato come strumento di supporto alla decisione umana (*human-in-the-loop*), piuttosto che come sistema completamente autonomo. Il framework sviluppato fornisce, pertanto, una base per future ricerche in questo dominio emergente.

References

[1] chromadb. Chroma — trychroma.com. https://www.trychroma.com/. [Accessed 25-01-2026].

[2] DVWA Team. Damn vulnerable web application. https://github.com/digininja/DVWA, 2024. Accessed: 2026-01-18.

[3] Python Software Foundation. subprocess — Subprocess management — docs.python.org. https://docs.python.org/3/library/subprocess.html#security-considerations. [Accessed 22-01-2026].

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[5] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[6] man7. iptables-restore(8) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man8/iptables-restore.8.html. [Accessed 22-01-2026].

[7] MohammadMoein Shafi, Arash Habibi Lashkari, and Arousha Haghighian Roudsari. Ntlflowlyzer: Towards generating an intrusion detection dataset and intruders behavior profiling through network and transport layers traffic analysis and pattern extraction. *Computers & Security*, 148:104160, 2025.

[8] Wikipedia. Retrieval augmented generation — Wikipedia, the free encyclopedia. http://it.wikipedia.org/w/index.php?title=Retrieval%20augmented%20generation&oldid=148895962, 2026. [Online; accessed 22-January-2026].