Predictive Deception — LLM-based Command Anticipation in SSH Honeypots

Raffaele Neri
raffaele.neri2@studio.unibo.it
Università di Bologna
Gravina in Puglia , Puglia, Italia

Matteo Melotti
matteo.melotti5@studio.unibo.it
Università di Bologna
Bologna, Emilia Romagna, Italia

Enrico Borsetti
enrico.borsetti@studio.unibo.it
Università di Bologna
San Lazzaro di Savena, Emilia Romagna,
Italia

1 Introduzione

Gli SSH honeypot tradizionali operano secondo un modello puramente reattivo: registrano e analizzano i comandi degli attaccanti solo dopo la loro esecuzione. Sebbene tale approccio risulti efficace nella raccolta di informazioni, esso limita la capacità del sistema di anticipare le mosse dell'intrusore e riduce il potenziale delle tecniche di inganno, che rimangono passive e dipendenti dall'iniziativa dell'attaccante. Il problema centrale risiede dunque nell'incapacità dei sistemi attuali di **prevedere il comportamento dell'utente malevolo**, lasciando scoperta un'importante opportunità di difesa proattiva.

A ciò si aggiunge un'ulteriore criticità: gli honeypot tradizionali, proprio a causa del loro comportamento statico e prevedibile, risultano spesso relativamente semplici da identificare per un attaccante esperto. Indicatori come risposte di sistema poco realistiche, assenza di alcune funzionalità tipiche di un ambiente reale o pattern ripetitivi nell'interazione possono rivelarne la natura artificiale. Una volta riconosciuto l'inganno, l'attaccante può interrompere l'azione o, peggio, manipolare l'honeypot al fine di depistare il difensore, compromettendo l'efficacia complessiva del sistema di monitoraggio.

In questo lavoro esploriamo un paradigma alternativo, la *deception predittiva*, nel quale un Large Language Model (LLM) analizza in tempo reale la sequenza di comandi di una sessione di attacco e stima quale sarà la prossima azione dell'attaccante. Questa capacità consente al sistema di predisporre in anticipo artefatti ingannevoli — come file, configurazioni o credenziali — e di esporli all'attaccante prima che vengano richiesti.

Basandoci su dataset pubblici su larga scala,come il *CyberLab Honeynet* , valutiamo la fattibilità e l'efficacia dell'integrazione di modelli LLM predittivi nei flussi operativi di un SSH honeypot, analizzando il loro impatto sulle capacità di rilevazione, sul tempo di ingaggio e sulla qualità dell'intelligence generata.

In sintesi, questo lavoro si propone di:

- **valutare** se un LLM è in grado di predire il prossimo comando in una sessione SSH e quali fattori vanno ad incidere sulla qualità della predizione eseguita;
- **integrare** tali predizioni nella generazione di artefatti di deception coerenti con il comportamento dell'attaccante;
- **posizionare** gli artefatti prodotti all'interno del filesystem dell'honeypot per l'interazione con l'attaccante;

Contents

2 Background e Stato dell'Arte

Gli SSH honeypot rappresentano strumenti fondamentali per analizzare il comportamento degli attaccanti dopo un accesso non autorizzato. Tra questi, **Cowrie** è uno dei sistemi più diffusi grazie alla sua capacità di emulare un'istanza SSH realistica, registrare comandi, trasferimenti di file, interazioni con il filesystem e sequenze complete di comandi. Tuttavia, tali honeypot operano secondo un paradigma **puramente reattivo**: osservano e registrano attività già avvenute, senza capacità predittive o adattive.

Le tecniche di deception tradizionali si concentrano sulla creazione di ambienti credibili e di artefatti ingannevoli — come file sensibili, configurazioni o credenziali — con l'obiettivo di aumentare l'ingaggio dell'attaccante e ottenere intelligence di valore.

Parallelamente, l'avanzamento dei *Large Language Models* (LLM) ha mostrato la capacità dei modelli di apprendere e generare sequenze complesse, come codice, script o log di sistema. Studi recenti nel campo della sicurezza[1] dimostrano che gli LLM possono comprendere, seppur con qualche limitazione, contesti tecnici e suggerire azioni coerenti con un obiettivo malevolo o difensivo. Ciò apre la possibilità di utilizzare tali modelli per analizzare sessioni SSH in tempo reale e **prevedere le azioni successive dell'attaccante**.

La letteratura sugli honeypot evidenzia i limiti dei sistemi puramente passivi e la crescente necessità di tecniche più dinamiche e adattive.

3 Dataset

Il modello predittivo è stato testato utilizzando dataset pubblici di attacchi SSH avvenuti all'interno di un honeypot **Cowrie**. La loro combinazione fornisce sia ampiezza quantitativa sia profondità qualitativa, permettendo di catturare pattern ricorrenti e strategie reali adottate dagli attaccanti.

CyberLab Honeynet Dataset. Il dataset, pubblicato su Zenodo, contiene nove mesi di log provenienti da circa cinquanta honeypot Cowrie distribuiti geograficamente. Ogni file rappresenta una sessione di attacco giornaliera, strutturato in formato JSON e include eventi di sessione, comandi eseguiti, interazioni con il filesystem e metadata associati.

3.1 Download e Decompressione del Dataset

Per automatizzare la fase di acquisizione dei file dal record Zenodo è stato sviluppato uno script dedicato che permette sia il download completo sia la selezione casuale di un numero limitato di file.

Lo script:

- interroga l'API Zenodo per ottenere la lista dei file disponibili;
- supporta salvataggio locale o su dispositivo esterno (due modalità differenti: la seconda si appoggia su decompressione locale per maggiore velocità di esecuzione);
- controlla lo spazio disponibile sul dispositivo target;
- gestisce il download progressivo dei file `.gz` con barre di avanzamento;
- esegue la decompressione in formato `.json`(come sottolineato in precedenza, in entrambe le modalità la decompressione avviene sempre in locale);
- evita la riscrittura di file già scaricati.

Un estratto della funzione principale `downloading_and_decompression()` è riportato di seguito:

[1] Tra i lavori più rilevanti si citano *PentestGPT*, *LLMSec*, *HoneyGPT* e varie analisi su capacità offensive/difensive dei modelli generativi, che evidenziano come gli LLM possano comprendere contesti tecnici, generare payload realistici e supportare attività di ricognizione e attacco.

```python
def downloading_and_decompression(args):
    api = f"https://zenodo.org/api/records/{RECORD_ID}"
    meta = requests.get(api).json()
    gz_files = [f for f in meta["files"] if f["key"].endswith(".gz")]

    # Selezione dei file: random oppure tutti
    if args.n > 0:
        selected = random.sample(gz_files, args.n)
    else:
        selected = gz_files

    # Controllo spazio disponibile
    free = shutil.disk_usage(args.dst_path).free
    if free < total_gz_size:
        sys.exit("Spazio insufficiente")

    # Download e decompressione
    for file in selected:
        scarica .gz  →  decomprimi  →  sposta nella cartella finale
```

Figure 1: Estratto dello script per il download e la decompressione dal record Zenodo.

Questa procedura permette la riproducibilità dell'intero dataset anche su sistemi con spazio limitato, grazie alla gestione progressiva dei file.

3.2 Preprocessing

La fase di preprocessing è stata realizzata tramite due script principali, `analyze_and_clean.py` e `merge_cowrie_datasets.py`, che trasformano i log grezzi del dataset CyberLab in sequenze di comandi strutturate e riutilizzabili per l'analisi predittiva.

In sintesi, `analyze_and_clean.py` si occupa di estrarre le sessioni a partire dagli eventi `cowrie.command.input`, ricostruire la sequenza di comandi per ogni `session_id` e applicare funzioni di pulizia dedicate (ad esempio `normalize_command()` per la normalizzazione e `filter_short_sessions()` per la selezione delle sessioni sufficientemente lunghe).

Lo script `merge_cowrie_datasets.py` opera invece su più file giornalieri, invocando iterativamente `analyze_and_clean.py` e producendo un dataset unico, privo di duplicati, con uno split standard `TRAIN`/`TEST` e un insieme di statistiche aggregate sull'intero dataset.

Nel complesso, il preprocessing affronta le principali criticità tipiche dei log di honeypot — **rumore**, **duplicati**, **sessioni incomplete**, **errori di encoding** — e restituisce un dataset pulito, consistente e adatto alla valutazione del modulo predittivo. I dettagli operativi delle singole fasi sono descritti nelle sottosezioni seguenti.

3.2.1 Estrazione delle sessioni

Ogni file JSON giornaliero del dataset CyberLab contiene numerose sessioni rappresentate come dizionari annidati. Lo script `analyze_and_clean.py` ricostruisce le sequenze di comandi tramite:

- selezione degli eventi con `eventid = cowrie.command.input`;
- ordinamento cronologico degli eventi della giornata;
- costruzione della sequenza di comandi per ciascun `session_id`.

A tal fine viene utilizzata una struttura `defaultdict(list)` per raggruppare i comandi per sessione, garantendo una ricostruzione fedele dell'ordine di esecuzione e preservando il contesto temporale di ogni interazione.

3.2.2 Pulizia e normalizzazione dei comandi

La pulizia delle stringhe di comando è gestita dalla funzione `normalize_command(cmd)`, che si occupa di:

- rimozione di prefissi non necessari (es. `CMD:` o marker specifici del logger);
- mascheramento di elementi sensibili, quali:
  - password in comandi del tipo `echo | passwd`;
  - path in directory temporanee (`/tmp`, `/var/tmp`);
  - URL e indirizzi IP, quando si utilizza la modalità `CLEAN`;
- normalizzazione della spaziatura;
- eliminazione di componenti ridondanti o non informative.

Il risultato è una rappresentazione più compatta e uniforme dei comandi, che riduce la variabilità sintattica non rilevante e facilita il confronto tra sequenze diverse.

```python
def normalize_command(cmd: str) -> str:
    cmd = cmd.strip()
    cmd = re.sub(r'^CMD:\s*', '', cmd)
    cmd = re.sub(r'echo\s+-e\s+"[^"]+"(\|passwd\|bash)?', 'echo <SECRET>|passwd', cmd)
    cmd = re.sub(r'echo\s+"[^"]+"\|passwd', 'echo <SECRET>|passwd', cmd)
    cmd = re.sub(r'/var/tmp/[\.\w-]*\d{3,}', '/var/tmp/<FILE>', cmd)
    cmd = re.sub(r'/tmp/[\.\w-]*\d{3,}', '/tmp/<FILE>', cmd)
    cmd = re.sub(r'\b[\w\.-]+\.(log|txt|sh|bin|exe|tgz|gz)\b', '<FILE>', cmd)
    cmd = re.sub(r'(https?|ftp)://\S+', '<URL>', cmd)
    cmd = re.sub(r'\b\d{1,3}(?:\.\d{1,3}){3}\b', '<IP>', cmd)
    cmd = re.sub(r'echo\s+"admin\s+[^"]+"', 'echo "admin <SECRET>"', cmd)
    cmd = re.sub(r'\s+', ' ', cmd).strip()
    return cmd
```

Figure 2: Funzione `normalize_command()`.

3.2.3 Filtraggio delle sessioni brevi

Per ridurre il contributo di bot molto semplici o connessioni parziali viene applicato un filtro basato sulla lunghezza minima delle sessioni. La funzione `filter_short_sessions()`:

- mantiene solo le sessioni con numero di comandi $\geq N$, dove N è un parametro impostato dall'utente;
- scarta automaticamente le sequenze troppo brevi per essere significative dal punto di vista comportamentale.

Questo passaggio riduce sensibilmente il rumore e concentra l'analisi su sessioni in cui l'attaccante ha eseguito un numero sufficiente di azioni permettendo di osservare pattern più strutturati.

3.2.4 Aggregazione finale e merge dei file

Lo script `merge_cowrie_datasets.py` elabora l'insieme completo dei file giornalieri e produce:

- un file `RAW` con comandi non normalizzati e/o un file `CLEAN` con comandi normalizzati;
- un file `TRAIN` ed un file `TEST` (split casuale 70%–30%);
- un riepilogo delle statistiche di aggregazione.

Durante il merge:

- vengono eliminati i duplicati basati sull'intera sequenza di comandi;
- vengono raccolte statistiche globali sugli eventi osservati (conteggi per `eventid`, distribuzione delle lunghezze, ecc.);

3.3 Problemi Riscontrati

Durante la pulizia del dataset sono stati individuati diversi problemi comuni nei log di honeypot:

- **Rumore**: molte sessioni consistono in tentativi automatizzati privi di significato;
- **Comandi duplicati**: bot e worm ripetono ciclicamente comandi identici;
- **Sessioni incomplete**: connessioni chiuse dopo pochi secondi;
- **Errori di encoding**: presenza di caratteri non UTF-8 o stringhe corrotte.

La pipeline di preprocessing riduce tali effetti filtrando le sessioni brevi e normalizzando tutte le sequenze, producendo un dataset pulito, coerente e adatto alle fasi successive del modello predittivo.

4 Architettura del Sistema

L'architettura proposta integra un modello LLM all'interno di un honeypot SSH con l'obiettivo di prevedere, in tempo reale, il comando successivo dell'attaccante e generare artefatti prima che vengano richiesti. Il sistema è articolato in due componenti principali: il **modulo predittivo basato su LLM** e il **motore di deception**. L'intera pipeline opera durante l'interazione dell'attaccante con il sistema, aggiornando costantemente il contesto e producendo predizioni.

4.1 Raccolta in Tempo Reale

Il sistema intercetta i comandi registrati nel log e costruisce una **finestra di contesto aggiornata** per ciascuna sessione, basata sulla sequenza dei comandi grezzi osservati all'interno dell'honeypot. In questo modo, la fase operativa utilizza direttamente il comportamento reale dell'attaccante, senza ulteriori trasformazioni, concentrandosi esclusivamente sulla predizione del comando successivo a partire dalla history della sessione.

Il modulo mantiene gli ultimi k comandi come **finestra attiva di contesto**, aggiornata a ogni interazione, e la utilizza come input per la generazione del prompt.

4.2 Modulo Predittivo e Pipeline di Valutazione

Il modulo predittivo supporta quattro varianti distinte, ottenute combinando il modello LLM utilizzato (**CodeLlama** o **Gemini**) con l'eventuale approccio **RAG (Retrieval Augmented Generation)**. Le configurazioni disponibili sono quindi:

- **CodeLlama senza RAG**
- **CodeLlama con RAG**

- **Gemini senza RAG**
- **Gemini con RAG**

Nonostante le differenze implementative, tutte seguono lo stesso principio: dato il contesto corrente, con o senza esempi aggiuntivi, il modello genera un insieme di possibili comandi successivi ordinati per probabilità.

$$(c_1, \ldots, c_k) \rightarrow \{\hat{c}_{k+1}^{(1)}, \ldots, \hat{c}_{k+1}^{(n)}\}$$

4.2.1 TopK

Le configurazioni TopK adottano un approccio di solo prompting, basato esclusivamente sulla finestra di contesto corrente della sessione. In questo scenario, il modello LLM non dispone di memoria storica o esempi recuperati dal database vettoriale: la predizione viene ottenuta analizzando unicamente gli ultimi comandi digitati dall'attaccante. Lo script principale che racchiude la logica fondamentale per l'esecuzione di questo approccio è affidata allo script `core_topk.py`

Prompt con RAG. Il modello riceve esclusivamente:

- la finestra corrente di comandi,
- può ricevere un insieme di vincoli rigidi a seconda della funzione di prompting scelta, con o senza whitelist (**whitelist di comandi**, file e directory),
- l'obbligo di generare esattamente N comandi.

```
def make_prompt_topk_whitelist(context: List[str], k: int) → str:
    ctx = "\n".join(context[-10:])
    return f"""
You are an AI simulating a cyber-attacker inside an SSH honeypot.
Your task is to predict the EXACT next command the attacker will type.

INSTRUCTIONS:
1. Analyze the 'CURRENT SESSION' below.
2. Output the {k} most likely next commands.
3. Command can ONLY be a combination of commands from the WHITELIST, combining if necessary with files present in
WHITELISTFILES or folders present in WHITELISTFOLDERS. The whitelists are below.
4. Commands can be constructed using pipelines (linux command '|')
5. Commands can present redirections ('>' or '>>') when the target is a whitelisted file or a file inside a
whitelisted folder
6. Output ONLY raw commands, one per line. No explanations.

CURRENT SESSION HISTORY:
{ctx}

WHITELIST (containing commands):
{_whitelist_commands}

WHITELISTFILES (containing critics files that can be used with previous commands):
{_whitelist_files}

WHITELISTFOLDERS (containing critics folders that can be used with previous commands):
{_whitelist_folders}

PREDICT NEXT {k} COMMANDS (Raw text only):
""".strip()
```

Figure 3: Funzione `make_prompt_topk_whitelist()`

```python
def make_prompt_topk_without_whitelist(context: List[str], k: int) -> str:
    ctx = "\n".join(context[-10:])
    return f"""
You are an AI simulating a cyber-attacker inside an SSH honeypot.
Your task is to predict the EXACT next command the attacker will type.

INSTRUCTIONS:
1. Analyze the 'CURRENT SESSION' below.
2. Output the {k} most likely next commands.
3. Output ONLY raw commands, one per line. No explanations.

CURRENT SESSION HISTORY:
{ctx}

PREDICT NEXT {k} COMMANDS (Raw text only):
""".strip()
```

Figure 4: Funzione `make_prompt_topk_without_whitelist()`

La struttura dei prompt rimane comunque **uniforme**, così da garantire comparabilità tra le varianti.

4.2.2 Retrieval-Augmented Generation (RAG)

Nelle configurazioni che sfruttano il RAG, il contesto locale della sessione viene arricchito con esempi reali estratti da un database vettoriale costruito sulle sessioni di attacco del dataset Cowrie. Il codice fondamentale per eseguire questo approccio è contenuto all'interno dello script `core_rag.py`.

Costruzione DB RAG Per quanto riguarda la costruzione del DB ci si affida alla classe `VectorContextRetriever` che implementa la logica centrale per trasformare le sequenze di comandi in embedding mediante `sentence-transformers` (`all-MiniLM-L6-v2`), memorizzarle in un database vettoriale **ChromaDB**, interrogarlo per recuperare i contesti pi'u simili alla sessione in corso e restituire, insieme alla history dei comandi, anche il *prossimo comando reale* osservato in attacchi precedenti.

Questa componente permette di fornire al modello LLM esempi pertinenti, migliorando la capacità di predire il prossimo comando di un attaccante. La classe offre diverse funzionalità rilevanti:

- `index_file` — costruisce il database vettoriale generando finestre scorrevoli di tipo *contesto + next_command*, evitando duplicazioni e includendo un meccanismo di recovery in caso di interruzione durante l'indicizzazione;
- `retrieve` — dato un contesto corrente, ricerca i vettori più simili e restituisce gli esempi strutturati da includere nel prompt RAG, nonchè il prossimo comando inserito dall'attaccante nei contesti simili restituiti.

```python
class VectorContextRetriever:
    # Inizializzazione RAG DB
    def __init__(self, persist_dir: str, collection_name="honeypot_attacks"):
        print(f"── Inizializzazione RAG DB ({persist_dir}) ──")

        # Creazione client che gestisce un vector database ChromaDB, database contenente embeddings
        self.client = chromadb.PersistentClient(path=persist_dir)
        # Modello di embedding utile per eseguire ricerca all'interno di un db in quanto veloce e leggero → ogni
vettore è costituito da 384 elementi
        self.emb_fn = embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-v2")
        # Creazione della tabella honeypot_attacks (parametro passato) all'interno del DB
        self.collection =
self.client.get_or_create_collection(name=collection_name,embedding_function=self.emb_fn)
```

Figure 5: Estratto del modulo di indexing e retrieval utilizzato per RAG.

```python
def load_seen_vectors(self):
        seen = set()

        all_docs = self.collection.get(
            include=["documents", "metadatas"]  # document = context; metadata["next_command"]
        )

        for context, meta in zip(all_docs["documents"], all_docs["metadatas"]):
            key = (context, meta["next_command"])
            seen.add(key)

        return seen
```

Figure 6: Funzione **load_seen_vectors()**

Costruzione del Prompt con RAG La generazione del prompt è affidata alla funzione `make_rag_prompt()` ed è strutturata come segue:

- prompt con istruzioni precise da rispettare per LLM
- il contesto (inteso come sequenza di comandi) precedenti al comando di cui bisogna eseguire la predizione,
- gli esempi di attacchi simili recuperati dal DB vettoriale,

L'inclusione degli esempi riduce le hallucination e rafforza la coerenza con i pattern storici.

```python
def make_rag_prompt(context_list: List[str], rag_text: str, k: int) -> str:
    current_history = "\n".join(context_list[-10:])
    return f"""
You are an AI simulating a cyber-attacker inside an SSH honeypot.
Your task is to predict the EXACT next command the attacker will type.

INSTRUCTIONS:
1. Analyze the 'CURRENT SESSION' below.
2. Look at the 'SIMILAR PAST ATTACKS' provided (Retrieval Augmented Generation) to understand attacker patterns.
3. Output the {k} most likely next commands.
4. Output ONLY raw commands, one per line. No explanations.

================================================
{rag_text}
================================================

CURRENT SESSION HISTORY:
{current_history}

PREDICT NEXT {k} COMMANDS (Raw text only):
""".strip()
```

Figure 7: Funzione `make_rag_prompt()`.

4.2.3 Funzioni di Chiamata ai Modelli

Ogni script di valutazione utilizza funzioni wrapper che gestiscono la comunicazione con il modello:

- `query_gemini()` — previa creazione di un Client inizializzato con le chiavi per poter usufruire delle API, invia il prompt a un modello Gemini (di default `gemini-flash-latest`), con i filtri di sicurezza disattivati;
- `query_ollama()` — interagisce con un modello locale tramite le API REST di Ollama.

```
def query_gemini(prompt):
    TRY:
        response = client_gemini.models.generate_content(
    model=model_name,
    contents=prompt,
    config={
        "temperature": temp,
        "top_p": 0.1,
        "max_output_tokens": 1024,
        "safety_settings": safety_config
    }
)
        IF response IS EMPTY:
            RETURN ""
        RETURN response.text

    EXCEPT error:
        IF model_not_found(error):
            EXIT
        RETURN ""
```

Figure 8: Funzione `query_gemini()`.

```
def query_ollama(prompt: str, model: str, url: str, temp: float = 0.0, timeout: int=120) -> str:
    payload = {"model": model, "prompt": prompt, "stream": False, "temperature": temp, "options": {"top_p": 0.1}}

    try:
        response = requests.post(url, json=payload, timeout=timeout)
        response.raise_for_status()
        text = response.json().get("response", "")

        return text.strip()

    except Exception as exc:
        print(f"[OLLAMA ERROR] {exc}")
        return ""
```

Figure 9: Funzione `query_ollama()`.

4.2.4 Pipeline di Valutazione

Gli script:

- `evaluate_gemini_topk.py`,
- `evaluate_gemini_rag.py`,
- `evaluate_ollama_topk.py`,
- `evaluate_ollama_rag.py`

oltre a contenere le funzione wrapped per comunicare con i rispettivi LLM, rappresentano gli script eseguibili utilizzati per la fase di valutazione. All'interno dei main contengono una chiamata alla funzione `prediction_evaluation()` implementata nelle rispettive librerie (`core_rag.py`

o `core_topk.py`) a seconda dello script considerato.

La funzione `prediction_evaluation()` è implementata in due varianti distinte: una dedicata alla modalità **RAG** e una alla modalità **Top-*k***. Entrambe attraversano l'intero dataset di test e, per le sessione selezionate, ricostruiscono il contesto operativo dell'attaccante.

Nella versione **RAG**, il contesto locale viene arricchito con esempi recuperati dal database vettoriale, e il prompt viene generato utilizzando le informazioni fornite dal modulo di retrieval. Nella versione **Top-*k***, invece, il prompt è costruito senza consultare il database vettoriale e si basa esclusivamente sul contesto fornito e sulla eventuale presenza di **whitelist**.

In entrambi i casi, il prompt risultante viene inviato al modello selezionato (Gemini o CodeLlama). Una volta ottenuta la risposta, la funzione confronta i comandi generati con quello realmente eseguito dall'attaccante, così da valutare la correttezza della predizione. Tutti i risultati vengono infine salvati in un file `JSONL`, che raccoglie sia le predizioni sia le informazioni necessarie al calcolo delle metriche finali.

```
Fase di valutazione dei modelli LLM
---

1. Caricamento del dataset di test (JSONL)

2. Per ogni sessione:
   - estrazione del contesto (ultimi k comandi)

3. Se RAG attivo:
   - query al database ChromaDB
   - recupero dei r esempi più simili
   - costruzione del blocco "Similar Past Attacks"

4. Costruzione del prompt:
   - make_rag_prompt()   (modalità RAG)
   - make_prompt_topk()  (modalità standard)

5. Chiamata al modello LLM:
   - Gemini (API)
   - CodeLlama via Ollama

6. Parsing della risposta:
   - filtraggio comandi
   - normalizzazione
   - estrazione dei Top-k

7. Confronto con il comando reale successivo:
   - calcolo Top-1 accuracy
   - calcolo Top-k accuracy

8. Salvataggio dei risultati:
   - file JSONL dettagliato

Fine processo
```

Figure 10: Schema semplificato della funzione `prediction_evaluation()`.

5 Risultati

L'adozione delle quattro varianti predittive (**CodeLlama**, **CodeLlama + RAG**, **Gemini**, **Gemini + RAG**) consente confronti diretti tra modelli **open-source** e **closed-source**, permettendo di valutare:

- l'impatto del RAG sulla qualità delle predizioni;
- il comportamento del sistema in scenari operativi reali.

Questa architettura rende il framework flessibile, sperimentale e facilmente estendibile.

In questa sezione presentiamo i risultati sperimentali ottenuti dalle quattro varianti del modulo predittivo, valutate con un contesto di lunghezza 3 o 5.

Le valutazioni sono state eseguite utilizzando gli script `evaluate_gemini_rag.py`, `evaluate_gemini_topk.py` , `evaluate_ollama_rag.py`e `evaluate_ollama_topk.py`, che misurano in modo sistematico la capacità dei modelli di generare predizioni Top-k corrette sulla base del contesto della sessione e, quando presente, degli esempi recuperati tramite RAG.

5.1 Risultati del Modello Gemini con RAG

In questa sezione vengono presentati i risultati ottenuti dall'esecuzione dello script `evaluate_gemini_rag.py`, che applica il modello Gemini integrato con un approccio RAG (Retrieval-Augmented Generation).

La Figura 11 mostra l'output ottenuto utilizzando una **finestra di contesto pari a 5** (5 comandi precedenti come input del modello) con $k = 5$ possibili predizioni per il prossimo comando. L'esperimento è stato condotto su un insieme di 100 task estratti dal dataset di test. Oltre all'accuratezza Top-1 e Top-5, viene riportato il numero di predizioni influenzate direttamente dal RAG, evidenziando quanto il recupero del contesto storico contribuisca alla qualità del risultato.

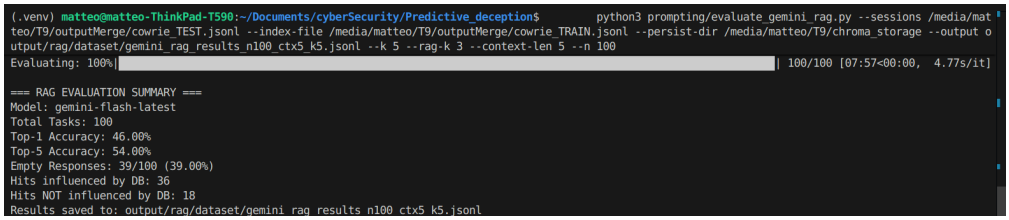

Figure 11: Esempio di output dello script `evaluate_gemini_rag.py` con finestra di contesto pari a 5.

La Figura 12 presenta invece un secondo esperimento, questa volta utilizzando una **finestra di contesto pari a 3**. Il valore più ridotto fornisce al modello meno informazioni sul comportamento passato dell'attaccante, rendendo la predizione più complessa e permettendo di osservare la sensibilità del sistema alla lunghezza del contesto.

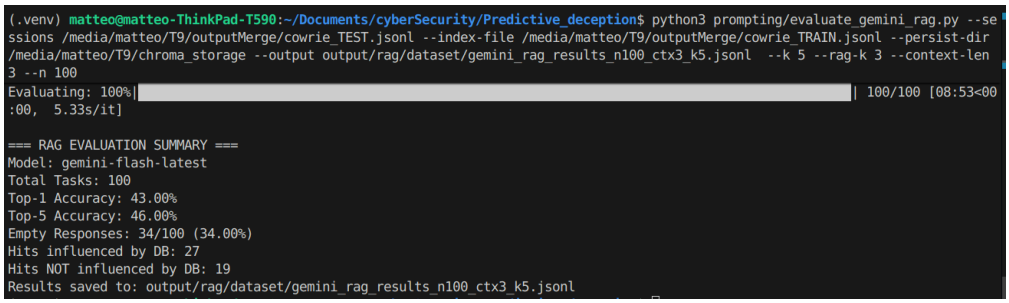

Figure 12: Esempio di output dello script `evaluate_gemini_rag.py` con finestra di contesto pari a 3.

Per l'esecuzione di questo esperimento sono stati utilizzati due differenti indicizzazioni del DB, in quanto gli embedding in esso contenuti sono fortemente influenzati dalla lunghezza del contesto.

Dal confronto tra i due esperimenti emerge come una finestra di contesto maggiore tenda a favorire il modello nella ricostruzione della strategia dell'attaccante, aumentando la probabilità di individuare il comando successivo corretto. Tuttavia, anche con una finestra di contesto limitata, il modello beneficia del supporto del RAG, che permette di richiamare sessioni simili registrate nel database vettoriale, migliorando la qualità delle predizioni anche in condizioni di informazione ridotta.

5.2 Risultati del Modello Gemini senza RAG

In questa sezione vengono presentati i risultati ottenuti dallo script `evaluate_gemini_topk.py`, utilizzato per valutare il modello Gemini in assenza del supporto RAG.

La Figura 13 mostra l'esecuzione del test con *whitelist* attiva, che vincola il modello a selezionare le predizioni tramite insiemi predefiniti di comandi, file o cartelle critiche, precedentemente forniti al sistema. Vengono riportati i valori di accuratezza Top-1, Top-5 e il numero di predizioni vuote su un totale di 100 task valutati.

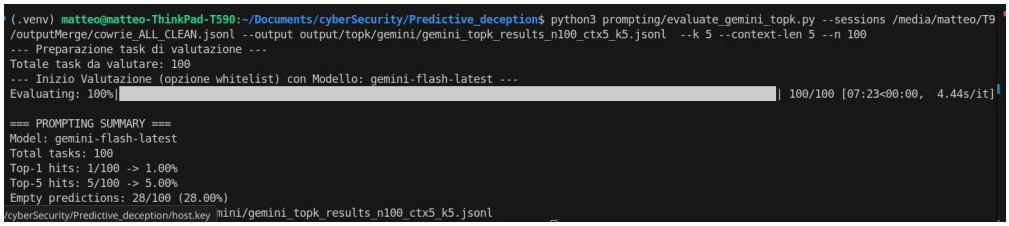

Figure 13: Esempio di output dello script `evaluate_gemini_topk.py` con utilizzo di whitelist.

La Figura 14 presenta invece il risultato dell'esperimento senza *whitelist*, consentendo al modello di generare liberamente qualsiasi comando. Questa modalità produce predizioni più varie ma anche più rumorose rispetto alla versione vincolata. Tuttavia, nel caso specifico del modello Gemini, la modalità con *whitelist* non migliora le prestazioni come atteso: anzi, risulta leggermente peggiore.

Questo comportamento è dovuto al fatto che la *whitelist* introduce un prompt più rigido e strutturato, che entra in conflitto con alcune regole fondamentali del prompt engineering (ad esempio: evitare istruzioni lunghe, evitare regole negative, evitare elenchi densi all'interno di un singolo blocco di istruzioni). Il risultato è un degradamento della qualità del completamento, con un numero maggiore di predizioni vuote o fuori formato. Al contrario, la modalità senza vincoli lascia il modello più libero di seguire i pattern dei comandi osservati nel dataset, ottenendo nel complesso risultati più stabili e coerenti.

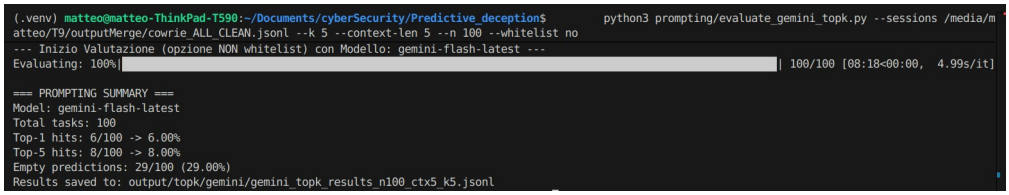

Figure 14: Esempio di output dello script `evaluate_gemini_topk.py` senza utilizzo di whitelist.

5.3 Risultati del Modello Codellama con RAG

In questa sezione vengono presentati i risultati ottenuti eseguendo lo script `evaluate_ollama_rag.py`, che valuta le prestazioni del modello Codellama integrato con la pipeline RAG (Retrieval-Augmented Generation).

La Figura 15 mostra i risultati ottenuti utilizzando una finestra di contesto pari a 5. In questo scenario il modello dispone di più informazioni sul comportamento recente dell'attaccante, con un conseguente miglioramento dell'accuratezza Top-1 e Top-5. Inoltre viene indicato il numero di predizioni influenzate dal RAG, evidenziando il contributo del recupero storico alla generazione del comando successivo.

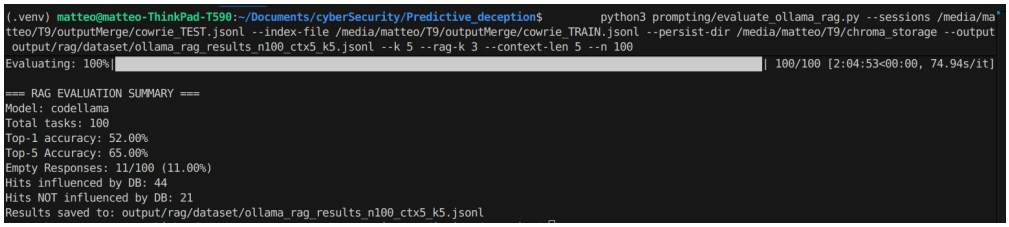

Figure 15: Esempio di output dello script `evaluate_ollama_rag.py` con finestra di contesto pari a 5.

La Figura 16 presenta invece la valutazione con una finestra di contesto pari a 3. In questo caso il modello ha a disposizione un numero ridotto di comandi recenti, rendendo la predizione più complessa. Il confronto con la configurazione precedente consente di osservare chiaramente la sensibilità del modello alla lunghezza del contesto fornito.

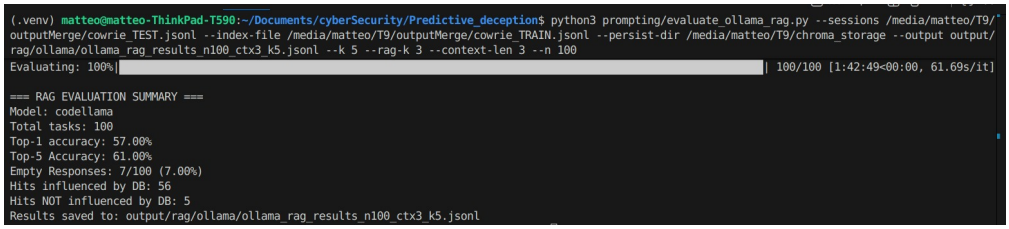

Figure 16: Esempio di output dello script `evaluate_ollama_rag.py` con finestra di contesto pari a 3.

5.4 Risultati del Modello Codellama senza RAG

In questa sezione vengono presentati i risultati ottenuti eseguendo lo script `evaluate_ollama_topk.py`, che valuta le prestazioni del modello Codellama in assenza di qualunque supporto da parte del sistema RAG.

La Figura 18 riporta invece i risultati ottenuti disattivando la whitelist. In questa configurazione il modello è libero di generare qualunque comando, rendendo la predizione più difficile e aumentando il rischio di output non pertinenti.

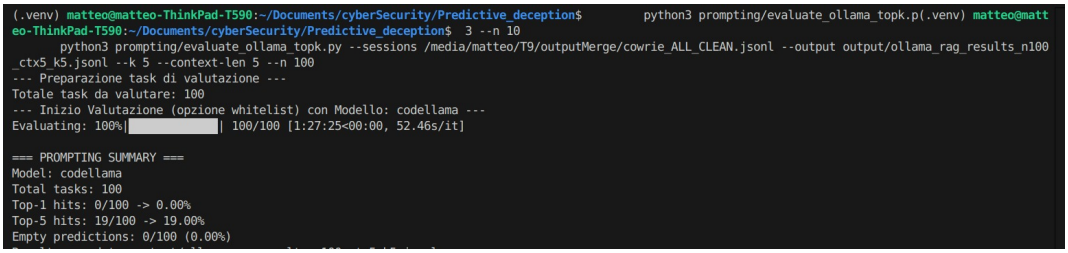

Figure 17: Esempio di output dello script `evaluate_ollama_topk.py` con utilizzo di whitelist.

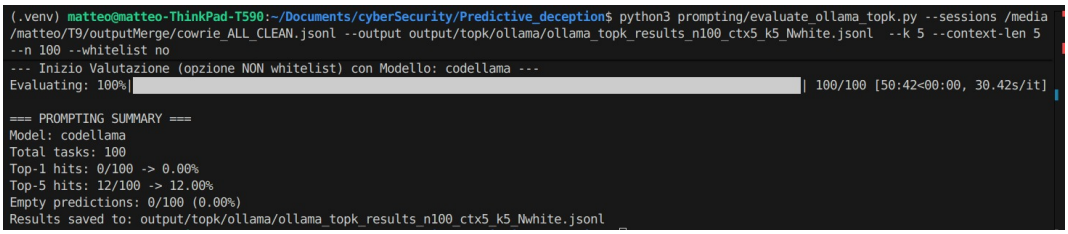

Figure 18: Esempio di output dello script `evaluate_ollama_topk.py` senza utilizzo di whitelist.

Come è possibile osservare dalle due figure, nel caso specifico di Codellama, i risultati con whitelist hanno ottenuto l'effetto sperato, portando ad un aumento della percentuale di prediction.

5.5 Confronto complessivo tra le Varianti

In questa sezione vengono confrontati in modo organico tutti gli esperimenti condotti sulle quattro varianti del modulo predittivo. Sono state eseguite complessivamente otto configurazioni distinte, riassunte come segue:

- **CodeLlama + RAG**: test con finestra di contesto pari a 3 e 5;
- **Gemini + RAG**: test con finestra di contesto pari a 3 e 5;
- **CodeLlama senza RAG (Top-k)**: valutazione con whitelist attiva e disattiva, entrambe con contesto 5;
- **Gemini senza RAG (Top-k)**: valutazione con whitelist attiva e disattiva, entrambe con contesto 5.

Per ciascuna variante sono stati misurati Top-1, Top-5, il numero di predizioni vuote e, quando applicabile, l'influenza del RAG sul risultato. La Tabella 1 riassume in modo compatto le diverse configurazioni, includendo la finestra di contesto effettivamente utilizzata per ciascun esperimento.

Table 1: Confronto sintetico delle otto configurazioni testate.

Modello / Configurazione	**RAG**	**Whitelist**	**Context**
CodeLlama + RAG	Sì	–	3
CodeLlama + RAG	Sì	–	5
Gemini + RAG	Sì	–	3
Gemini + RAG	Sì	–	5
CodeLlama Top-k	No	Sì	5
CodeLlama Top-k	No	No	5
Gemini Top-k	No	Sì	5
Gemini Top-k	No	No	5

Dall'analisi congiunta, riportata nella Tabella 2, emerge un quadro chiaro. Le configurazioni basate su RAG offrono prestazioni superiori in maniera consistente: sia Gemini sia CodeLlama traggono beneficio dal recupero di esempi storici, con incrementi evidenti delle predizioni nella Top-5.

Le modalità senza RAG presentano prestazioni sensibilmente inferiori e risultano più sensibili alla variabilità del dataset, con una maggiore tendenza alla generazione di comandi non pertinenti. L'uso della whitelist mitiga parzialmente questo comportamento, migliorando la qualità delle predizioni pur mantenendo una capacità predittiva limitata rispetto alle varianti con RAG.

Nel complesso, valutando sia l'efficacia della prediction che il tempo effettivo per eseguirla, Gemini + RAG rappresenta complessivamente la soluzione migliore in quanto, seppur diminuiscano le prediction di un fattore del circa 10% rispetto a CodeLlama + RAG, il tempo per eseguire le prediction è dimezzato rispetto al suddetto modello.

Table 2: Confronto prestazionale tra tutte le varianti testate (Top-1, Top-5, predizioni vuote e influenza RAG).

Configurazione	**Top-1 (%)**	**Top-5 (%)**	**Empty (%)**	**RAG Influence**
CodeLlama + RAG (ctx = 3)	57%	61%	7%	56
CodeLlama + RAG (ctx = 5)	52%	65%	11%	44
Gemini + RAG (ctx = 3)	43%	46%	34%	27
Gemini + RAG (ctx = 5)	46%	54%	39%	36
CodeLlama Top-k WL (ctx = 5)	0%	19%	0%	–
CodeLlama Top-k NWL (ctx = 5)	0%	12%	0%	–
Gemini Top-k WL (ctx = 5)	1%	5%	28%	–
Gemini Top-k NWL (ctx = 5)	6%	8%	29%	–

Come mostrato nella Tabella 2, le differenze prestazionali tra le otto varianti testate sono significative e mettono in evidenza sia l'effetto del RAG sia l'impatto della lunghezza del contesto.

- **CodeLlama + RAG (ctx = 3)** è la configurazione con la *migliore Top-1* assoluta, raggiungendo il 57% di accuratezza e un 61% in Top-5, con solo il 7% di predizioni vuote. Il valore di *RAG Influence* pari a 56 indica che, in più della metà dei task, il recupero di esempi storici ha contribuito direttamente alla predizione.
- **CodeLlama + RAG (ctx = 5)** ottiene la *migliore Top-5* in assoluto (65%), con una Top-1 comunque elevata (52%) e una percentuale di predizioni vuote contenuta (11%). Rispetto al contesto 3, il modello sfrutta meglio l'informazione aggiuntiva per individuare il comando corretto entro le prime 5 posizioni, pur risultando leggermente meno preciso sulla Top-1 (condizione dipendente dal comando da predirre).
- **Gemini + RAG (ctx = 3)** mostra prestazioni più basse rispetto a CodeLlama + RAG: 43% Top-1, 46% Top-5 e un valore di Empty relativamente elevato (34%). L'influenza del RAG (27) è inferiore a quella osservata per CodeLlama, suggerendo un minore sfruttamento degli esempi storici recuperati.
- **Gemini + RAG (ctx = 5)** migliora leggermente la Top-1 (46%) e la Top-5 (54%) rispetto al contesto 3, ma a costo di un ulteriore aumento delle predizioni vuote (39%). Anche in questo caso il valore di *RAG Influence* (36) rimane inferiore a quello di CodeLlama, indicando un beneficio meno marcato dal retrieval.

Per quanto riguarda le configurazioni **senza RAG**, i limiti sono ancora più evidenti:

- **CodeLlama Top-k (WL e NWL, ctx = 5)** mostra una Top-1 pari a 0% in entrambi i casi e una Top-5 molto bassa (19% con whitelist, 12% senza). Non vengono generate predizioni vuote, ma la capacità predittiva è estremamente limitata: il modello fatica a individuare il comando corretto anche all'interno delle prime 5 scelte.
- **Gemini Top-k WL (ctx = 5)** raggiunge solo l'1% di Top-1 e il 5% in Top-5, con una percentuale di Empty molto alta (28%). La whitelist vincola il modello a un set di comandi predefiniti, ma introduce un comportamento conservativo, con molte risposte vuote o non risolutive.
- **Gemini Top-k NWL (ctx = 5)** migliora leggermente (6% Top-1, 8% Top-5), ma mantiene una quota elevata di predizioni vuote (29%). L'assenza di whitelist permette più esplorazione, al prezzo di output più rumorosi e spesso non pertinenti.

Nel complesso, la tabella conferma che:

- le configurazioni **CodeLlama + RAG** sono le più robuste, con il miglior equilibrio tra Top-1, Top-5, pochi Empty e forte *RAG Influence*, al costo di una valutazione decisamente più lenta;
- **Gemini + RAG** beneficia del retrieval ma rimane meno efficace, seppur accettabile, rispetto al precedente, anche a causa dell'elevato numero di predizioni vuote;
- tutte le varianti **senza RAG** mostrano prestazioni nettamente inferiori: l'assenza di memoria storica degli attacchi riduce drasticamente la capacità di seguire pattern reali;
- la **whitelist** aiuta a limitare output completamente fuori distribuzione, ma da sola non è sufficiente a compensare la mancanza di contesto strutturale fornito dal RAG e in certi casi può appensantire, e non poco, il prompt e di conseguenza anche la risposta del modello.

Questi risultati evidenziano come il recupero di esempi storici tramite RAG sia il fattore più determinante per il miglioramento delle prestazioni, in particolare per CodeLlama, che passa da prestazioni quasi nulle in modalità Top-k a valori competitivi e utilizzabili in uno scenario di deception operativa.

6 Ambiente di Deception Basato su FakeShell e Defender Runtime

In questa sezione descriviamo in dettaglio l'architettura di deception implementata all'interno dell'ambiente virtualizzato Vagrant, basata su due componenti principali: una **FakeShell** che simula un terminale Bash realistico e un **Defender Runtime** che osserva il comportamento dell'attaccante, predice i comandi successivi tramite un modello Gemini integrato con RAG (Retrieval-Augmented Generation) e genera dinamicamente artefatti ingannevoli nel file system reale.

6.1 Ambiente di Esecuzione: Macchina virtuale Vagrant

L'intera infrastruttura di test è ospitata all'interno di una macchina virtuale Vagrant, il cui provisioning è stato eseguito tramite task Ansible, che fornisce:

- un ambiente Linux isolato e riproducibile;
- provisioning automatico di pacchetti e dipendenze tramite Ansible;

All'interno della VM sono stati creati due utenti distinti (`vagrant`-utente default e `user`), utilizzati per simulare sessioni di attacco multiple. L'accesso avviene tramite SSH con chiave privata, configurando Vagrant per esporre la porta di servizio (tipicamente la porta 22 interna mappata su porta 2222 della macchina host).

Una volta stabilita la connessione SSH con utente `user` verso la VM, all'ipotetico attaccante si palesa difronte una fakeshell (script python che sostituisce la shell di default /bin/bash), che ha il normale comportamento di una shell bash con il vantaggio di una cattura più facile dei comandi inseriti (loggati all'interno del file /var/log/fakeshell.json). In parallelo, dall'utente vagrant viene avviato lo script *defender.py* che monitora continuamente il log JSON generato dalla shell fittizia.

6.2 FakeShell: Shell Realistica con Logging

fakeshell.py è uno script Python che simula un terminale Bash ad alta interazione, ma con un comportamento completamente controllato. Lo script presenta all'attaccante un prompt realistico del tipo:

`user@hostname:/path/current$`

utilizzando le informazioni reali di utente e hostname ottenute tramite le librerie standard di sistema.

La FakeShell implementa le seguenti funzionalità principali:

- **prompt colorato e realista**, con indicazione di utente, hostname e current working directory (CWD);
- **esecuzione dei comandi reali** lo script cattura ogni comando inserito e tramite `/bin/bash` esegue il comando, in un terminale interattivo, così da supportare anche comandi che richiedono interazione con l'utente;
- **logging JSON** di ogni comando inserito, con metadati contestuali.

Ogni comando digitato viene registrato come una linea JSON nel file `/var/log/fakeshell.json`. La struttura del log include:

- `timestamp`: istante di esecuzione del comando;
- `ip`: indirizzo IP associato alla sessione (di default settato a `192.168.1.150`);
- `user`: utente che simula l'attaccante;
- `cwd`: directory corrente al momento dell'esecuzione;
- `cmd`: stringa del comando inserito.

In termini di implementazione, la funzione `log_command` si occupa di serializzare una entry JSON per ogni comando:

```python
def log_command(cmd, cwd):
    entry = {
        "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
        "ip": ip,
        "user": user,
        "cwd": cwd,
        "cmd": cmd
    }
    with open(LOG_FILE, "a", encoding="utf-8") as f:
        f.write(json.dumps(entry) + "\n")
```

Figure 19: Funzione log_command()

Il ciclo principale della FakeShell costruisce il prompt, legge il comando da input, applica eventuali alias, invoca `log_command` e, infine, esegue il comando tramite un processo figlio associato ad un PTY:

```python
while True:
    symbol = "#" if user == "root" else "$"
    prompt = f"\033[1;32m{user}@{hostname}\033[0m:\033[1;34m{cwd}\033[0m{symbol} "
    cmd = input(prompt)
    if not cmd.strip():
        continue
    log_command(cmd, cwd)
    # gestione cd ...
    # esecuzione comando via pty.fork() + os.execve("/bin/bash", ["bash", "-c", cmd], env)
```

Figure 20: Loop principale della FakeShell.

In questo modo il sistema raccoglie una traccia dettagliata e contestualizzata del comportamento dell'attaccante.

6.3 Defender Runtime: Predizione RAG e Generazione di Artefatti

Il *Defender Runtime* è uno script Python che implementa la logica difensiva. Il suo compito è:

1. seguire in tempo reale il file di log JSON generato dalla FakeShell;
2. mantenere una storia dei comandi per ogni sessione;
3. utilizzare un modello Gemini con approccio RAG per predirre i prossimi comandi plausibili;
4. per ogni comando predetto, generare una descrizione di difesa e un artefatto (file) da collocare in percorsi realistici;
5. materializzare questi artefatti nel file system reale (ad es. sotto `/home/user`, `/var/log`, `/etc/...`);
6. quando l'attaccante inserisce il comando successivo, mantenere solo la branch coerente e ripulire gli artefatti legati alle predizioni sbagliate.

Ciclo di ascolto e parsing del log. Il Defender implementa una funzione `follow_log` che replica il comportamento di un `tail -f` sul file `/var/log/fakeshell.json`. Per ogni riga JSON valida:

- viene ricostruita una entry Python;
- si invoca `handle_new_command(entry)` per la entry appena costruita.

```python
def follow_log(path: str):
    while not os.path.exists(path):
        print(f"[WAIT] In attesa che esista il file di log: {path}")
        time.sleep(1)
    with open(path, "r", encoding="utf-8") as f:
        f.seek(0, os.SEEK_END)
        while True:
            line = f.readline()
            if not line:
                time.sleep(0.1)
                continue
            try:
                entry = json.loads(line.strip())
            except json.JSONDecodeError:
                continue
            yield entry
```

(a) Loop principale del Defender Runtime.

```python
def main():
    load_commands_state()
    load_active_artifacts()
    print("[*] Defender runtime attivo.")
    print("[*] Ascolto log:", HONEYPOT_LOG)
    for entry in follow_log(HONEYPOT_LOG):
        handle_new_command(entry)
```

(b) Flusso complessivo di deception tra FakeShell, log e Defender.

Figure 21: Panoramica del funzionamento del Defender Runtime e dell'architettura di deception.

Prompting con Gemini e RAG. Il Defender carica inizialmente la chiave API di Gemini da un file `.env` e configura un client `google.genai`. Inoltre apre un database vettoriale ChromaDB precedentemente indicizzato nel path `/home/vagrant/chroma_storage_ctx5`. La classe `VectorContextRetriever`, uguale alla precedente utilizzata per le predizioni ma adattata estrapolando solo le funzioni necessarie (quelle di inizializzazione e di retrieve del Db vettoriale) si occupa di:

- connettersi alla collection pre–esistente (`honeypot_attacks`);
- effettuare query di similarità sui vettori associati alle sequenze di comandi storici, restituendo i contesti di attacco passati più simili a quello passato, formattati come esempi da iniettare nel prompt (sezione RAG).

Sulla base della history dei comandi per una data sessione, il Defender costruisce il prompt RAG tramite la funzione `make_rag_prompt`, che inserisce sia la *CURRENT SESSION HISTORY* sia la sezione *SIMILAR PAST ATTACKS*. Il risultato è passato a `query_gemini`, che restituisce una lista di comandi predetti (Top-k).

Predizione e generazione delle difese. La funzione `handle_new_command` coordina l'intero flusso di difesa:

1. calcola la *session key* dalla entry di log;
2. aggiorna la history dei comandi inseriti nella sessione;
3. invoca la funzione per predirre i prossimi comandi tramite `predict_next_commands()`;
4. per ciascuna predizione invoca la funzione `plan_and_apply_defenses`.

```python
def handle_new_command(entry: Dict[str, Any]):
    session_key = make_session_key(entry)
    cmd = entry.get("cmd", "").strip()
    if not cmd:
        return
    print(f"[{datetime.now().isoformat()}] session={session_key} cmd={cmd}")
    cleanup_other_branches(session_key, cmd)
    update_history(session_key, cmd)
    predictions = predict_next_commands(session_key)
    print(f"   → Predizioni: {predictions}")
    plan_and_apply_defenses(session_key, predictions)
```

Figure 22: Gestione di un nuovo comando nel Defender.

La generazione delle difese avviene in due fasi:

- **progettazione** (fase "di pensiero" LLM): la funzione `create_defense_for_predicted_command()` costruisce un prompt che vincola Gemini a restituire esclusivamente un oggetto JSON rappresentante l'artefatto con i campi: `description`, `intended_path` e `content`; Se l'artefatto relativo al comando predetto è già catalogato nel file `defenses_index.json`, non viene inviata nessuna richiesta di generazione al modello, ma viene semplicemente riutilizzato l'artefatto precedentemente prodotto.
- **materializzazione** (fase "operative"): la funzione `materialize_defense_artifacts()` prende `intended_path` e il contenuto dell'artefatto e crea fisicamente il file nel file system reale, usando `sudo tee` per scrivere in percorsi protetti.

```python
def create_defense_for_predicted_command(command: str, session_key: str) -> Dict[str, Any]:
    cmd_safe = command.replace("%", "%%")
    prompt = f"""
    You must output ONLY a JSON object.
    {{
      "description": "short description of the defense",
      "intended_path": "/realistic/system/path/that/an/attacker/would_expect",
      "artifacts": [
        {{
          "path": "defense_artifacts/<SAFE_FILENAME>",
          "content": "<FILE CONTENT>"
        }}
      ]
    }}
    Generate JSON for predicted command: "{cmd_safe}".
    """
    raw = query_gemini(prompt, model_name=GEMINI_MODEL)
    try:
        defense = json.loads(raw)
    except:
        defense = { ... fallback ... }
    # fix intended_path e artifacts ...
    return defense
```

Figure 23: Creazione di una difesa per un comando predetto.

```python
def materialize_defense_artifacts(defense: Dict[str, Any], session_key: str, predicted_command: str) -> List[str]:
    paths: List[str] = []
    intended_real_path = defense.get("intended_path")
    if not intended_real_path:
        return paths
    arts = defense.get("artifacts", [])
    if not arts:
        return paths
    art = arts[0]
    content = art.get("content", "")
    real_path = intended_real_path
    os.makedirs(os.path.dirname(real_path), exist_ok=True)
    proc = subprocess.run(["sudo", "tee", real_path],
                          input=content.encode("utf-8"),
                          stdout=subprocess.DEVNULL,
                          stderr=subprocess.DEVNULL)
    if proc.returncode == 0:
        active_artifacts[real_path] = {
            "command": predicted_command,
            "session": session_key,
            "timestamp": int(time.time())
        }
        paths.append(real_path)
    return paths
```

Figure 24: Materializzazione degli artefatti nel file system reale.

Pulizia delle branch alternative. Quando l'attaccante invia il comando successivo, il Defender verifica se questo appartiene all'insieme delle predizioni correnti. In caso affermativo, mantiene soltanto gli artefatti associati al comando effettivamente scelto e rimuove:

- i file creati per le predizioni errate (via `sudo rm -f`);
- le entry corrispondenti nel file runtime `active_artifacts.json`, file json che contiene gli artefatti attualmente posizionati all'interno del filesystem.

In questo modo la deception rimane coerente con la "timeline" reale della sessione, evitando l'accumulo di file inutili o potenzialmente incoerenti.

6.4 Sintesi del Flusso di Deception

Combinando i due script si ottiene un sistema di deception dinamico con il seguente flusso:

1. l'attaccante si connette via SSH alla VM, loggandosi come utente `user` e viene posto davanti alla FakeShell;
2. ogni comando inserito viene eseguito realmente dal sistema (via Bash e PTY) e registrato nel log JSON;
3. il Defender monitora in tempo reale il log, aggiorna la storia della sessione e, tramite RAG + Gemini, predice i prossimi comandi;
4. per ogni comando predetto genera una strategia di difesa (artefatto e path realistico) e crea i file corrispondenti nel file system della VM;
5. quando il comando successivo viene effettivamente eseguito, il sistema mantiene solo la branch coerente con quel comando, rimuovendo gli artefatti associati alle altre predizioni.

Questa architettura consente di passare da una deception statica a una **deception guidata da modelli predittivi**, in cui il contenuto ingannevole non è predefinito ma viene generato e mantenuto on–the–fly in funzione del comportamento dell'attaccante e dei pattern appresi dai dataset di honeypot storici.

7 Conclusioni

Il lavoro presentato ha mostrato come l'integrazione di un modello LLM all'interno di un honeypot SSH permetta di passare da una semplice registrazione passiva dei comandi ad una forma di *deception predittiva*. Il modulo predittivo osserva in tempo reale le sequenze di comandi loggate (e.g. nel file `fakeshell.json`), costruisce una finestra di contesto e, nelle varianti con RAG, la arricchisce con esempi storici recuperati da un database vettoriale ChromaDB. Su questa base il modello genera una lista Top-k di possibili comandi successivi. Il defender runtime utilizza tali predizioni per attivare la componente di deception: per i comandi ritenuti più probabili vengono generati artefatti mirati (file di configurazione, log, directory e credenziali fasulle) che vengono effettivamente creati nel filesystem della macchina honeypot, in posizioni realistiche rispetto al comportamento osservato. Quando l'attaccante prosegue l'interazione, il sistema conserva gli artefatti coerenti con il comando realmente eseguito e ripulisce quelli associati alle diramazioni non intraprese, mantenendo il contesto credibile e coerente nel tempo.

Dal punto di vista operativo, questo approccio offre diversi vantaggi: l'honeypot diventa *proattivo*, poiché è in grado di preparare risorse ingannevoli prima che vengano richieste; la qualità dell'intelligence migliora, perché gli artefatti spingono l'attaccante a interagire con elementi che sembrano sensibili, producendo log più ricchi e strutturati; l'uso del RAG riduce le *hallucination* più gravi, ancorando le predizioni a pattern realmente osservati nelle sessioni Cowrie. I risultati sperimentali mostrano come le configurazioni con RAG superino in modo netto le varianti puramente Top-k sia in termini di accuratezza (Top-1/Top-5), sia per coerenza dei comandi generati, confermando che la memoria storica degli attacchi rappresenta un elemento chiave per l'efficacia della deception.

Restano tuttavia alcune criticità. La capacità predittiva dipende dalla copertura del dataset: i comportamenti frequenti vengono modellati bene, mentre attacchi nuovi o poco rappresentati rimangono difficili da anticipare, soprattutto con finestre di contesto molto brevi o nelle fasi finali delle sessioni, dove il comportamento dell'avversario tende a divergere dai pattern tipici. Nelle configurazioni senza RAG aumentano i casi di comandi poco sensati o non validi, che richiedono meccanismi di filtraggio (normalizzazione, whitelist, scarto delle predizioni vuote) per non degradare il realismo dell'ambiente. Inoltre, l'uso di LLM e database vettoriali introduce costi computazionali e, nelle versioni cloud-based, nuove superfici di rischio legate all'esposizione delle API, alla gestione delle chiavi e alla latenza di rete. Anche sul piano della deception, la coerenza degli artefatti generati è cruciale: file o percorsi inverosimili possono insospettire un attaccante esperto e vanificare l'inganno.

Nonostante questi limiti, l'architettura proposta risulta flessibile ed estendibile. Il RAG consente di aggiornare progressivamente la "memoria" del sistema con nuovi log di attacco, mentre il modulo predittivo può essere sostituito o affiancato da altri modelli (locali via Ollama o cloud) senza stravolgere la pipeline. La matrice di deception può evolvere nel tempo introducendo nuovi tipi di artefatti e scenari, man mano che emergono nuove tattiche e strumenti offensivi. In questa prospettiva, la *predictive deception* rappresenta una direzione promettente per gli honeypot SSH: non si limita a osservare l'attaccante, ma ne anticipa i movimenti e li sfrutta per costruire un ambiente controllato, ricco di segnali e di opportunità di osservazione, mantenendo allo stesso tempo un buon compromesso tra realismo, efficacia difensiva e complessità di implementazione.

8 Struttura del Repository

L'intero progetto è organizzato in modo modulare per separare **preprocessing**, **valutazione**, **RAG**, motore di **deception** e **strumenti di supporto**. La struttura della repository, mostrata in Figura 25, riflette un'organizzazione chiara che facilita la manutenzione, la riproducibilità e l'estensione del sistema.

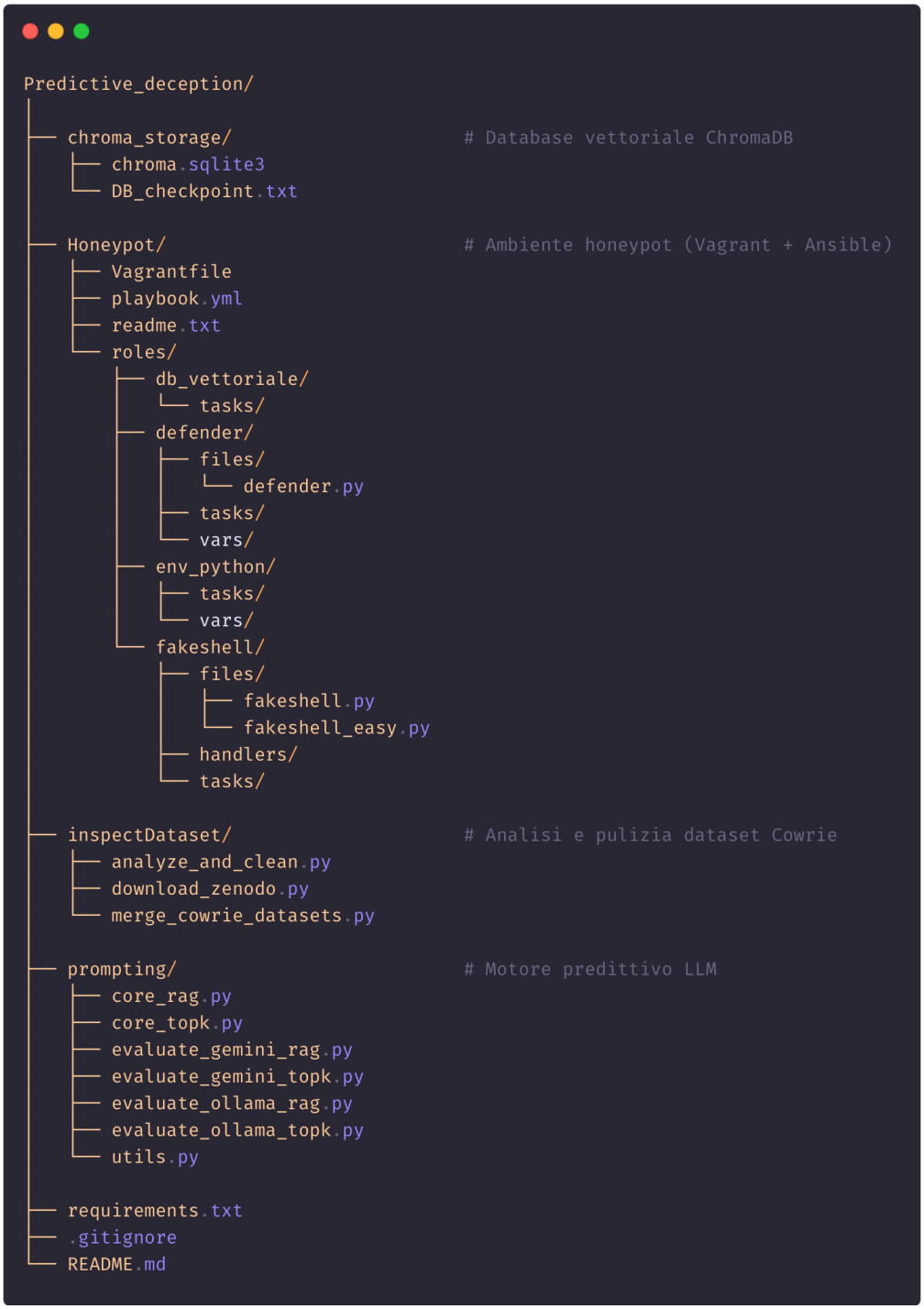

Figure 25: Struttura della repository.

9 Riferimenti e Risorse

9.1 Dataset Pubblici di Honeypot

- CyberLab Honeynet Dataset, `https://zenodo.org/records/3687527`
- PANDAcap SSH Honeypot Dataset, `https://zenodo.org/records/3759652`
- SIHD: Smart Industrial Honeypot Dataset, `https://ieee-dataport.org/documents/sihd-smart-industrial-ho`
- HoneySELK Cyber Attacks Dataset, `https://ieee-dataport.org/open-access/dataset-cyber-attacks-honey`

9.2 Strumenti e Repository

- Cowrie SSH/Telnet Honeypot, `https://github.com/cowrie/cowrie`
- Canarytokens (servizio), `https://canarytokens.org`
- Canarytokens (self-hosted), `https://github.com/thinkst/canarytokens`

9.3 Riferimenti Bibliografici

- Nawrocki, M., et al. (2016). "A Survey on Honeypot Software and Data Analysis." arXiv:1608.06249.
- Deng, G., et al. (2023). "PentestGPT: Evaluating LLMs for Automated Penetration Testing." arXiv:2308.06782.
- Alata, E., et al. (2006). "Lessons Learned from High-Interaction Honeypot Deployment." *EDCC*.
- Whitham, B. (2017). "Canary Tokens and Deception." Thinkst Applied Research.