UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA IN INFORMATICA

TESI DI LAUREA
IN

Analisi dei Pattern delle Chiamate Api per la Classificazione Malware

RELATORI:
Prof.ssa Annalisa Appice
Dr. Giuseppina Andresini

LAUREANDO:
Giacomo Gaudio

ANNO ACCADEMICO 2024 2025

Indice

Disclaimer

Tutti i marchi, nomi commerciali, prodotti e loghi menzionati in questa tesi sono di proprietà dei rispettivi titolari. L'autore non rivendica alcun diritto di proprietà su tali marchi o nomi commerciali e li utilizza solo a scopo informativo e descrittivo, senza alcun intento di violazione.

Sommario

La crescente diffusione delle minacce informatiche e la continua evoluzione delle tecniche di offuscamento adoperate dagli attaccanti hanno reso progressivamente meno efficaci i metodi tradizionali di analisi statica per la rilevazione del software malevolo. Il presente lavoro di tesi affronta tale problematica proponendo un approccio di analisi dinamica basato sul monitoraggio delle sequenze di chiamate API (Application Programming Interface) in ambiente Windows, con l'obiettivo di classificare le diverse famiglie di malware sfruttando tecniche di Machine Learning supervisionato.

La metodologia sviluppata ha previsto la raccolta e la standardizzazione di quattro dataset eterogenei reperiti in letteratura, seguita da una fase di ingegneria delle caratteristiche basata sul modello Bag-of-Words (BoW). Tale approccio ha permesso di trasformare le sequenze di chiamate API in vettori numerici - chiamati *Feature Vector* - basati sulla frequenza delle invocazioni API, ignorando l'ordine temporale.

La validazione empirica è stata condotta confrontando le prestazioni di due algoritmi di classificazione allo stato dell'arte: *RandomForest* e *XGBoost*. I risultati sperimentali evidenziano una maggiore efficacia del classificatore *RandomForest*, che ha dimostrato una superiore capacità di generalizzazione rispetto a *XGBoost*, in particolare nella gestione di dataset sbilanciati e nell'identificazione delle classi minoritarie. Lo studio conferma che l'analisi delle frequenze delle chiamate API costituisce una firma comportamentale valida per distinguere efficacemente tra software legittimo e diverse categorie di malware, ottenendo buone prestazioni di classificazione.

Capitolo 1

Introduzione

Con l'avvento dell'era digitale, attività e servizi, in ambito e-commerce, pubblica amministrazione, servizi finanziari e sanitari, vengono sempre di più digitalizzati in Italia. Secondo i dati ISTAT,l'86,2% delle famiglie italiane dispone di un accesso ad internet a testimonianza di un uso sempre più popolare [1]. Tuttavia, l'aumento della connettività comporta inevitabilmente anche una maggiore esposizione alle minacce informatiche. Nel 2024, l'Italia ha registrato il 10% degli attacchi informatici globali; secondo il rapporto Clusit, circa un terzo di tali attacchi è stato veicolato tramite l'utilizzo di malware [2]. Per malware si intende un istanza di un programma il cui intento sia malevole come carpire informazioni personali e/o arrecare danni ai dispositivi [3]. Per poter identificare i malware esistono due tipi di analisi: statica e dinamica. L'analisi statica prevede l'analisi del codice binario, analizzando ogni ramo di esecuzione possibile alla ricerca di codice malevolo. L'analisi dinamica invece prevede l'esecuzione del programma all'interno di una sandbox, ovvero un ambiente isolato e controllato che impedisce al software di arrecare danni reali al sistema. In questo ambiente è possibile osservare i suoi comportamenti alla ricerca di quelli simili ai malware [4]. Con riferimento alle informazioni veicolate dalle API (Application Programming Interface), per comportamenti intendiamo l'insieme (ordinato) delle chiamate API al Kernel per permettere l'esecuzione dell'applicativo; un API è un modo per interagire con un sistema senza l'ausilio di una interfaccia grafica [5]. Un singolo "comportamento" è effettivamente la singola chiamata API prendendo il nome di "API Call"; mentre la lista (ordinata) prende il nome "API call sequence" [6]. L'obbiettivo di questa tesi è classificare un programma in base ai suoi comportamenti. Per raggiungere tale scopo è stato formalizzato e valutato un approccio che apprende un classificatore, utilizzando sequenze di chiamate API come input e algoritmi di machine learning per distinguere le varie categorie.

1.1 Formato Windows PE malware

Per poter essere eseguito in *Microsoft Windows*, un programma deve presentare una logica applicativa interpretabile dal sistema operativo. A tale scopo, il sistema operativo Windows utilizza il formato `PE` (Portable Executable), così denominato in quanto progettato per non essere vincolato a una specifica architettura [7]. Un file in formato **PE** ha nel seguente ordine le seguenti intestazioni:

1. **Stub MS-DOS**: area iniziale del file PE, compatibile con il sottosistema `MS-DOS`. In assenza di istruzioni specifiche da parte dello sviluppatore, contiene un programma di default che visualizza il messaggio: "Impossibile eseguire questo programma in modalità DOS".

2. **Firma PE**: campo che identifica il file come appartenente al formato **Portable Executable (PE)** e ne consente il corretto riconoscimento da parte del loader di Windows.

3. **Intestazione COFF**: fornisce informazioni fondamentali sul file, come il tipo di macchina su cui è destinato a essere eseguito, il numero di sezioni e la data di compilazione.

4. **Intestazione facoltativa (solo immagine)**: sezione che specifica le informazioni necessarie al caricamento ed esecuzione del programma. È composta da tre sottosezioni principali:

    - **Campi standard intestazione facoltativi** : includono l'indirizzo di entry point del programma, la dimensione del codice e dei dati, e altri parametri di base.
    - **Campi specifici dell'intestazione facoltativa di Windows**: contengono informazioni più dettagliate, come la versione minima del sistema operativo richiesto, le dimensioni massime di heap e stack, e i valori di allineamento della memoria.
    - **Directory dati intestazione facoltative**: riferimenti alle tabelle e alla loro dimensione delle risorse esterne come le ddl.

Una ddl è una libreria che contiene codice e dati utilizzabili da più di un programma contemporaneamente. Ogni programma può utilizzare le ddl del sistema operativo Windows per ottenere memoria, accesso a risorse, far apparire elementi a schermo ed accedere alla rete [8]. I malware utilizzano queste librerie per poter eseguire comportamenti malevoli.

1.2 Categorie Malware

La definizione di *malware* è generica e include tutte le tipologie di software malevolo; per agevolare le attività di rilevamento e classificazione, è utile suddividerli in categorie più specifiche sulla base delle loro finalità e modalità di azione. Nel contesto di questo lavoro di tesi, sono state individuate e considerate le seguenti specializzazioni di software:

- **Unknown**: applicazioni per le quali non è stato possibile determinate con certezza se sono benevoli o meno.
- **Goodware**: software legittimo, privo di finalità malevoli.
- **Malware**: denominazione generica per software malevolo, utilizzata nei casi in cui non sia possibile definire una categoria più specifica.
- **Backdoor**: programma che crea un accesso nascosto al sistema compromesso, permettendo all'attaccante di controllarlo da remoto [9].
- **Trojan**: software che si maschera da applicazione legittima ma che, una volta eseguito, consente operazioni malevole come la modifica o la cancellazione dei dati [10].
- **Virus**: programma in grado di infettare altri file e diffondersi all'interno di un sistema, con lo scopo di danneggiare il corretto funzionamento dell'host [10].
- **Worm**: simili ai virus, ma in grado di diffondersi autonomamente attraverso la rete, senza richiedere l'intervento dell'utente [10].
- **Dropper**: componente malevolo progettato per scaricare o installare altri malware sul sistema target [11].
- **Spyware**: software che raccoglie informazioni sensibili dall'utente o dal sistema infetto e le trasmette a un'entità esterna [10].
- **Adware**: programma che raccoglie informazioni sulle abitudini dell'utente al fine di proporre pubblicità mirata; in alcuni casi può deviare la navigazione verso siti malevoli [10].
- **Packed**: malware compresso (almeno una volta) tramite tecniche di *packing* per eludere i controlli degli antivirus [12].

1.3 Chiamata Api Windows

Le chiamate API (Application Programming Interface) di sistema consentono agli sviluppatori di accedere alle risorse della macchina necessarie per l'esecuzione di un applicativo. Attraverso tali interfacce è possibile interagire con componenti fondamentali come la rete, la memoria, i dispositivi di input/output e l'interfaccia grafica. Le API risultano fortemente dipendenti dal sistema operativo su cui vengono eseguite. In ambiente Windows, le API garantiscono che un'applicazione possa funzionare correttamente su diverse versioni del sistema operativo, sfruttando al tempo stesso le specifiche funzionalità introdotte nelle specifiche versioni [13]. Le API di Windows coprono un ampio spettro di funzionalità, tra cui: interfaccia utente e shell, gestione dell'input e della messaggistica, accesso ai dati e archiviazione, diagnostica, grafica e multimedialità, dispositivi, servizi di sistema, sicurezza e identità, installazione e manutenzione di applicazioni, amministrazione e gestione del sistema, rete e connettività Internet.

Un esempio rappresentato dal malware di tipo **Spyware**, che può invocare ripetutamente la funzione `GetKeyboardState` [14] per recuperare lo stato degli ultimi 256 tasti premuti dall'utente, potenzialmente contenenti credenziali sensibili. Nella documentazione ufficiale, i requisiti della funzione sono riassunti in Tabella 1.1.

Tabella 1.1: Requisiti

Voce	**Valore**
Client minimo supportato	Windows 2000 Professional (solo app desktop)
Server minimo supportato	Windows 2000 Server (solo app desktop)
Piattaforma di destinazione	Windows
Intestazione	`winuser.h` (include `Windows.h`)
Libreria	`User32.lib`
DLL	`User32.dll`
Set di API	`ext-ms-win-ntuser-rawinput-l1-1-0` (Windows 10, versione 10.0.14

Come si nota, alla voce **ddl** è indicata la libreria `User32.dll`. Questa verrà referenziata nella sezione *Directory dati dell'intestazione facoltativa* dell'eseguibile corrispondente, permettendo al sistema operativo di risolvere la chiamata in fase di esecuzione.

Nel contesto di questo lavoro di tesi, non siamo interessati alla presenza di una singola chiamata API in sé, quanto piuttosto alla frequenza con cui tali chiamate compaiono all'interno delle sequenze. È infatti la distribuzione delle invocazioni a costituire la base per la classificazione del comportamento del software.

1.4 Contributi

La presente tesi è articolata in quattro capitoli principali: *Introduzione*, *Stato dell'Arte*, *Approccio Proposto*, *Validazione Empirica* e *Appendice*.

Nel capitolo di *Introduzione* vengono presentati il contesto di riferimento, le motivazioni, l'obiettivo del lavoro e il contributo originale fornito.

Il capitolo dedicato allo *Stato dell'Arte* offre un inquadramento teorico e metodologico, approfondendo i principali approcci di analisi del software (statica e dinamica), i fondamenti di *machine learning* con particolare attenzione al compito di classificazione, gli algoritmi utilizzati (Random Forest [15] e XGBoost [16]) e le metriche di valutazione. La sezione si conclude con una panoramica dei lavori correlati nel campo della rilevazione di malware in ambiente Windows.

Nel capitolo *Approccio Proposto* viene descritta in dettaglio la soluzione implementata. Si illustra la modellazione mediante diagrammi UML, l'architettura software sviluppata in Python secondo i principi della programmazione a oggetti.

Successivamente, nel capitolo *Validazione Empirica* verranno presentanti i dataset adottati, la standardizzazione addotta, le metriche scelte per la valutazione, i risultati sperimentali ottenuti.

Nel penultimo capitolo, *Conclusioni e Sviluppi Futuri*, verranno tratte le conclusioni del lavoro svolto e saranno delineati alcuni possibili ampliamenti e direzioni di ricerca futura.

Infine, l'*Appendice* contiene note tecniche relative all'ambiente di sviluppo e ulteriori dettagli implementativi, utili a comprendere lo sviluppo effettivo.

Capitolo 2

Background e Stato dell'Arte

In questo capitolo vengono presentati i fondamenti concettuali per il riconoscimento dei malware sulla base dei loro comportamenti. Saranno introdotte le due principali tecniche di analisi, statica e dinamica, mettendone in evidenza punti di forza e limitazioni. In seguito, l'attenzione verrà posta sull'analisi dinamica, approccio adottato in questa tesi, illustrandone i principi di machine learning adoperati per la sua realizzazione.

2.1 Analisi Statica e Dinamica

L'analisi statica si concentra sull'esame del codice del programma alla ricerca di specifiche sequenze di istruzioni, denominate *virus signature* (firma del virus) [17]. L'efficacia dei sistemi basati su questa tecnica dipende dalla dimensione e dall'aggiornamento del database delle firme e dall'impossibilità, per il software, di modificare nel tempo la propria struttura. Per aggirare tali sistemi di rilevamento, i creatori di malware hanno sviluppato diverse tecniche di offuscamento, che consentono di generare firme differenti rispetto a quelle originarie, rendendo più complessa la loro individuazione. Tra le principali tecniche ritroviamo:

- **Dead Code Insertion** (inserimento di codice morto): aggiunta di istruzioni prive di funzionalità ai fini malevoli, il cui unico scopo è alterare la firma rilevata [17].
- **Code Transposition** (trasposizione del codice): riordinamento delle istruzioni, in modo che la rappresentazione binaria risulti differente pur mantenendo inalterata la logica di esecuzione [17].
- **Register Reassignment** (riassegnazione dei registri): modifica dei registri utilizzati nelle operazioni interne al programma [17].
- **Instruction Substitution** (sostituzione delle istruzioni): rimpiazzo di blocchi di codice con altri semanticamente equivalenti [17].

Il continuo sviluppo di tecniche di offuscamento e l'aggiornamento dei database di firme ha dato vita a un vero e proprio ciclo vizioso, in cui l'evoluzione dei malware procede di pari passo con quella dei sistemi di rilevamento. Ne consegue che l'analisi statica tende a concentrare gli sforzi non tanto sull'identificazione diretta del codice malevolo, quanto piuttosto sulla gestione delle tecniche di offuscamento, finendo così per allontanarsi dal compito primario di rilevare il malware e, soprattutto, di coglierne i reali comportamenti.

L'analisi dinamica supera i limiti dell'approccio statico, poiché osserva i comportamenti effettivi del software durante l'esecuzione. In questo modo non solo riduce l'impatto delle tecniche di offuscamento sul codice, ma consente anche di evidenziare quei casi limite che l'analisi statica difficilmente riesce a cogliere, risultando complessivamente più efficace [18]. Per poter eseguire un analisi dinamica, esistono diverse tecniche:

- **Hooking** (Aggancio): intercetta chiamate API per monitorare o modificare il comportamento di un programma. Utilizzato spesso dagli antivirus per controllare processi sospetti. Limite: i malware possono rilevare la presenza di hook tramite checksum o controlli interni [18]. Un software per intercettare le chiamate API di Windows è *Detours* [19].

- **Dynamic Binary Instrumentation** (Strumentazione Dinamica del Binario): analizza e modifica il codice binario mentre il programma è in esecuzione, permettendo di osservare ogni istruzione o comportamento in tempo reale. Più difficile da rilevare rispetto all'hooking [18]. Un software che permette tale realizzazione è *Dynamorio* [20].

- **Virtualization** (Virtualizzazione): esegue un sistema operativo guest su hardware reale (host) isolato, sfruttando meccanismi di separazione. Veloce e relativamente sicuro, ma alcuni malware possono rilevare VM tramite differenze hardware/software [18].

- **Application Level Emulation** (Emulazione a Livello di Applicazione): crea un ambiente finto che emula il sistema operativo e l'architettura di istruzioni per singole applicazioni. Utile per analisi real-time e unpacking[1], ma limitato nella fedeltà: i malware possono rilevare l'emulazione usando istruzioni o API rare [18].

- **Whole System Emulation** (Emulazione dell'Intero Sistema): emula completamente l'hardware di un PC, permettendo l'installazione di un intero sistema operativo guest. Molto utile per osservare comportamenti complessi, unpacking e analisi dettagliata, ma più lento rispetto alla virtualizzazione [18].

[1] Per *unpacking* si intende il processo di estrazione del codice originario

Un esempio concreto di implementazione della tecnica di **hooking** è presentato nell'articolo di *Wtrace* [21], dove l'utilizzo della libreria *Detours* permette di intercettare le chiamate API di un processo in esecuzione. Il risultato di tale analisi può essere osservato nella Figura 2.1, che mostra un output delle API Call registrate di un programma in esecuzione:

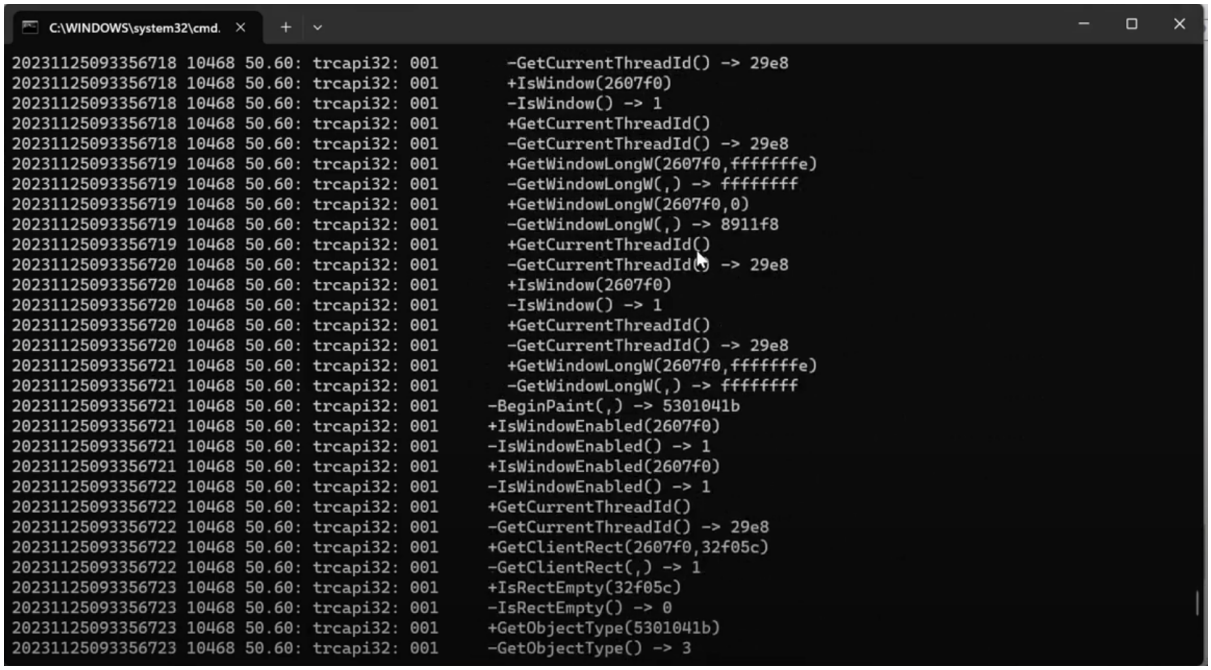

Figura 2.1: API Call catturate

Tuttavia, il semplice elenco delle chiamate API non è sufficiente per mostrare la vera natura di un software. É necessario definire una logica in grado di estrarre pattern a partire da questi dati in grado di identificare in maniera automatica i comportamenti malevoli. In questa tesi tali logiche sono implementate tramite *machine learning*.

2.2 Machine Learning

Il *machine learning* (ML) è una branca dell'intelligenza artificiale (IA) che consente a computer e macchine di apprendere dai dati, imitando i processi cognitivi umani [22]. In termini pratici, il ML consiste nell'**addestramento** di un software, detto **modello**, per svolgere compiti specifici come effettuare **previsioni** o generare contenuti (testo, immagini, audio o video) a partire da dati osservati [23]. Un modello di *machine learning* può essere visto come un costrutto matematico che riceve dati in ingresso (*input*) e produce un risultato (*output*), in modo analogo a una funzione matematica [24]. L'**allenamento** di un modello consiste nel processo di identificazione dei parametri ottimali interni, così che il modello possa compiere previsioni accurate; ciò avviene fornendogli una serie di

esempi [25]. Un **esempio** è un'istanza di input descritta da un insieme di variabili organizzate in un **vettore delle caratteristiche** (*feature vector*) [26]. Ogni variabile è detta **caratteristica** (*feature*) e rappresenta un'informazione specifica del dominio del problema [27]. Le caratteristiche possono essere di tipo **numerico**, quando esprimono valori quantitativi, oppure **categorico**, quando descrivono attributi qualitativi. Un insieme di esempi costituisce un *dataset*.

I modelli di *machine learning*, in base al tipo di addestramento e ai dati disponibili, possono essere suddivisi in quattro categorie principali:

- **Apprendimento supervisionato**: il modello impara a fare previsioni a partire da dati etichettati, ossia accompagnati dalle risposte corrette, scoprendo le relazioni tra input e output [23].
- **Apprendimento non supervisionato**: il modello lavora con dati privi di etichette e ha come obiettivo l'identificazione di strutture o pattern nascosti nei dati [23].
- **Apprendimento per rinforzo**: il modello interagisce con un ambiente, ricevendo feedback sotto forma di ricompense o penalità, e sviluppa progressivamente una strategia ottimale per il raggiungimento di un obiettivo [23].
- **Apprendimento generativo**: i modelli non si limitano a classificare o predire, ma generano nuovi contenuti (testo, immagini, audio, ecc.) a partire dagli input forniti dall'utente [23].

In questa tesi ci concentreremo sull'**apprendimento supervisionato**, affrontando in particolare il compito di **classificazione**.

2.3 Classificazione

Il compito di **classificazione** rientra tra quelli affrontabili tramite *apprendimento supervisionato*. In questo paradigma, gli esempi utilizzati durante l'addestramento sono accompagnati dall'output corretto, detto **etichetta** (label) [28]. Nel caso specifico della classificazione, l'etichetta corrisponde a una **classe**, ossia il valore discreto che identifica a quale categoria appartiene l'esempio. Allenando quindi un modello su un dataset etichettato, l'obiettivo della classificazione è predire la classe corretta di un nuovo esempio non etichettato.

Esistono molteplici compiti di classificazione, distinti uno dall'altro dal numero delle classi da distinguere e da quanto sono mutualmente esclusivi [29].

- **Classificazione binaria**: le classi tra cui predire sono due e mutualmente esclusive [29].

- **Classificazione multi classe**: le classi tra cui predire sono più di due e mutualmente esclusive [29].
- **Classificazione a multipla etichetta**: un esempio può appartenere a più classi [29].

In questo lavoro di tesi verrà affrontata la classificazione multi classe.

2.3.1 Algoritmi di Classificazione

Esistono molteplici algoritmi per la classificazione, in questa tesi verranno affrontanti *XGboost* e *Random forest*.

Random forest

Prima di introdurre l'algoritmo di apprendimento *random forest*, è utile esaminare due concetti fondamentali alla sua comprensione: *bagging* e *decision tree*.

L'algoritmo di apprendimento *bagging* appartiene alla categoria dei metodi di **apprendimento d'insieme**, il cui obiettivo è ridurre la varianza[2] all'interno di un dataset rumoroso [31]. L'apprendimento d'insieme si basa sull'idea che un gruppo di modelli che collaborano possa raggiungere prestazioni superiori rispetto a un singolo modello. Questo principio viene spesso associato al concetto di "saggezza delle folle", secondo cui un insieme di persone fornisce mediamente decisioni migliori di quelle di un singolo esperto [31]. Nel *bagging*, i singoli modelli dell'insieme vengono addestrati — anche in parallelo — su sottoinsiemi del dataset originario, generati tramite campionamento con reinserimento (bootstrap) [30]. Ciò implica che uno stesso esempio possa comparire più volte all'interno di uno o più sottoinsiemi. Al termine dell'addestramento, la previsione finale del sistema viene ottenuta aggregando le risposte di tutti i modelli. Nel caso della classificazione, la classe predetta corrisponde a quella che ha ricevuto il maggior numero di voti.

L'algoritmo *decision tree* è costituito da una serie di domande organizzate gerarchicamente a forma di albero. Le domande sono generalmente chiamate *condizioni*, *split* o *test*. Ogni nodo interno (non foglia) contiene una condizione, mentre ogni nodo foglia rappresenta una previsione. Per effettuare una predizione, si parte dal nodo radice e si percorre l'albero fino a raggiungere un nodo foglia, scegliendo il ramo da seguire in base all'esito della condizione presente in ciascun nodo. Il percorso che va dalla radice fino alla foglia è chiamato *percorso di inferenza* [32].

L'algoritmo *Random Forest* è un'estensione dell'algoritmo di *bagging*, in cui i modelli interni sono costituiti da *decision tree*. Oltre a essere addestrati su sottoinsiemi

[2] La varianza si riferisce alla sensibilità del modello rispetto alle fluttuazioni nei dati di addestramento [30].

casuali degli esempi del dataset (*bagging*), ad ogni nodo viene selezionato un sottoinsieme casuale di *feature* candidate per effettuare lo *split*. Questo introduce diversità tra gli alberi, riducendo la correlazione tra di essi e aumentando la robustezza del modello complessivo [33].

XGBoost

Per una corretta comprensione dell'algoritmo di apprendimento *XGBoost*, è utile introdurre preliminarmente l'algoritmo di apprendimento *boosting* e il concetto di *ascesa del gradiente*.

Il *boosting* è una tecnica di apprendimento sequenziale in cui più modelli vengono addestrati uno dopo l'altro, ciascuno dei quali cerca di correggere gli errori commessi dal modello precedente, al fine di ottenere una previsione progressivamente più accurata [34]. Il *boosting* appartiene alla categoria degli apprendimenti d'insieme.

La *discesa del gradiente* [35] è un algoritmo di ottimizzazione che permette di trovare i valori ottimali dei parametri di un modello minimizzando una funzione di costo. L'idea di base è semplice: si parte da valori iniziali arbitrari per i parametri, si calcola la pendenza (il *gradiente*) della funzione di costo in quel punto e si aggiornano i parametri muovendosi nella direzione opposta al gradiente, cioè verso la discesa più ripida. Il processo si ripete finché non si raggiunge un minimo (locale o globale). Un ruolo cruciale è svolto dal *tasso di apprendimento* (*learning rate*), che stabilisce la grandezza dei passi: se troppo grande vi è il rischio di superare il minimo, mentre se troppo piccolo l'addestramento risulta molto lento. Esistono diverse varianti:

- **Batch Gradient Descent**: aggiorna i parametri dopo aver visto tutti i dati di addestramento [35].
- **Stochastic Gradient Descent (SGD)**: aggiorna i parametri dopo ogni esempio, introducendo rumore ma anche la possibilità di sfuggire ai minimi locali [35].
- **Mini-batch Gradient Descent**: compromesso tra i due, suddivide i dati in piccoli blocchi [35].

L'obiettivo della discesa del gradiente è ridurre progressivamente l'errore, fino a convergere a un set di parametri che rende il modello il più accurato possibile.

L'algoritmo di apprendimento *XGBoost* si basa sulla tecnica del *boosting*, in cui un modello iniziale, l'albero decisionale, viene progressivamente migliorato tramite l'uso della *discesa del gradiente* [36].

2.4 Approcci di Machine Learning per Windows PE Malware Detection

Diversi studi hanno affrontato il problema della rilevazione dei malware in ambiente Windows basandosi sull'analisi delle chiamate API.

Ad esempio, Zhang et al. (2015) propongono un approccio innovativo per la rilevazione di malware basato sull'analisi dinamica delle sequenze di chiamate API. Gli autori applicano algoritmi di allineamento delle sequenze di DNA per identificare somiglianze tra le sequenze di chiamate API di programmi legittimi e malware. Gli algoritmi di allineamento delle sequenze risolvono il compito di calcolare il costo minimo necessario per trasformare una stringa a in una stringa b, consentendo operazioni di *inserimento* (aggiunta di un carattere) e *rimozione* (eliminazione di un carattere). Una delle soluzioni più note a questo problema è l'algoritmo di Smith-Waterman [37]. Nello studio, ogni esempio del dataset è rappresentato da un vettore di feature costituito dai costi di allineamento calcolati tra una sequenza sconosciuta e tutte le sequenze note del dataset. Il modello proposto ha mostrato buone prestazioni nella classificazione dei malware, evidenziando l'efficacia dell'approccio basato sull'allineamento delle sequenze di chiamate API [38].

Amer e Zelinka (2020) introducono un approccio innovativo che combina tecniche di analisi semantica e modellazione probabilistica delle sequenze di API. L'idea si ispira ai metodi utilizzati nell'elaborazione del linguaggio naturale (NLP[3]), trattando le sequenze di chiamate API come frasi e le singole API come parole. Per rendere questa rappresentazione più significativa, gli autori applicano l'algoritmo *Word2Vec* [40], che proietta ogni API in uno spazio vettoriale, in cui elementi che compaiono in contesti simili risultano vicini tra loro. I vettori così ottenuti vengono successivamente organizzati in gruppi omogenei mediante *k-means clustering* [41], un algoritmo non supervisionato che assegna ciascun elemento a un cluster in base alla distanza dal centroide[4]. In questo modo, ogni API originale viene sostituita dall'indice del cluster di appartenenza, riducendo la complessità senza perdere le relazioni semantiche. A partire da questa rappresentazione, gli autori costruiscono due matrici di transizione distinte: una per le sequenze di software benigno e una per quelle malevole. In tali matrici, ciascuna cella rappresenta la probabilità di passare da uno stato i a uno stato j. Per classificare una nuova sequenza di API, questa viene mappata nella rappresentazione basata sui cluster e analizzata tramite un modello a catene di Markov, in cui la transizione dipende unicamente dallo stato corrente. Calcolando la probabilità complessiva della sequenza rispetto ai due modelli (goodware e malware), la classificazione viene effettuata scegliendo l'ipotesi con la

[3] *Natural Language Processing* è un ramo dell'intelligenza artificiale che abilita i computer a comprendere e interpretare il linguaggio umano [39]

[4] Il centroide è il vettore medio di tutti gli elementi che appartengono a un cluster

probabilità più elevata. I risultati mostrano performance molto promettenti: l'approccio raggiunge un'accuratezza fino al 99% e consente di effettuare previsioni precoci già dalle prime chiamate API, dimostrando l'efficacia della combinazione tra rappresentazione semantica e modellazione probabilistica.

Un contributo più recente è quello di Gond e Mohapatra (2025), che affrontano la classificazione dei malware considerando il fenomeno del *concept drift*, ovvero la capacità dei malware di adattare i propri comportamenti non solo nel tempo, ma anche in funzione del contesto di esecuzione, ad esempio il sistema operativo o l'ambiente (virtualizzato o reale) in cui operano. Gli autori combinano tecniche di deep learning con meccanismi di adattamento al drift: le sequenze di API raccolte tramite sandbox vengono trasformate in rappresentazioni linguistiche tramite n-grammi[5] e filtrate per rilevanza. Successivamente, un algoritmo genetico genera varianti delle feature per simulare nuovi schemi comportamentali emergenti. La classificazione è affidata a reti neurali artificiali, ricorrenti e convoluzionali CNN (Convolutional Neural Networks) [43], raggiungendo accuratezze fino al 99,5% anche in presenza di mutazioni nei dati. L'integrazione del genetic algorithm consente di ridurre la perdita di performance dovuta al concept drift [44].

Nello studio "Malware Detection based on API Calls Frequency" [45] La metodologia si articola in diverse fasi: in primo luogo, le chiamate API sono state estratte dai file e trasformate in un vettore di feature. Per ottimizzare il processo di addestramento e rimuovere le funzionalità ridondanti, è stata applicata la tecnica di *feature selection* tramite un algoritmo *Random Forest*. Questo ha permesso di identificare le 45 feature più significative. Successivamente, questi vettori di feature ottimizzati sono stati utilizzati per addestrare vari algoritmi di machine learning. Tra tutti i modelli testati, l'algoritmo*Support Vector Machine* (SVM [46]) ha dimostrato le prestazioni migliori, raggiungendo un'accuratezza del 93% nel distinguere il malware dai file benigni.

Sono stati provati anche approcci a grafo, come nello studio "A malware classification method based on directed API call relationships" [47]. A differenza degli approcci tradizionali che le considerano come una semplice lista, i ricercatori hanno sviluppato un modello che le tratta come un **grafo diretto**. In questo grafo, ogni nodo è una singola API e gli archi direzionali rappresentano l'ordine e la sequenza con cui le chiamate vengono eseguite. La metodologia si concentra sul modello **FSGCN** (First-order and Second-order Graph Convolutional Networks), il quale analizza il grafo su due livelli per estrarre informazioni cruciali. A un livello di *primo ordine*, il modello esamina le connessioni dirette tra le API per comprendere la sequenza immediata. A un livello di *secondo ordine*, analizza le relazioni indirette (percorsi a due passi), per comprendere il contesto più ampio del comportamento del programma. Questa combinazione di analisi consente di

[5]Un **n-gramma** è una sotto sequenza contigua di n elementi [42]

creare un **vettore di feature** unificato, che racchiude sia i dettagli della sequenza che il contesto relazionale. Il vettore di feature viene poi convertito in un'immagine, sfruttando l'efficacia delle **CNN** per la classificazione finale. L'efficacia di questo approccio dimostra che l'analisi delle relazioni strutturali e dell'ordine delle chiamate API è un metodo più robusto per la classificazione dei malware.

Questi studi evidenziano come l'analisi delle sequenze di API, se combinata con tecniche di rappresentazione avanzata e modelli adattativi, possa migliorare sensibilmente l'efficacia della malware detection in ambienti Windows.

Capitolo 3

Approccio Proposto

In questo capitolo viene illustrato il percorso seguito per la realizzazione del progetto. Nella sezione *Descrizione* vengono presentate le scelte metodologiche e le attività di tesi svolte durante lo sviluppo, con l'obiettivo di fornire una panoramica chiara dell'approccio adottato. La sezione *OOP & UML*, invece, descrive la progettazione dell'approccio proposto secondo i principi della programmazione orientata agli oggetti e illustrandola mediante diagrammi UML a supporto della progettazione.

Tutto il codice sviluppato, con annessa documentazione, è disponibile al repository github *ACPAMC* [48].

3.1 Descrizione

L'approccio proposto, in questa tesi riguardante la **classificazione di software** - nel formato PE - è l'analisi dei pattern comportamentali espressi dalle sequenze di chiamate API. Il percorso metodologico seguito può essere suddiviso nelle seguenti fasi principali: la raccolta e la standardizzazione dei dati, l'ingegneria delle caratteristiche, l'addestramento dei classificatori e la valutazione finale.

3.1.1 Raccolta e Standardizzazione dei Dati

Sono stati raccolti dataset rappresentanti le sequenze di chiamate API generate durante l'esecuzione dei file eseguibili PE. Tali dataset risultano distribuiti in formati eterogenei, come CSV[1], JSON[2], e formati personalizzati, rendendo difficoltosa un'analisi congiunta e

[1]Il formato **CSV** (Comma Separated Value) è un formato di condivisione dati in cui ogni riga rappresenta un dato di cui le caratteristiche sono separate da virgola [49]

[2]Il formato **JSON** (JavaScript Object Notation) è un formato per lo scambio di dati basato su una collezione di coppie nome/valore e di array [50].

omogenea. Per superare questa limitazione, si è resa necessaria una fase di **standardizzazione** in un unico formato: il **JSON**. La struttura adottata per ogni file standardizzato è stata pensata per rappresentare in modo chiaro e minimale le informazioni essenziali per l'addestramento. Lo schema per il JSON scelto è illustrato in Figura 3.1:

Key	Descrizione
`[].application_type`	Classe del software (*malware*, *goodware*, ...)
`[].apis`	Sequenza di chiamate API (array di stringhe)

```
[
    {
        "application_type": "malware",
        "apis": [
            "getProcessId",
            "deleteProcess"
        ]
    },
    {
        "application_type": "goodware",
        "apis": [
            "getMessageBox",
            "openDialog"
        ]
    }
]
```

Figura 3.1: Descrizione dei campi e corrispondente esempio JSON.

3.1.2 Ingegneria delle Caratteristiche: Il Modello Bag-of-Words

L'obiettivo della classificazione è distinguere le categorie di software basandosi sul loro comportamento, espresso dalle sequenze di chiamate API (un esempio di sequenza di chiamate API in Figura 3.2).

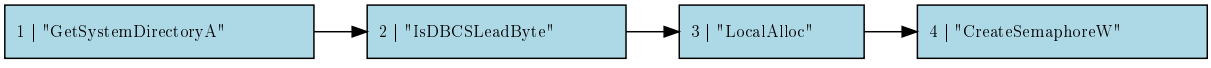

Figura 3.2: Esempio delle prime 4 chiamate API di un malware di tipo Dropper.

Per tradurre queste sequenze in un formato numerico interpretabile dagli algoritmi di Machine Learning, è stato costituito un **vettore delle caratteristiche** (feature vector) basato sulle **frequenze** con cui ogni singola API compare nella sequenza. A tal fine, è stato adottato l'approccio **Bag-of-Words** (BoW). In letteratura, questa tecnica,

comunemente usata in applicazioni di Natural Language Processing, consente di rappresentare un documento testuale (nel nostro caso, la sequenza di API di un programma) come un vettore di interi. In questo vettore, ogni elemento indica il numero di occorrenze di una determinata parola (cioè di una chiamata API), mentre l'indice corrisponde alla parola di riferimento, ignorando l'ordine temporale con cui le API compaiono nella sequenza [51]. Per implementare il modello BoW e ottenere il vettore delle caratteristiche, è stata eseguita la seguente procedura:

1. **Creazione del Vocabolario**: tutte le chiamate API univoche presenti nel dataset sono state estratte per comporre il vocabolario. Tale vocabolario è rappresentano da un vettore di stringhe ordinato.

2. **Inizializzazione vettore delle caratteristiche**: per ogni esempio del dataset, si crea un vettore di interi, inizializzato a 0, avente cardinalità pari al vettore del vocabolario.

3. **Popolamento**: per ogni chiamata API presente nella sequenza del esempio, si individua la sua posizione nel vettore del vocabolario, denominando questa posizione con il valore i, si incrementa di 1 il valore nella posizione i del vettore delle caratteristiche.

Il risultato finale è un vettore di frequenze assolute a lunghezza fissa, in cui ogni componente quantifica il numero di occorrenze di una specifica API.

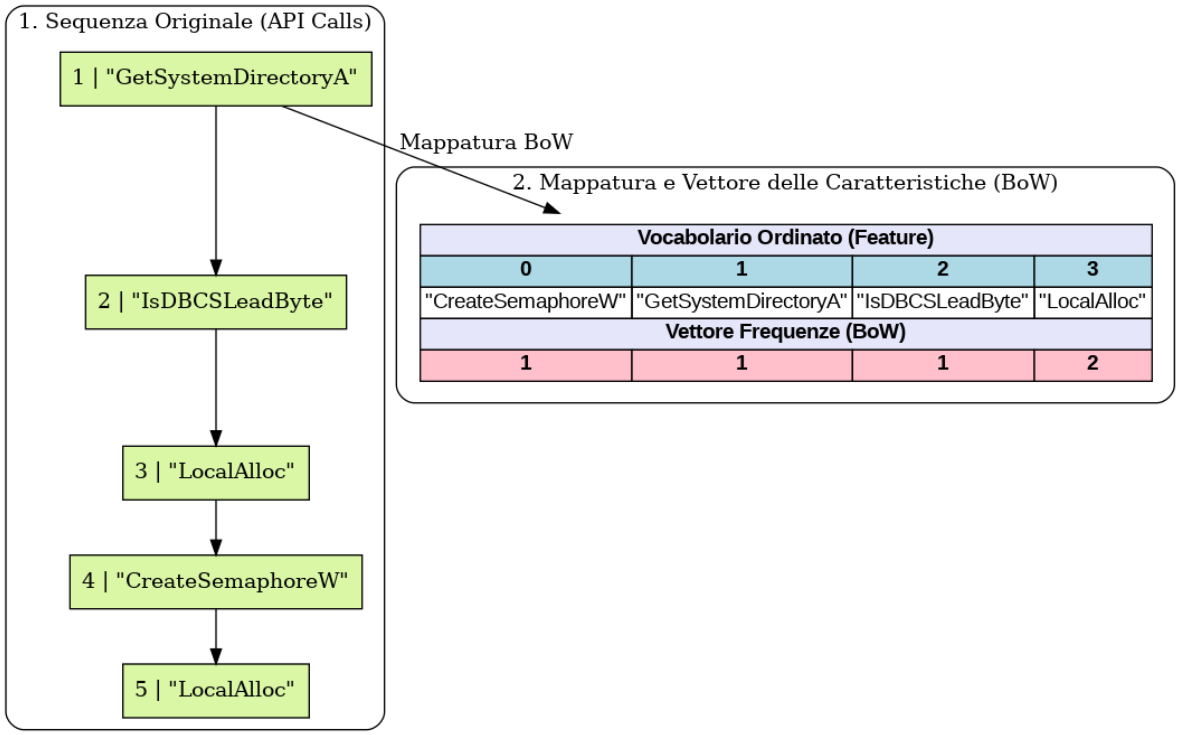

Figura 3.3: Esempio di ingegnerizzazione delle caratteristiche

3.1.3 Addestramento e Valutazione

Per ogni dataset e algoritmo di classificazione utilizzato è stato addestrato un modello dedicato attraverso l'impiego del sistema. L'avvio del sistema richiede la specifica dei seguenti parametri di configurazione presenti in Tabella 3.1.

Parametro	Descrizione
`file_name`	Nome del file del dataset standardizzato da utilizzare per l'intera fase di addestramento e/o valutazione.
`random_state`	Valore seme (*seed*) per la gestione dei numeri casuali, utile a garantire la replicabilità delle divisioni del dataset.
`training_size`	Proporzione del dataset da dedicare al *training set*; la parte complementare è destinata al *testing set*.
`classifier`	Algoritmo di classificazione da istanziare e addestrare.
`train`	Flag booleano che forza l'addestramento del modello; se impostato a `False`, viene riutilizzato un modello precedentemente salvato (se disponibile).

Tabella 3.1: Tabella di configurazione dei parametri del sistema.

Una volta impostati i parametri, il sistema provvede a caricare in memoria il dataset standardizzato corrispondente, applicando la logica descritta in sottosezione 3.1.2 per la rappresentazione vettoriale delle caratteristiche. Il dataset così ottenuto viene suddiviso in *training set* e *testing set* secondo la percentuale definita dal parametro *training_ size*,

fissato in questo lavoro a 0.8, corrispondente all'80% dei dati destinati alla fase di addestramento. La divisione è effettuata mediante **campionamento stratificato**(*Stratified Sampling*) [52], in modo da garantire la corretta proporzionalità delle diverse classi di software in entrambi i sottoinsiemi, prevenendo distorsioni del modello e assicurando un'adeguata rappresentanza anche delle classi minoritarie. Il classificatore sviluppato è di tipo **supervisionato** e **multi classe**, ossia apprende a distinguere le classi a partire da esempi etichettati. Il sistema è in grado di riconoscere tra applicazioni **benigne** e **malware**, e, se fornito dal dataset, è capace di distinguere i diversi **tipi di malware** (*dropper*, *trojan*, ecc.). In particolare, il sistema riceve in input il *training set* composto da vettori di caratteristiche e dalle corrispondenti etichette, e produce in output un modello addestrato che prende in input un esempio nuovo (che fa parte del *testing set*) e ne predice la classe. Nel lavoro proposto, il classificatore può essere implementato tramite uno dei due algoritmi considerati: *Random Forest* o *XGBoost*, entrambi forniti dalla libreria *Scikit-learn* [53].

3.2 OOP & UML

La fase di implementazione del progetto è stata condotta adottando il paradigma della **Programmazione Orientata agli Oggetti** (OOP). Questo approccio si è rivelato fondamentale per strutturare il codice in modo *modulare*, *manutenibile* e *scalabile*, in quanto ha permesso di modellare le entità del problema (come i dataset, le sequenze di API e i classificatori) come componenti software interconnesse. L'efficacia dell'OOP si fonda su quattro pilastri concettuali:

- *Astrazione*: Consente di modellare le entità come classi, focalizzandosi unicamente sugli attributi e sulle interazioni rilevanti per il contesto applicativo, nascondendo la complessità sottostante [54].
- *Incapsulamento*: Protegge lo stato interno di un oggetto, nascondendolo e consentendo l'interazione con le sue funzionalità per garantirne l'integrità [54].
- *Ereditarietà*: Permette di creare nuove astrazioni (sottoclassi) basandosi su astrazioni esistenti (superclassi), facilitando il riuso del codice e stabilendo gerarchie logiche tra le classi [54].
- *Polimorfismo*: Consente a oggetti diversi di rispondere allo stesso messaggio (chiamata di metodo) in modi specifici alla propria classe, permettendo di implementare proprietà o metodi ereditati in forme distinte tra diverse astrazioni [54].

Per formalizzare e documentare questa architettura basata sui principi OOP, è stato utilizzato l'**Unified Modeling Language** (UML) [55]. Nello specifico, il **Diagramma**

delle Classi è stato scelto per illustrare la struttura del sistema e le relazioni tra i suoi componenti. Il diagramma delle classi completo che definisce l'architettura logica del progetto è presentato in Figura 3.4. Far riferimento alla sezione 6.2 per una guida su come leggere il diagramma UML.

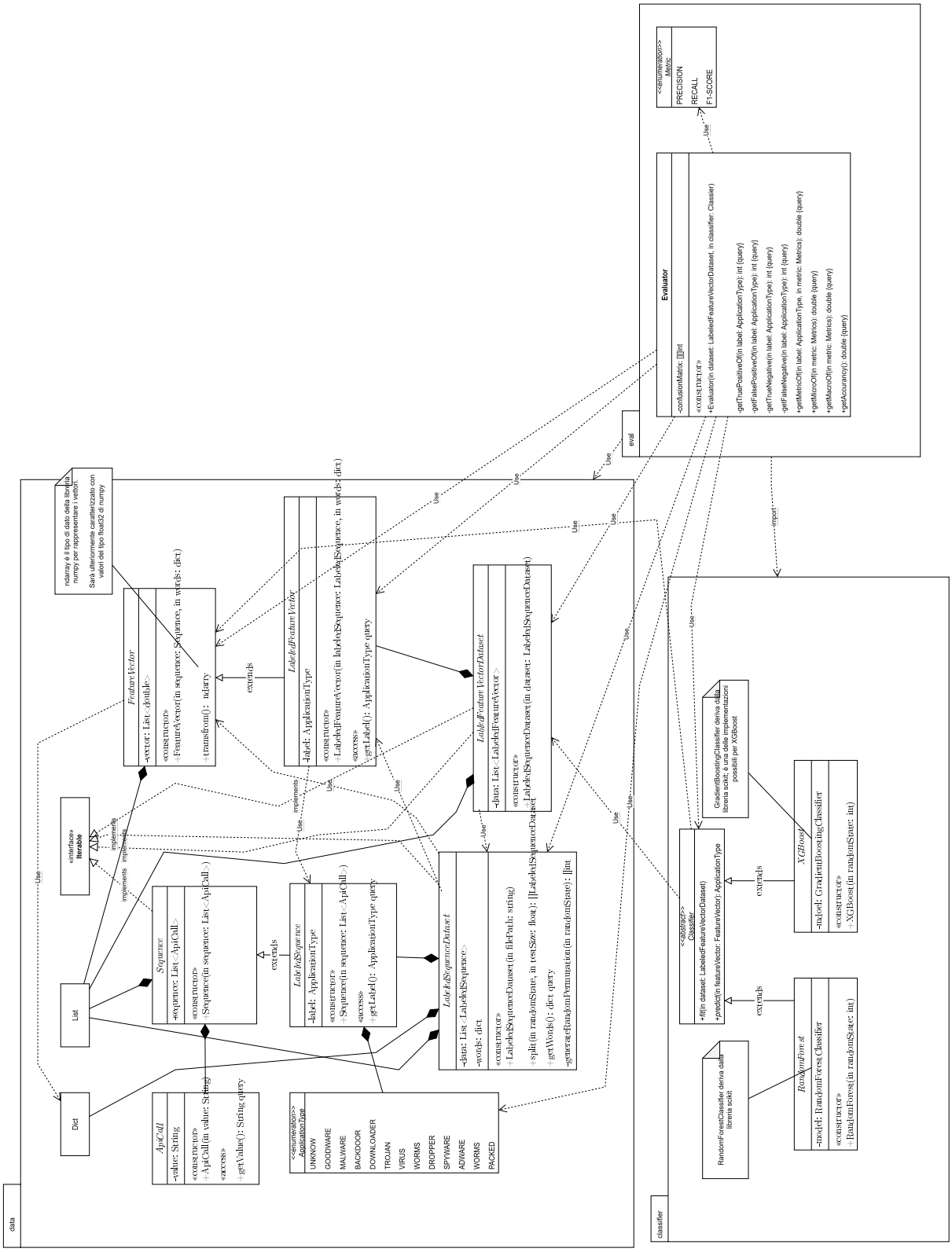

Figura 3.4: Diagramma delle classi UML che descrive l'architettura OOP del sistema.

L'intero sistema è basato su 3 package: *data*, *classifier* e *eval*.

Il package *data* gestisce tutte le entità del dominio, dal caricamento dei dati in memoria fino alla loro rappresentazione vettoriale per l'addestramento dei classificatori. Le classi *ApiCall* e *Sequence* descrivono la traccia dinamica di un file PE: la prima incapsula una singola chiamata API, mentre la seconda raccoglie una sequenza ordinata di chiamate. Per aggiungere l'informazione relativa alla classe di appartenenza di una sequenza, il package introduce l'enumerazione *ApplicationType*, che elenca le possibili categorie (classi) del dominio, e la classe *LabeledSequence*, che associa ogni sequenza alla propria classe. La gestione complessiva del dataset è affidata alla classe *LabeledSequenceDataset*, che raccoglie e organizza un insieme di sequenze etichettate, occupandosi anche del caricamento dei dati in formato JSON e della logica di suddivisione del dataset. Infine, per rappresentare i dati in una forma adatta agli algoritmi di classificazione, sono state introdotte le classi *FeatureVector*, *LabeledFeatureVector* e *LabeledFeatureVectorDataset*, che modellano rispettivamente un singolo vettore di caratteristiche, un vettore con la relativa etichetta e l'intero insieme dei vettori etichettati.

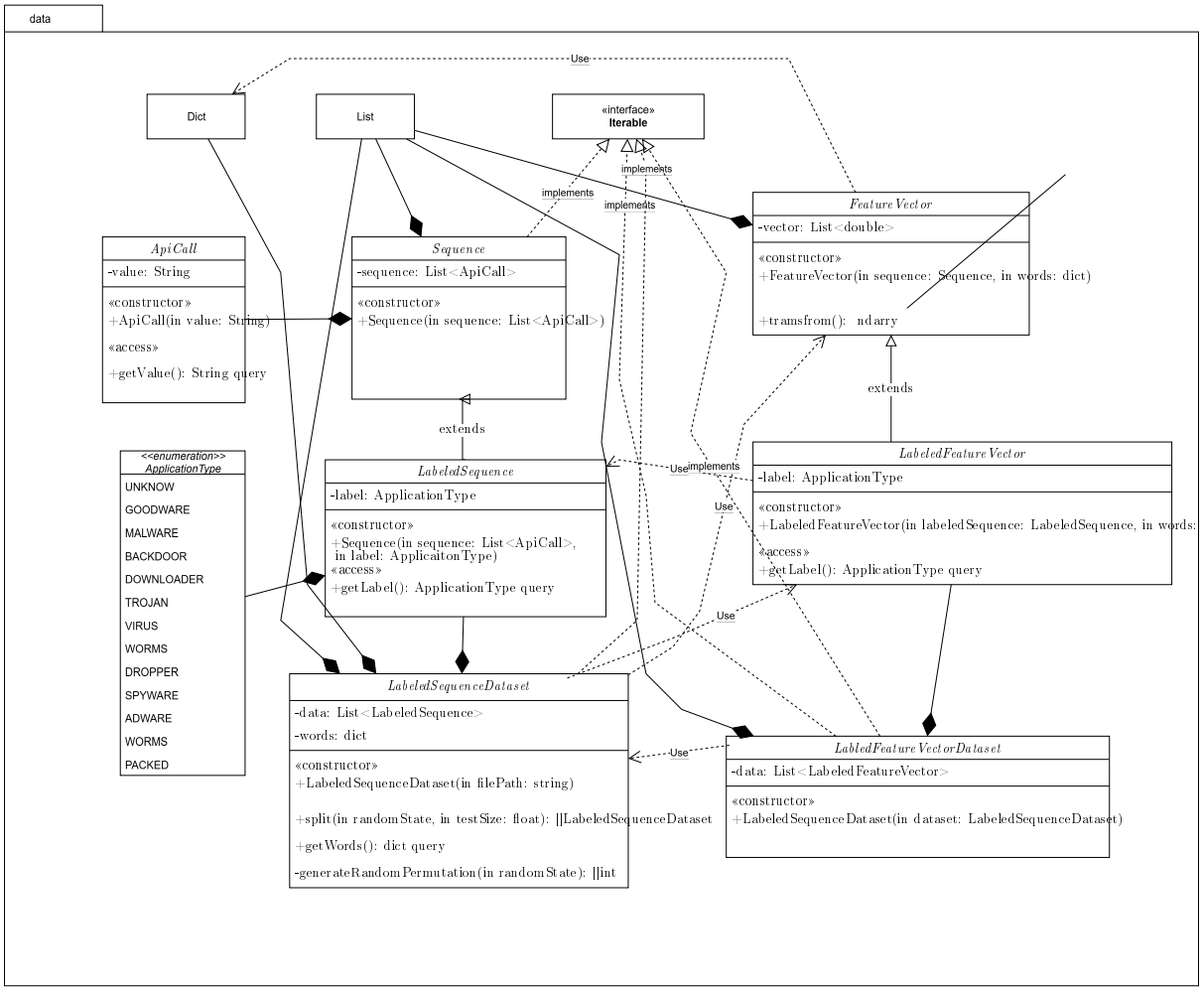

Figura 3.5: UML - Package data

Il package *classifier* rappresenta l'astrazione e le sue specializzazioni per un classificatore. La classe astratta *Classifier* ingloba i metodi *predict* e *fit*, responsabili rispettivamente della classificazione e dell'allenamento del modello. Le classi *RandomForest* e *XGBoost* specializzano la classe *Classifier*, implementando gli omonimi algoritmi di classificazione.

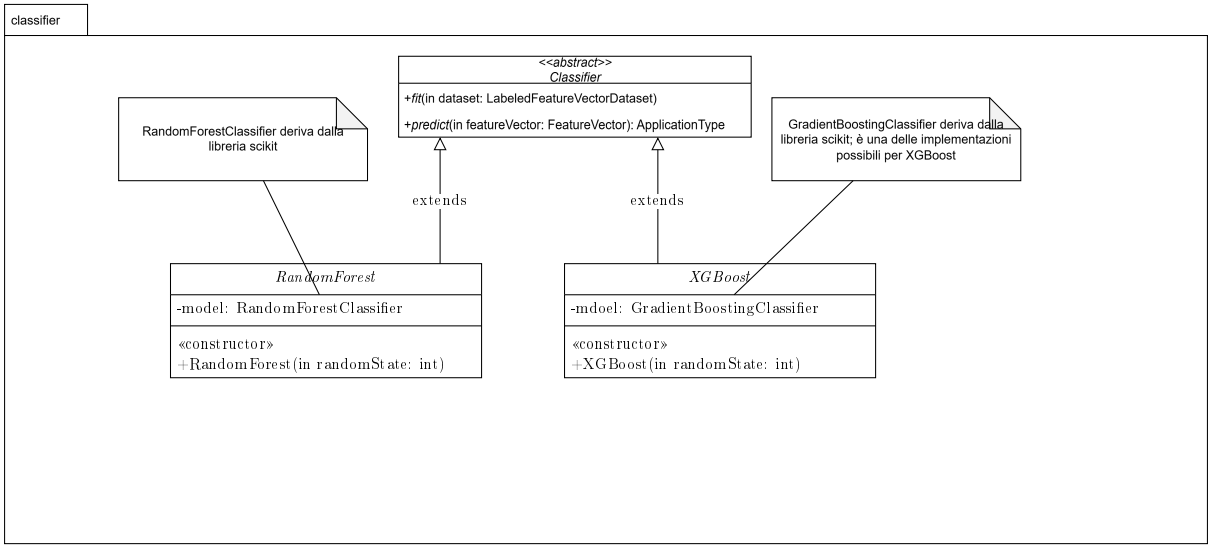

Figura 3.6: UML - Package classifier

Il package *eval* è responsabile delle metriche di valutazione del modello. La classe *Evaluator* implementa le principali metriche di valutazione, includendo al suo interno la matrice di confusione. La classe *Metric* è un'enumerazione utilizzata per parametrizzare i metodi di calcolo delle metriche nella classe *Evaluator*.

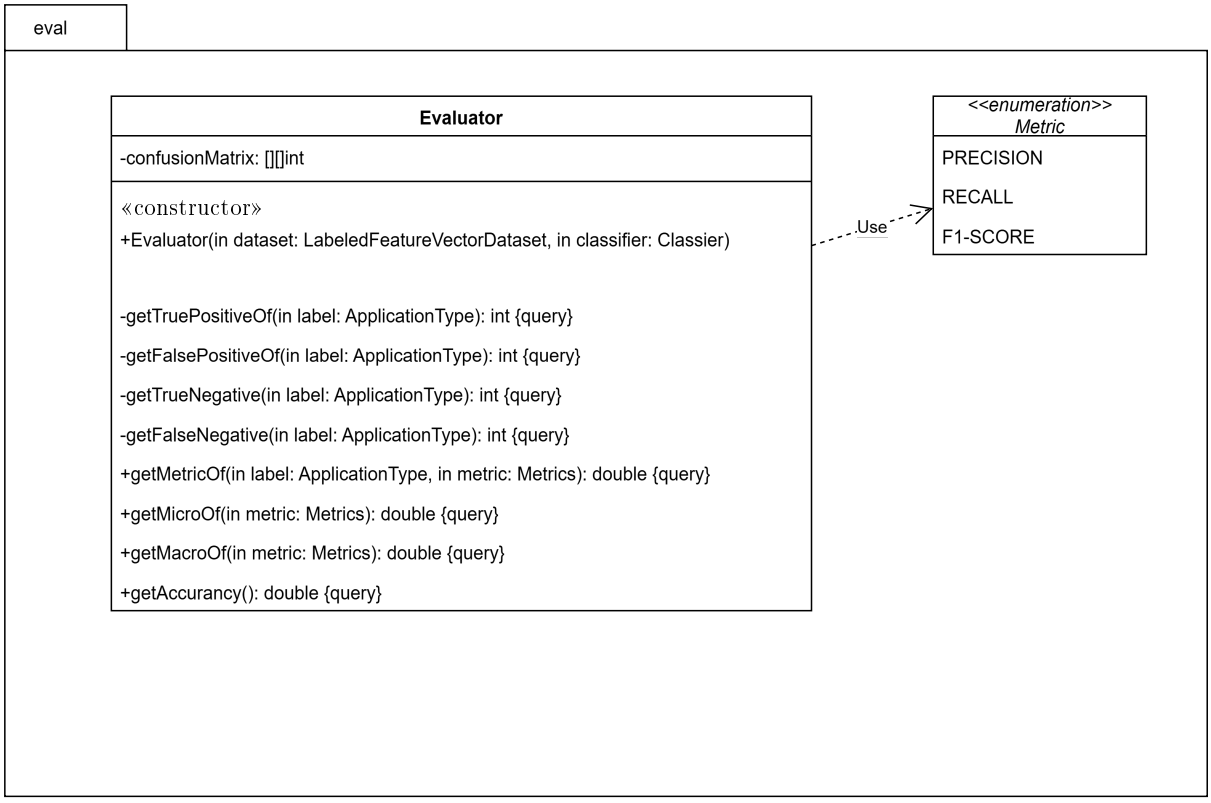

Figura 3.7: UML - Package eval

Capitolo 4

Validazione Empirica

Il presente capitolo illustra il processo di validazione empirica del lavoro svolto, descrivendo nel dettaglio i dataset utilizzati, le metriche adottate per la valutazione dei classificatori e i risultati ottenuti dall'approccio sviluppato.

4.1 Dataset

La validazione empirica è stata condotta impiegando quattro dataset pubblici di chiamate API reperiti dalla letteratura scientifica. Nel seguito, essi verranno indicati mediante i seguenti alias: il dataset proposto da *A Novel Approach to Detect Malware Based on API Call Sequence Analysis* [56] sarà denominato **apimds**; quello descritto in *Data augmentation based malware detection using convolutional neural networks* [57] sarà indicato come **octak**; il dataset introdotto da *API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques* [58] come **mpasco**; infine, il dataset derivante da *Quo Vadis: Hybrid Machine Learning Meta-Model Based on Contextual and Behavioral Malware Representations* [59] sarà indicato come **quovadis**.

Il dataset **apimds** è rappresentato in formato tabulare dove ogni riga (esempio) contiene nel seguente ordine le feature: la classe dell'esempio, l'hash del PE, e le restanti colonne le chiamate API effettuate. In particolare per l'addestramento è stata utilizzata la prima colonna (la classe) come etichetta e le colonne successive alla seconda per recuperare la sequenza di chiamate API. In Figura 4.1 un esempio del dataset. Il numero totale di esempi è pari a 23146, di cui 80% dedicati al training (18519) e il restante 20%(4627) al testing. La media di chiamate API per esempio è pari a 284.391 e il numero di chiamate distinte (quindi le feature) è pari a 1165. Le classi presenti sono *Malware* (27.4% - 6111), *Backdoor* (2.6% - 582), *Downloader* (4,1% - 917), *Trojan* (46.9% - 10450), *Virus* (14.6% - 3216) e *Packed* (4.3% - 964). La classe *Goodware* non è presente e la classe *Malware* è stata utilizzata per rappresentare tutti quegli esempi per i quali non è stato fornita

un ulteriore specializzazione dal dataset. In Figura 4.2 e Figura 4.3 vengono riportati le distribuzioni delle classi del dataset.

```
1 "Worm.Win32.Zwr.c","172b85e684b8426b4fe55c0245e4675ce1e3687b73a8c9fee2d1cc5cd
    ↰ 84a1787","GetSystemDirectoryA","IsDBCSLeadByte","LocalAlloc","CreateSem
    ↰ aphoreW","CreateSemaphoreA","GlobalAddAtomW"
2 "Trojan.Win32.FakeAV.rbzp","0000e27a0471230eecf42e629d36b617c2726e26181b3a7e7
    ↰ e4c03097ae225e5","GetProcessVersion","LocalAlloc","CreateSemaphoreW","C
    ↰ reateSemaphoreA","GlobalAddAtomW","lstrcpynW","LoadLibraryExW"
3 "Worm.Win32.Vobfus.dvpo","00038a453a4eda579348af45b1827393227c1e19be732a75859
    ↰ 402ce8ffb4a32","lstrcpyW","GetThreadLocale","lstrcmpiW","GlobalAlloc","
    ↰ GlobalLock","GlobalUnlock","GlobalReAlloc","RegisterClipboardFormatW"
4 "Packed.Win32.PolyCrypt.d","00121cae10b7647a75205dd6ea59a50124612139e5e01a1b4
    ↰ 3be16d0710428c8","GetCommandLineA","GetStartupInfoA","LockResource","Ge
    ↰ tModuleFileNameA","IsBadWritePtr","RegisterClipboardFormatW"
5 "Backdoor.Win32.Bredolab.nfz","0021cc116edb62aecd3ed772571c2606dcda9bac1931cd
    ↰ 573295fd997827e0b4","GetDeviceCaps","ReleaseDC","SearchPathW","CreateFi
    ↰ leW","CreateFileMappingW"
```

Figura 4.1: Una porzione del dataset **apimds** troncato per una maggiore leggibilità.

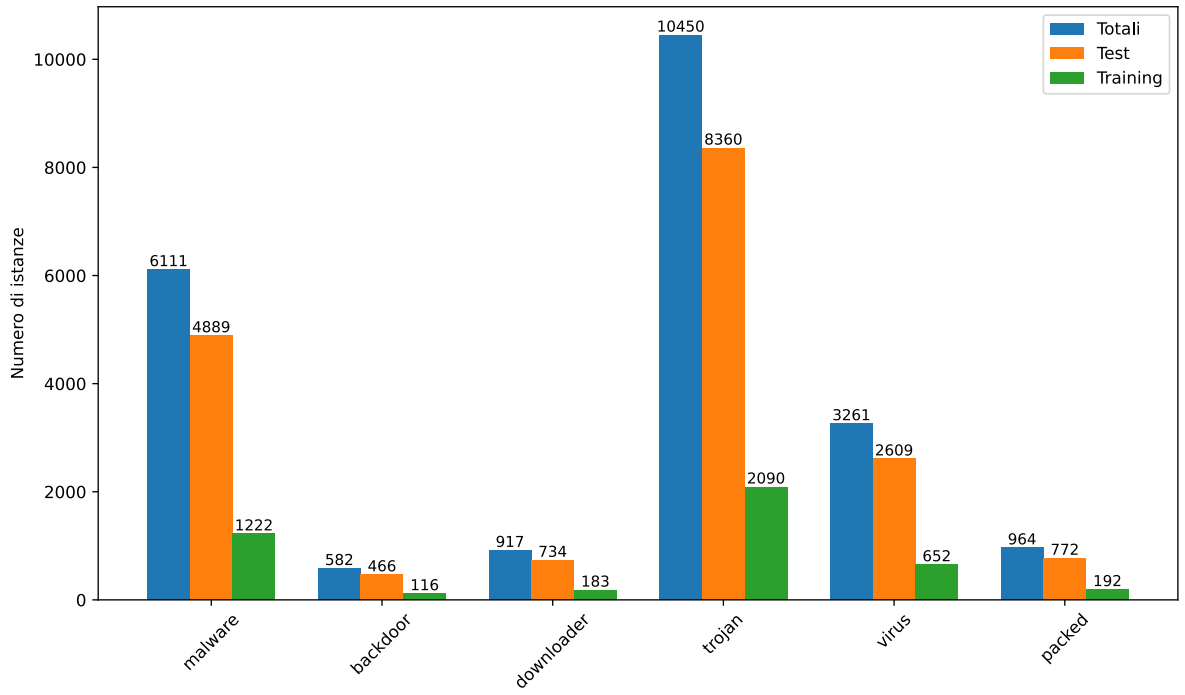

Figura 4.2: Distribuzione delle classi sul dataset **apimds** per training set e test set

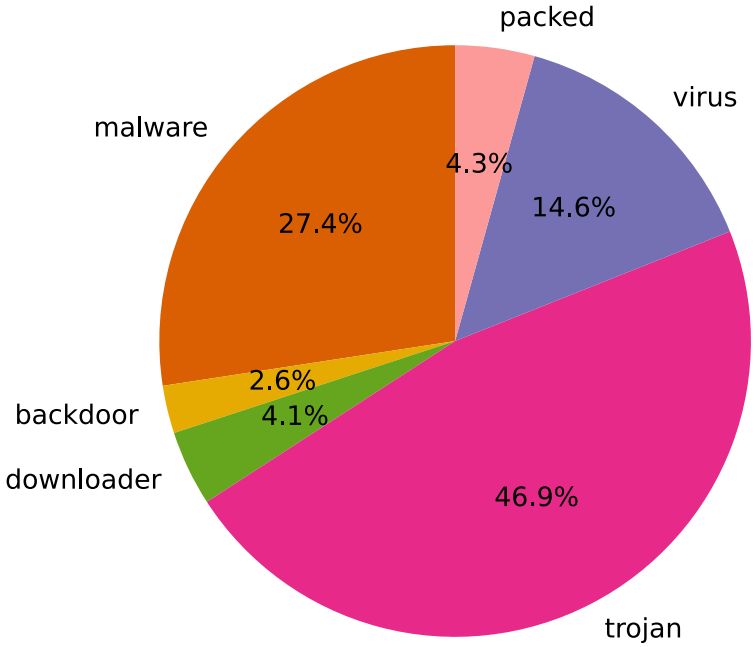

Figura 4.3: Distribuzione delle classi sul dataset **apimds**

Il dataset **octak** è disponibile su GitHub [60] ed è distribuito in formato tabulare su due tabelle. La prima tabella, dove ogni riga corrisponde un esempio, ha una sola feature che corrisponde alla classe dell'esempio. La seconda tabella, dove ogni riga corrisponde un esempio, ha una sola colonna in cui è presente la sequenza di chiamate API separate da spazio. In particolare, per l'addestramento, si è utilizzata l'i-esima riga della prima tabella per ricavare la classe e l'i-esima riga della seconda tabella per ricavare la sequenza di chiamate API per l'i-esimo esempio. In Figura 4.4 un esempio del dataset. Il numero di esempi è pari a 7107, di cui 80% dedicato al training (56880) e il restante 20% al testing set (1419). La media di chiamate API per esempio è pari a 19647.267 e il numero di chiamate distinte è pari a 278. Le classi presenti sono *Backdoor* (14.2% - 1001), *Adware* (5.4% - 379), *Downloader* (14.2% - 1001), *Dropper* (12.6% - 891), *Spyware* (11.8% - 832), *Trojan* (14.2% - 1001), *Virus* (14.2% - 1001), *Worm* (13.6% - 972). Non sono presenti istanze di *Goodware*. In Figura 4.5 e Figura 4.6 vengono riportati le distribuzioni delle classi del dataset.

```
Trojan
Backdoor
Trojan
Backdoor
Downloader
Worms
Spyware
```

```
ldrloaddll ldrgetprocedureaddress ldrloaddll
    ↰ ldrgetprocedureaddress
getsystemtimeasfiletime ntallocatevirtualmemory
    ↰ ntfreevirtualmemory
ldrgetdllhandle ldrgetprocedureaddress
    ↰ getsystemdirectorya copyfilea
ldrloaddll ldrgetprocedureaddress ldrloaddll
    ↰ ldrgetprocedureaddress
ldrloaddll ldrgetprocedureaddress ldrgetprocedureaddress
ntprotectvirtualmemory ntprotectvirtualmemory
    ↰ ntprotectvirtualmemory
ldrloaddll ldrgetprocedureaddress ldrgetprocedureaddress
```

Figura 4.4: Una porzione del dataset **octack** troncato per maggior leggibilità.

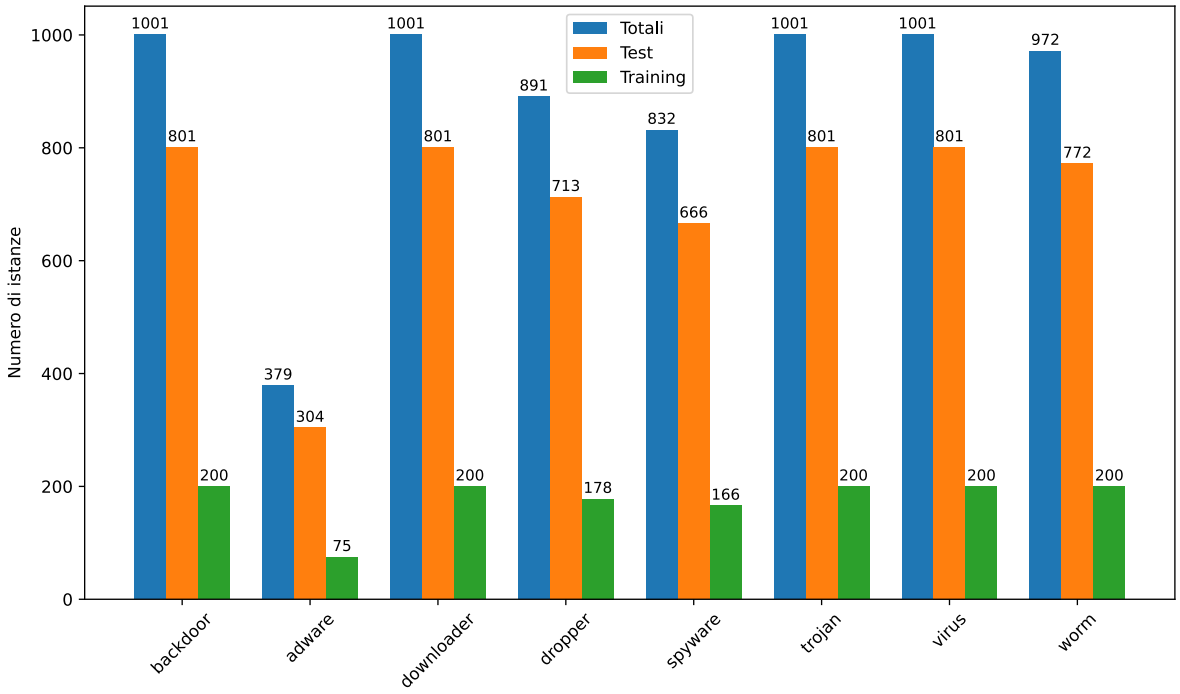

Figura 4.5: Distribuzione delle classi sul dataset **octak** per training set e test set

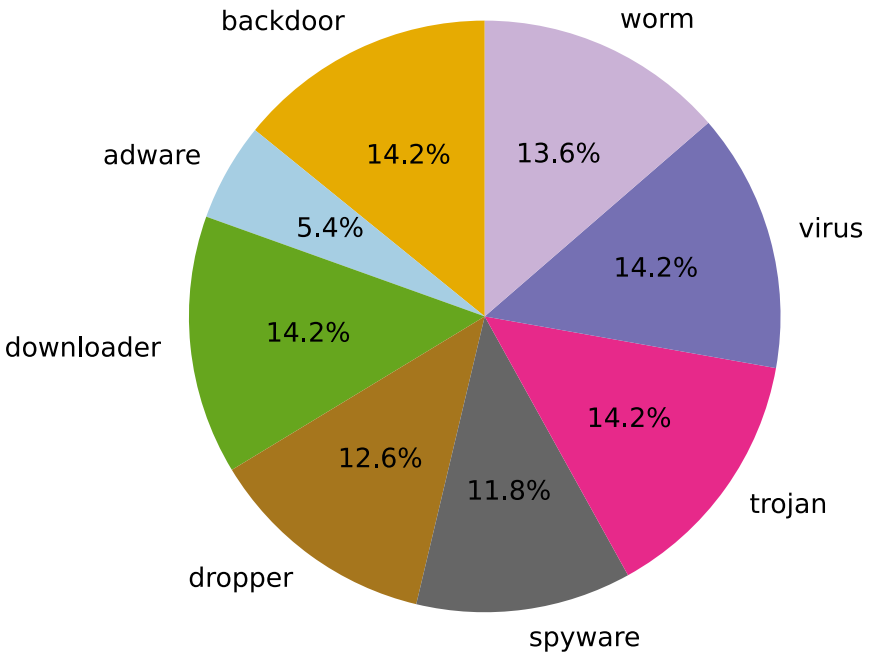

Figura 4.6: Distribuzione delle classi sul dataset **octak**

Il dataset **mpasco** è disponibile su GitHub [61], distribuito in formato tabulare, la prima riga è l'intestazione della tabella (sha265, labels, e una sequenza incrementale che indica la posizione della chiamata API nella sequenza) e le successive righe gli esempi.

Per l'addestramento le feature selezionate sono state la label (con valore 0 per *Goodware* altrimenti *Malware*), e le colonne della sequenza della chiamate API. In Figura 4.7 un esempio del dataset. Il numero di esempi è pari a 4112, di cui 80% dedicato al training (3598) e il restante 20% al testing set (514). La media di chiamate API per esempio è pari a 43.187 e il numero di chiamate distinte è pari a 291. Sono presenti solo due classi, *Malware* (50% - 2056) e *Goodware* (50% - 20560). In Figura 4.8 e Figura 4.9 vengono riportati le distribuzioni delle classi del dataset.

```
1 sha256,labels,0,1,2,
2 5c18291c481a192ed5003084dab2d8a117fd3736359218fee2aea1a164544c9e,0,LdrUnloadD
    ↰ ll,CoUninitialize,NtQueryKey
3 fbc42993285c97038894aa945b871caa9fc1030190b2c03c54b9d7ad2d799347,1,NtQueryVal
    ↰ ueKey,LdrUnloadDll,RegCloseKey
```

Figura 4.7: Una porzione del dataset **mpasco** troncato per una maggiore leggibilità.

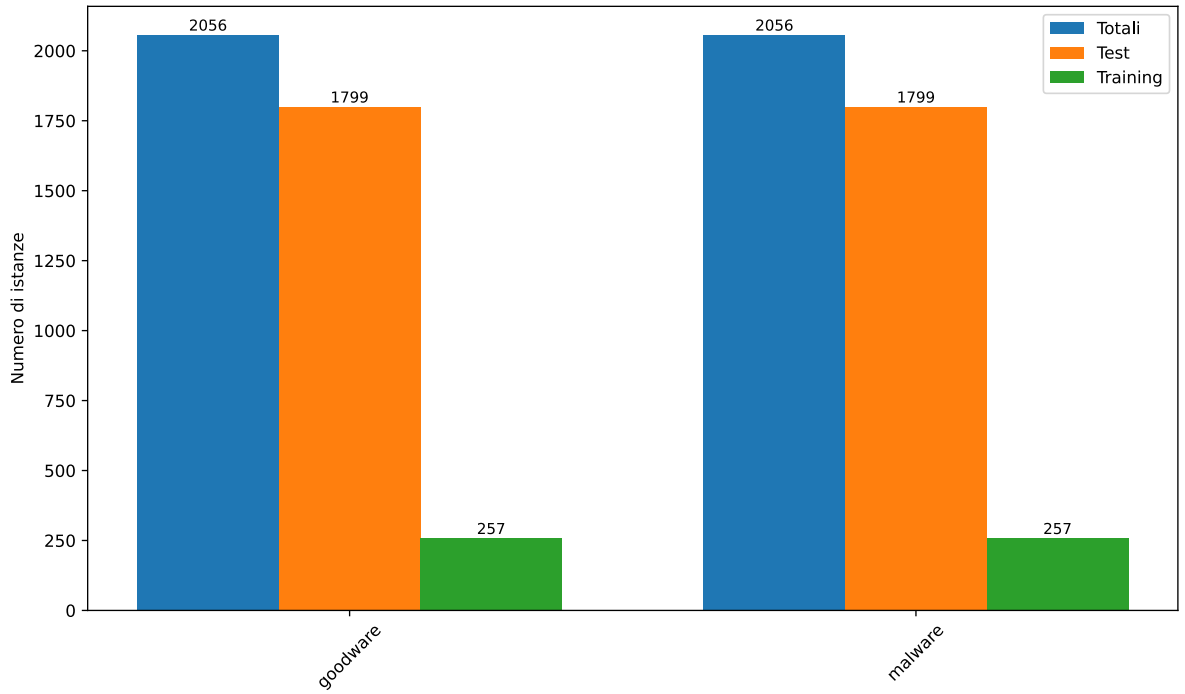

Figura 4.8: Distribuzione delle classi sul dataset **mpasco** per training set e test set

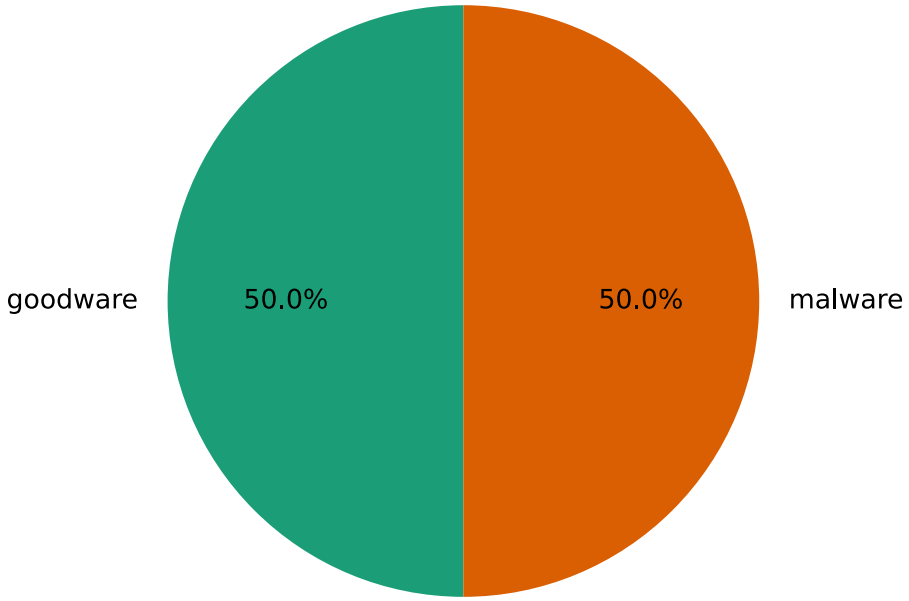

Figura 4.9: Distribuzione delle classi sul dataset **mpasco**

Il dataset **quovadis** è disponibile su GitHub [62] ed è distribuito come un archivio compresso. Nell'archivio sono presenti diverse cartelle che rispettano la denominazione `report_<classe>`, dove `classe` è la classe delle istanze degli esempi presenti. In una singola cartella di classe, sono presenti molteplici file JSON con la denominazione `<hash PE>.json`. I campi di interesse del JSON per la standardizzazione del dataset sono riportati in tabella:

Key	**Descrizione**
`[].apis`	Un array di oggetti JSON rappresentanti le chiamate API in sequenza
`[].apis[].api_name`	La chiamata API

In Figura 4.10 un esempio del dataset. Il numero di esempi è pari a 64023, di cui 80% dedicato al training (51219) e il restante 20% al testing set (12804). La media di chiamate API per esempio è pari a 196.144 e il numero di chiamate distinte è pari a 3024. Sono presenti tre classi, *Malware* (43.2% - 27670), *Goodware* (39.5% - 25291) e *Backdoor* (17.3% - 11062). Nella classe malware sono racchiusi tutti i malware non specializzati. In Figura 4.11 e Figura 4.12 vengono riportati le distribuzioni delle classi del dataset.

```
[
    {
        "apis": [
            {
                "api_name": "KERNEL32.GetSystemTimeAsFileTime"
            },
            {
                "api_name": "KERNEL32.GetCurrentThreadId"
            },
            {
                "api_name": "KERNEL32.GetCurrentProcessId"
            },
            {
                "api_name": "KERNEL32.QueryPerformanceCounter"
            }
        ]
    }
]
```

Figura 4.10: Una porzione del dataset **quovadis** troncato per una maggiore leggibilità.

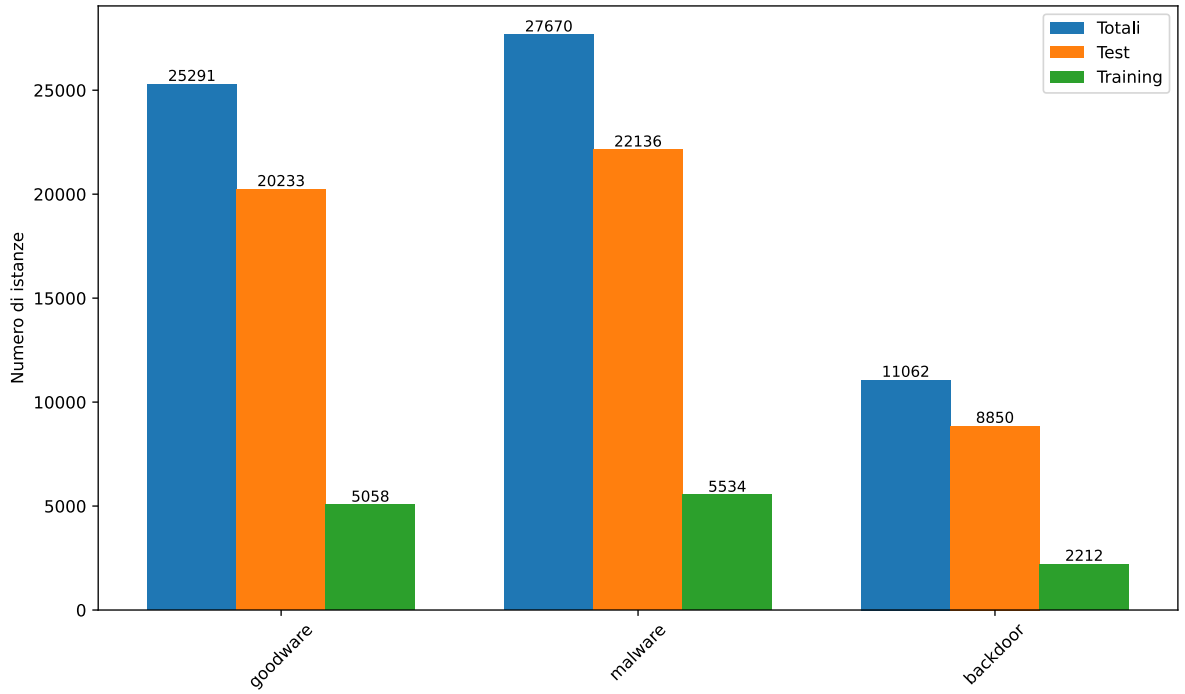

Figura 4.11: Distribuzione delle classi sul dataset **quovadis** per training set e test set

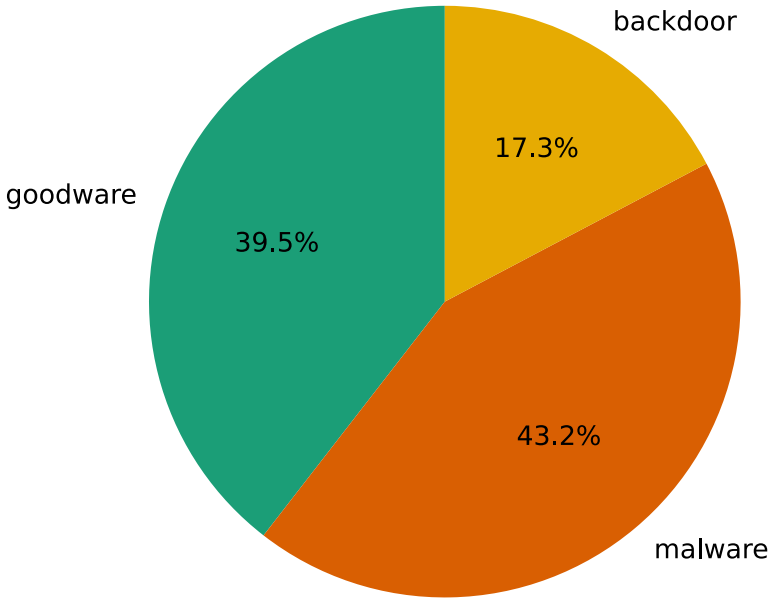

Figura 4.12: Distribuzione delle classi sul dataset **quovadis**

In Tabella 4.1 la tabella riassuntiva dei 4 dataset.

Dataset	Classificazione	Numero feature	Esempi train	Esempi test
apimds	Multiclasse	1165	18519	4627
octak	Multiclasse	278	56880	1419
mpasco	Binaria	291	3598	514
quovadis	Multiclasse	3024	51219	12804

Tabella 4.1: Caratteristiche dei dataset utilizzati.

4.2 Metriche di Valutazione

Per valutare l'efficacia di un classificatore è necessario quantificare quanto esso sia capace di effettuare previsioni corrette su dati diversi da quelli con cui è stato appreso. Uno strumento utile a questo scopo è la **matrice di confusione** [63]. Nella matrice, le righe rappresentano i valori effettivi mentre le colonne indicano quelli predetti; in letteratura è possibile incontrare anche la rappresentazione inversa. La matrice ha tante righe e colonne quante sono le classi presenti nel dataset. L'intersezione tra la riga i e la colonna j fornisce il numero di istanze della classe i che sono state classificate come classe j. In particolare, gli elementi della diagonale rappresentano le istanze correttamente predette. A partire dalla matrice di confusione è possibile calcolare diverse metriche di valutazione, tra cui la **precision**, la **recall** e l'**F1-score** per ciascuna classe. È inoltre possibile derivare le corrispondenti misure aggregate a livello *macro* e *micro*, utili per valutazioni globali sul dataset. Prima di entrare nel dettaglio delle metriche tornano utili i concetti di **true positive**, **true negative**, **false positive** e **false negative** considerando la matrice di confusione M avente n classi.

I **true positive** di una classe k equivalgono al numero di istanze di classe k che sono state predette come classe k:

$$TP_k = M_{k,k}$$

I **false positive** di una classe k rappresentano il numero di istanze appartenenti a classi diverse da k che sono state erroneamente predette come k:

$$FP_k = \sum_{\substack{i=1 \\ i \neq k}}^{n} M_{i,k}$$

I **false negative** di una classe k rappresentano il numero di istanze appartenenti a k che sono state erroneamente classificate come un'altra classe:

$$FN_k = \sum_{\substack{j=1 \\ j \neq k}}^{n} M_{k,j}$$

I **true negative** di una classe k rappresentano il numero di istanze appartenenti a classi diverse da k che sono state correttamente classificate come non k:

$$TN_k = \sum_{\substack{i=1 \\ i \neq k}}^{n} \sum_{\substack{j=1 \\ j \neq k}}^{n} M_{i,j}$$

La **precision** di una classe k è il rapporto tra i veri positivi e la somma dei veri positivi e dei falsi positivi:

$$\text{Precision}_k = \frac{TP_k}{TP_k + FP_k}$$

La **recall** (o sensibilità) di una classe k è il rapporto tra i veri positivi e la somma dei veri positivi e dei falsi negativi:

$$\text{Recall}_k = \frac{TP_k}{TP_k + FN_k}$$

L'**F1-score** di una classe k è la media armonica tra precision e recall:

$$\text{F1}_k = 2 \cdot \frac{\text{Precision}_k \cdot \text{Recall}_k}{\text{Precision}_k + \text{Recall}_k}$$

Le metriche **Precision**, **Recall** e **F1-Score** calcolate per ciascuna classe possono essere aggregate per ottenere valori complessivi sul modello intero. Due approcci comuni di aggregazione sono la media *macro* e la media *micro*:

$$\text{Macro}_{\text{precision}} = \frac{\sum_i^n Precision_i}{n}$$

$$\text{Macro}_{\text{recall}} = \frac{\sum_i^n Recalli}{n}$$

$$\text{Macro}_{\text{F1}} = \frac{\sum_i^n F1i}{n}$$

$$\text{Micro}_{\text{Precision}} = \frac{\sum_i^n TP_i}{\sum_i^n TP_i + \sum_i^n FP_i}$$

$$\text{Micro}_{\text{Recall}} = \frac{\sum_i^n TP_i}{\sum_i^n TP_i + \sum_i^n FN_i}$$

$$TP = \sum_{i=1}^{n} TP_i, \qquad FP = \sum_{i=1}^{n} FP_i, \qquad FN = \sum_{i=1}^{n} FN_i,$$

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN},$$

$$\text{Micro-F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Una metrica che riassume la prestazione globale è la *overall accuracy*:

$$\text{OverallAccuracy} = \frac{\sum_{i}^{n} M_{i,i}}{\sum_{i=1}^{n} \sum_{j=1}^{n} M_{i,j}}$$

4.3 Risultati

Per analizzare in dettaglio il comportamento dei modelli sulle diverse classi, sono state calcolate le matrici di confusione. Esse permettono di visualizzare le corrette classificazioni (diagonale principale) e gli errori di classificazione. Le matrici di confusione ottenute per il dataset **apimds** sono illustrate in Figura 4.13.

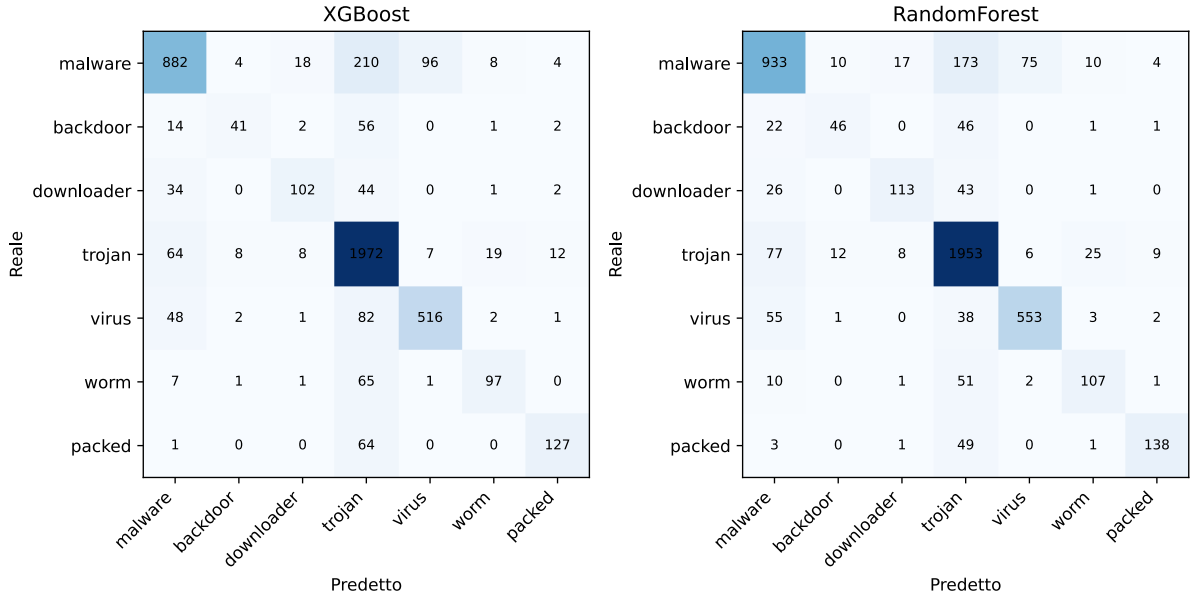

Figura 4.13: Matrici di confusione sul dataset **apimds**

Le classi meglio identificate per il modello *XGBoost* in ordine sono *Trojan*, *Malware* e *Virus*; lo stesso vale anche per il modello *RandomForest*. Queste classi coincidono con quelle più rappresentate nel dataset, come riportato in Figura 4.2, suggerendo che la distribuzione dei dati influisce in modo significativo sulla capacità di apprendimento del modello. Si osservano buone prestazioni anche sulle classi minoritarie: ciò suggerisce che tali categorie di malware facciano uso di chiamate API specifiche e scarsamente sovrapposte a quelle delle altre classi. In generale, quando il modello predice erroneamente, tende a classificare i campioni come appartenenti alla classe *Trojan*. Nel complesso, i due classificatori mostrano un comportamento simile, ma *RandomForest* ottiene risultati leggermente migliori.

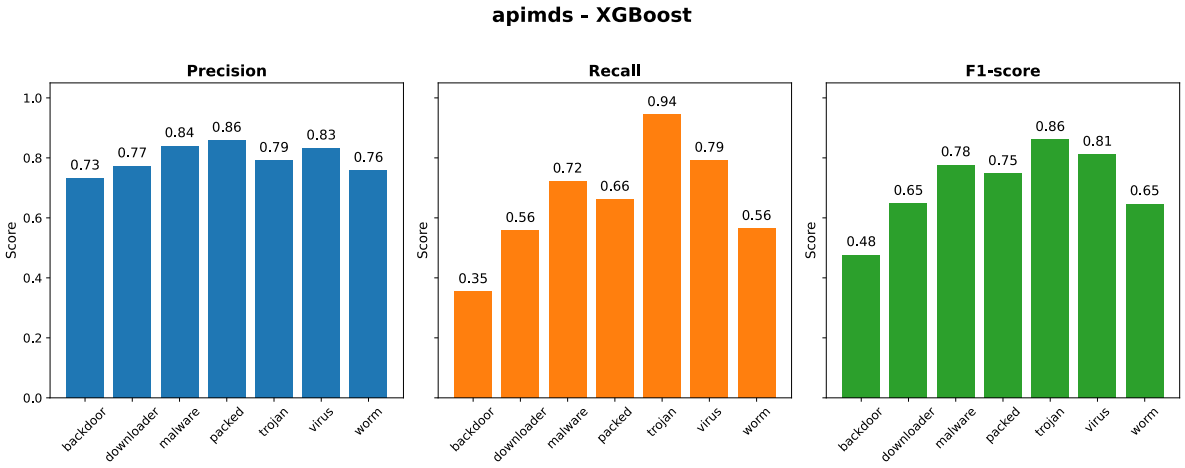

Figura 4.14: *Precision*, *Recall* e *F1-score* sul dataset **apimds** con modello *XGBoost*

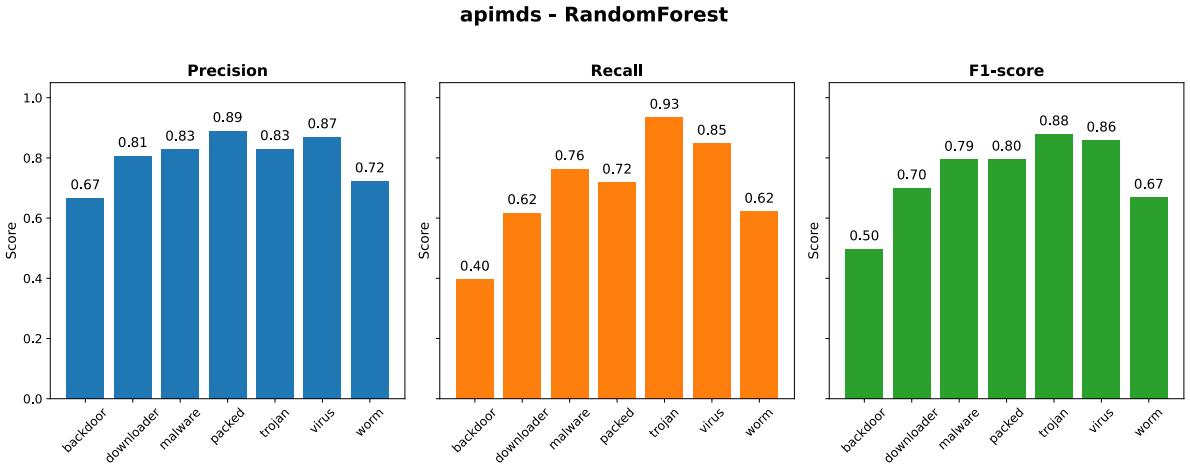

Figura 4.15: *Precision*, *Recall* e *F1-score* sul dataset **apimds** con modello *RandomForest*

In Figura 4.14 vengono illustrate le metriche *Precision*, *Recall* e *F1-score* per il modello *XGBoost*. Per quanto riguarda la *Precision*, il modello presenta valori generalmente buoni, con la maggior parte delle classi che supera il punteggio di 0.73. Questo risultato indica un basso tasso di falsi positivi: osservando le colonne della matrice di confusione in Figura 4.13, si nota infatti che pochi esempi appartenenti ad altre classi vengono erroneamente etichettati come *Backdoor* o altre categorie minoritarie. Relativamente alla *Recall*, si notano valori più variabili: il modello eccelle nella classificazione di *Trojan*, ottenendo un punteggio di 0.94. Si evidenziano, tuttavia, criticità per le classi *Backdoor*, *Downloader* e *Worm*, che registrano valori di *Recall* marcatamente bassi (0.35, 0.56 e 0.56). Analizzando la matrice di confusione, si individua la causa specifica di tale calo per la classe *Backdoor*: il modello fallisce nel riconoscere più della metà degli esempi reali. Su un totale di 116 campioni di *Backdoor*, ben 56 vengono erroneamente classificati come *Trojan*, superando addirittura il numero di predizioni corrette (41). Questo

errore sistematico, dovuto alla confusione con la classe maggioritaria, penalizza drasticamente la *Recall* e di conseguenza l'*F1-score* (0.48). Parallelamente, in Figura 4.15 sono mostrati i risultati per il modello *RandomForest*. Questo modello mantiene valori di *Precision* elevati, superando la soglia di 0.80 nella maggior parte delle classi (ad esempio, *Malware* 0.89 e *Trojan* 0.87). Tuttavia, il comportamento non è uniforme: mentre per alcune classi la precisione aumenta rispetto a *XGBoost*, per la classe *Backdoor* si osserva una diminuzione, scendendo da 0.73 a 0.67. Per quanto concerne la *Recall*, si osserva un trend analogo al modello precedente, con prestazioni eccellenti su *Trojan* (0.93) e *Virus* (0.85). Anche per *RandomForest* la classe *Backdoor* rappresenta il punto debole (0.40): la matrice di confusione conferma che 46 campioni vengono persi e classificati come *Trojan*. Ciononostante, si nota un lieve miglioramento nella capacità di recupero rispetto a *XGBoost*: la *Recall* sale a 0.40 contro il 0.35 del modello precedente. Anche le classi *Downloader* e *Worm* beneficiano di questo incremento, ottenendo entrambe 0.62 (rispetto al 0.56 di *XGBoost*). Dal confronto diretto dei due modelli emerge che *RandomForest* offre prestazioni complessivamente superiori su questo dataset. Analizzando l'*F1-score*, *RandomForest* ottiene un punteggio uguale o superiore a *XGBoost* in ogni singola classe. I vantaggi più evidenti si registrano nelle classi *Virus* (*F1-score* di 0.86 contro 0.81) e *Packed* (*F1-score* di 0.80 contro 0.75). Mentre entrambi i modelli eccellono nell'identificazione dei *Trojan* (*F1-score* 0.88 per entrambi), si registrano prestazioni critiche in termini di *F1-score* per la classe meno rappresentata, *Backdoor*, come evidenziato dalla distribuzione delle classi in Figura 4.3.

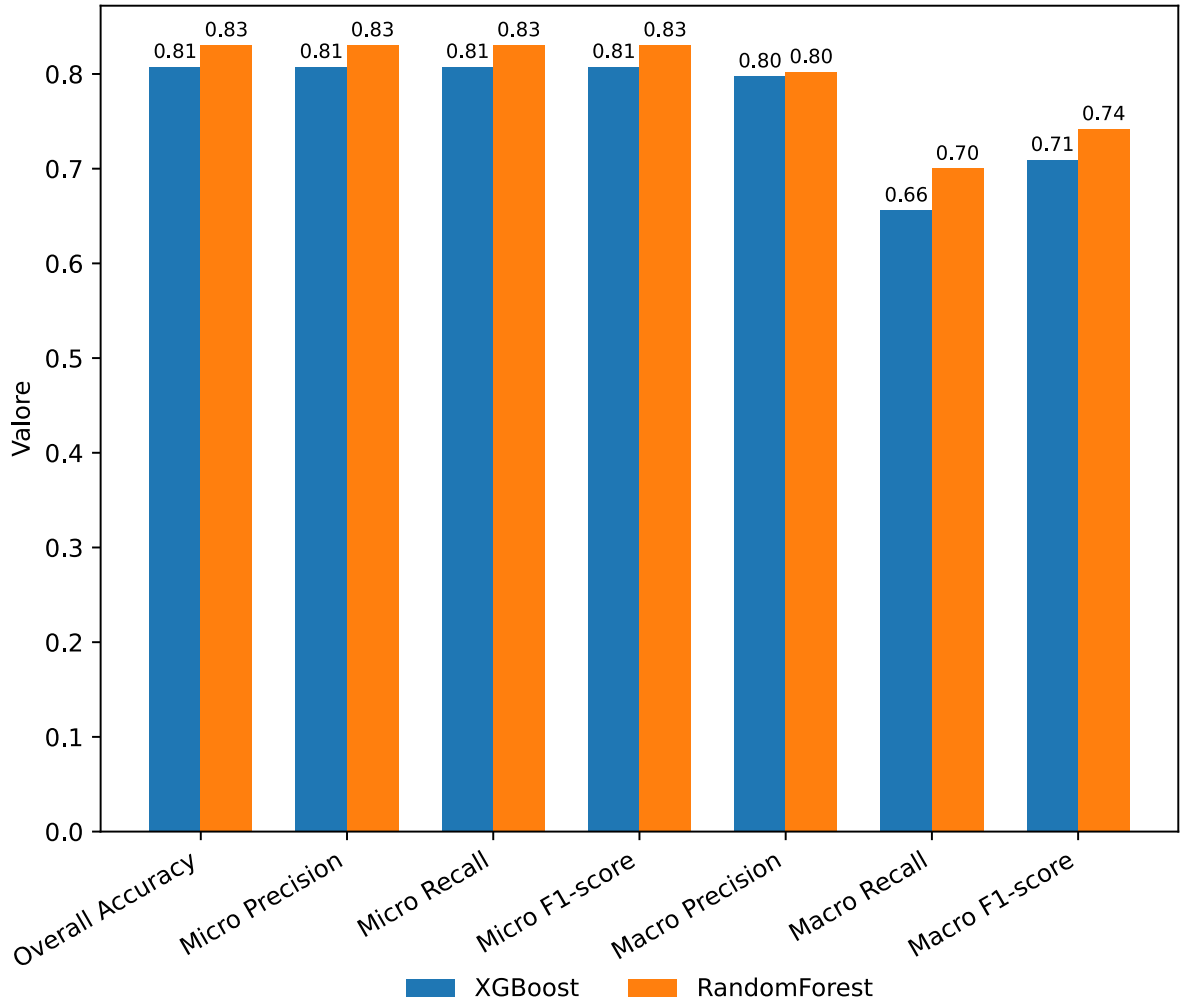

Figura 4.16: *Overall Accuracy*, *Micro* e *Macro* metriche sul dataset **apimds**

In Figura 4.16 sono riportate le metriche *Overall Accuracy*, *Micro Precision*, *Micro Recall*, *Micro F1-score*, *Micro Precision*, *Micro Recall*, *Micro F1-score* di entrambi i classificatori. L'analisi di queste metriche sul dataset rileva una chiara superiorità del classificatore *RandomForest* rispetto a *XGBoost*. Ad esempio in termini di *Overall Accuracy* il modello *RandomForest* ottiene un valore di 0.83 contro il 0.81 di *XGBoost*. In termini di *Macro F1-score*, che assegna la stessa importanza a tutte le classi, comprese quelle minoritarie e per questo più adatta rispetto alla *Overall Accuracy* per misurare le accuratezze di un modello in dati sbilanciati, il modello *RandomForest* ottiene uno score di 0.74 rispetto al 0.71 di *XGBoost*.

Le matrici di confusione ottenute per il dataset **octack** sono illustrate in Figura 4.17.

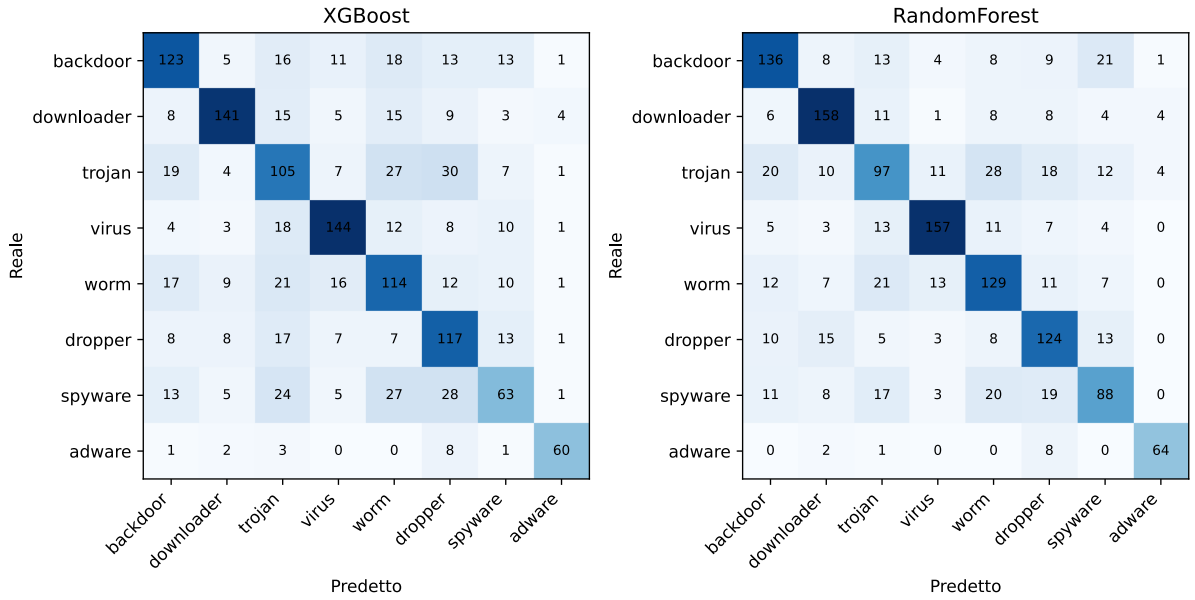

Figura 4.17: Matrici di confusione sul dataset **octack**

Le matrici di confusione mostrano che le classi più riconosciute per il modello *XGBoost*, in ordine sono *Virus*, *Downloader* e *Backdoor*; per il modello *RandomForest* risultano *Downloader*, *Virus* e *Backdoor*. Inoltre, a differenza del modello *XGBoost*, ottiene ottimi risultati anche per le classe *Spyware*. Tuttavia, anche in questo caso, come nel dataset precedente, si osserva una tendenza comune dei classificatori ad etichettare erroneamente come *Trojan* diversi esempi appartenenti in realtà ad altre classi. La classe *Adware* risulta invece ben distinta dalla altre, con pochissimi falsi positivi e falsi negativi, indice di chiamate API non presenti in altre classi. Entrambi i modelli mostrano un comportamento simile, ma *RandomForest* ottiene un numero maggiore di veri positivi complessivi, segno di una migliore capacità di classificazione.

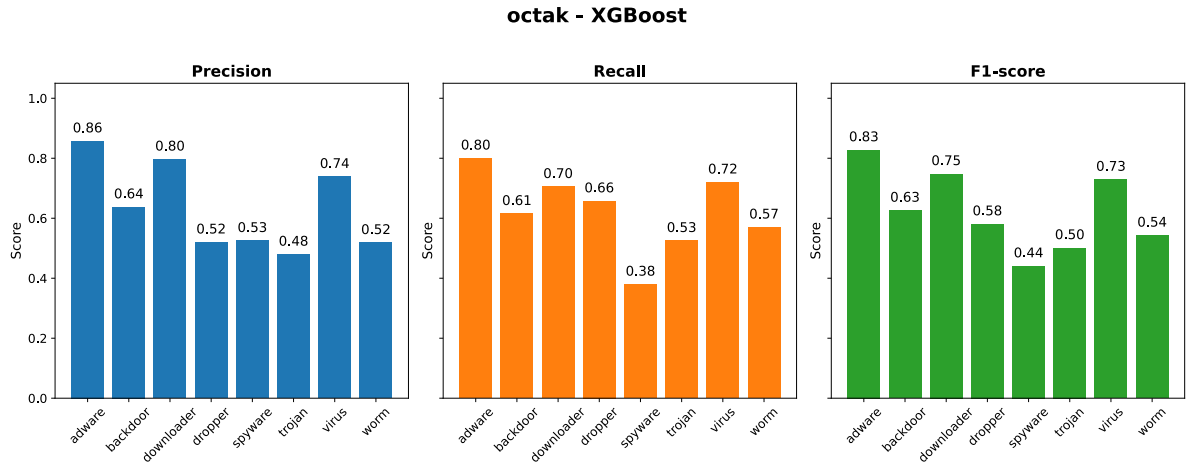

Figura 4.18: *Precision*, *Recall* e *F1-score* sul dataset **octack** con modello *XGBoost*

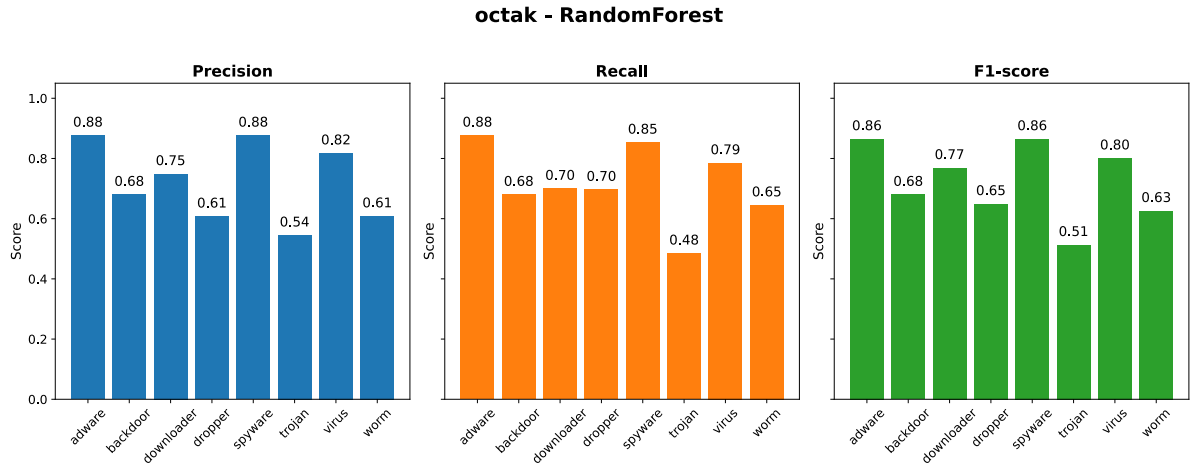

Figura 4.19: *Precision*, *Recall* e *F1-score* sul dataset **octack** con modello *RandomForest*

Il modello *XGBoost* in Figura 4.18 mostra prestazioni molto disomogenee. Ottiene ottimi risultati sulla classe *Adware* (*F1-score* 0.83), e buone prestazioni su *Backdoor* (*F1-score* 0.75) e *Virus* (*F1-score* 0.73). Tuttavia, dimostra significative difficoltà su più classi. La classe *Spyware* rappresenta il punto critico maggiore, registrando sia una *Precision* bassa (0.53) che una *Recall* insufficiente (0.38). Osservando la matrice di confusione, si nota che il modello fatica a distinguere questa classe: da un lato classifica erroneamente molti campioni di altre classi come *Spyware* (riducendo la precisione), dall'altro perde la maggior parte degli esempi reali di *Spyware*. Questi ultimi vengono erroneamente distribuiti tra diverse categorie, in particolare *Dropper* (28 errori), *Worm* (27) e *Trojan* (24); una criticità che si riflette sull'*F1-score*, il quale si attesta ad un valore di 0.44. Anche le classi *Trojan* (*F1-score* 0.50) e *Worm* (*F1-score* 0.54) registrano prestazioni mediocri, indicando una scarsa capacità di generalizzazione su queste categorie. Il modello *RandomForest* in Figura 4.19, al contrario, offre prestazioni notevolmente superiori. Le classi meglio classificate risultano *Adware* (*F1-score* 0.86), *Spyware* (*F1-score* 0.86) e *Virus* (*F1-score* 0.80). Tuttavia, la classe *Trojan* si conferma un punto debole per l'architettura, registrando un *F1-score* di 0.51, un valore sostanzialmente analogo a quello ottenuto da *XGBoost* (0.50). Questo indica che le caratteristiche della classe *Trojan* in questo dataset rimangono difficili da isolare per entrambi i classificatori. Dal confronto diretto sul dataset `octack` emerge comunque un chiaro vantaggio del modello *RandomForest*. Questo ottiene un *F1-score* superiore a *XGBoost* nella maggior parte delle classi, dimostrando una accuratezza e una capacità di generalizzazione nettamente migliori. Mentre *XGBoost* fallisce gravemente sulla classe *Spyware*, *RandomForest* riesce a gestirla con efficacia, spostando la criticità principale unicamente sulla classe *Trojan*. Nel complesso *RandomForest* tende a classificare meglio rispetto a *XGBoost*.

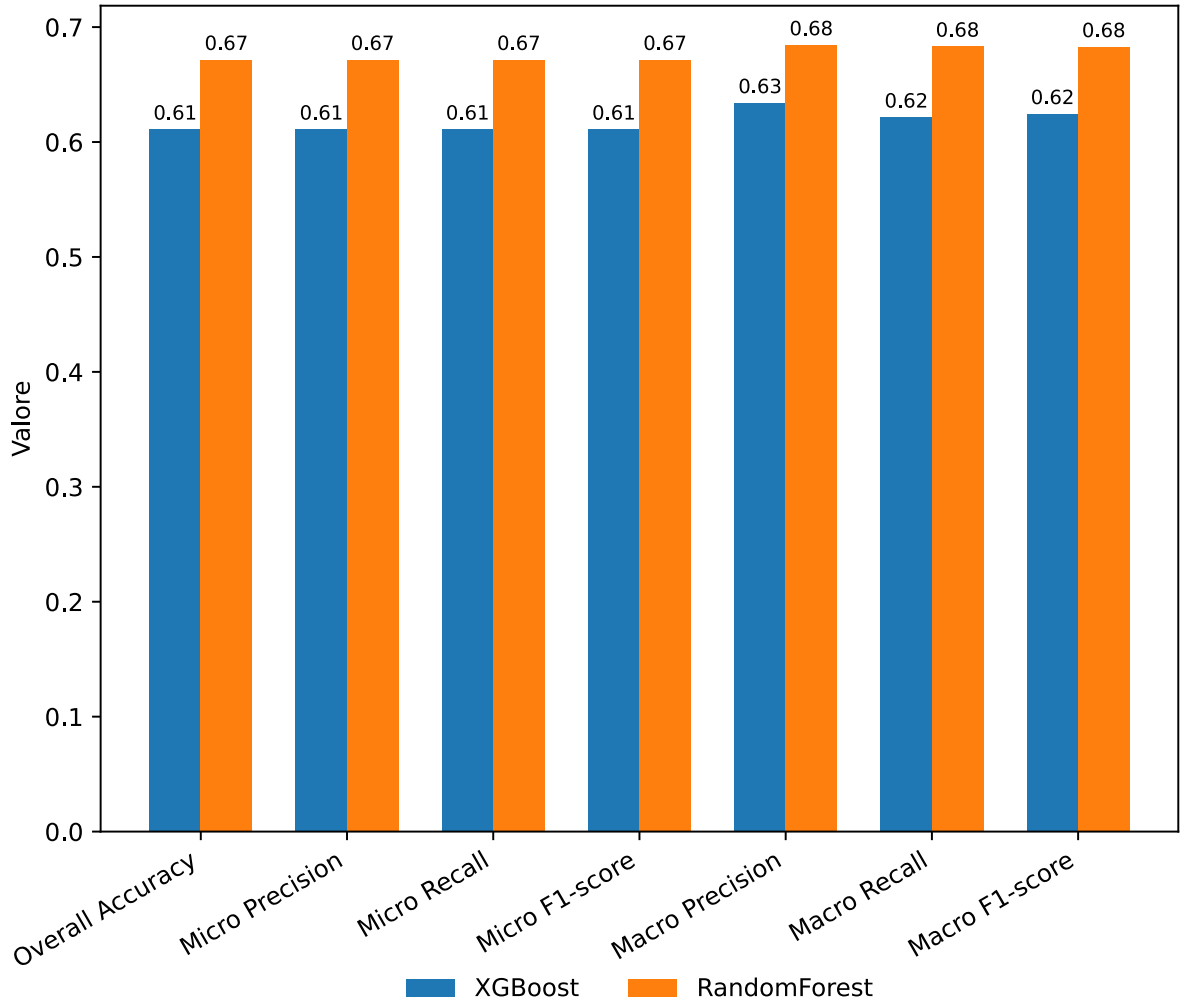

Figura 4.20: *Overall Accuracy*, *Micro* e *Macro* metriche sul dataset **octack**

Le metriche globali presenti in Figura 4.20 dimostrano come il classificatore *RandomForest* sia quello più ottimale ottendo prestazioni migliori in tutte le metriche.

Le matrici di confusione ottenute per il dataset **mpasco** sono illustrate in Figura 4.21.

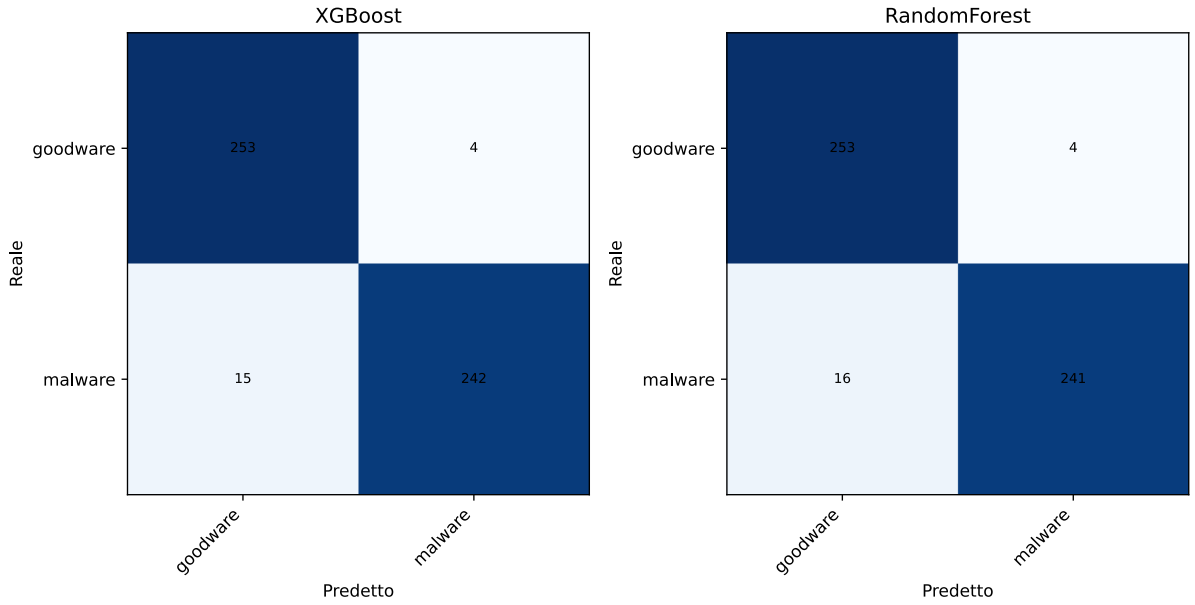

Figura 4.21: Matrici di confusione sul dataset **mpasco**

Il dataset **mpasco** presenta solo due classi, *Goodware* e *Malware*, ricadendo nel compito di classificazione binaria e distinguendosi dagli altri dataset analizzati per essere perfettamente bilanciato. I falsi negativi (*Malware* classificati come *Goodware*) risultano più numerosi rispetto ai falsi positivi (*Goodware* classificati come *Malware*). In un contesto di sicurezza informatica, tuttavia, sarebbe preferibile un comportamento che penalizzi maggiormente i falsi negativi: un *Malware* non rilevato ha conseguenze più gravi rispetto ad un *Goodware* non rilevato. Entrambi i modelli hanno ottime capacità di distinzione tra le due classi.

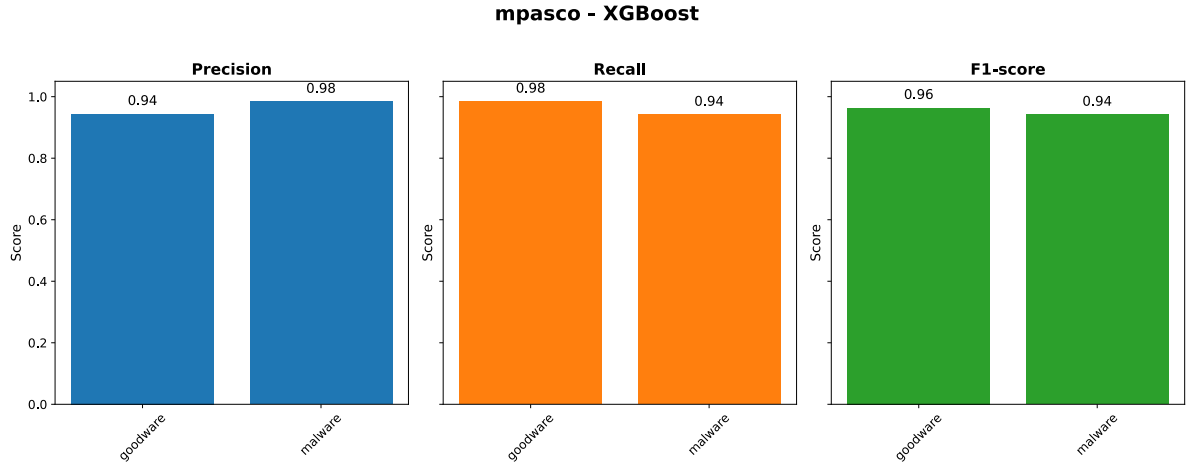

Figura 4.22: *Precision*, *Recall* e *F1-score* sul dataset **mpasco** con modello *XGBoost*

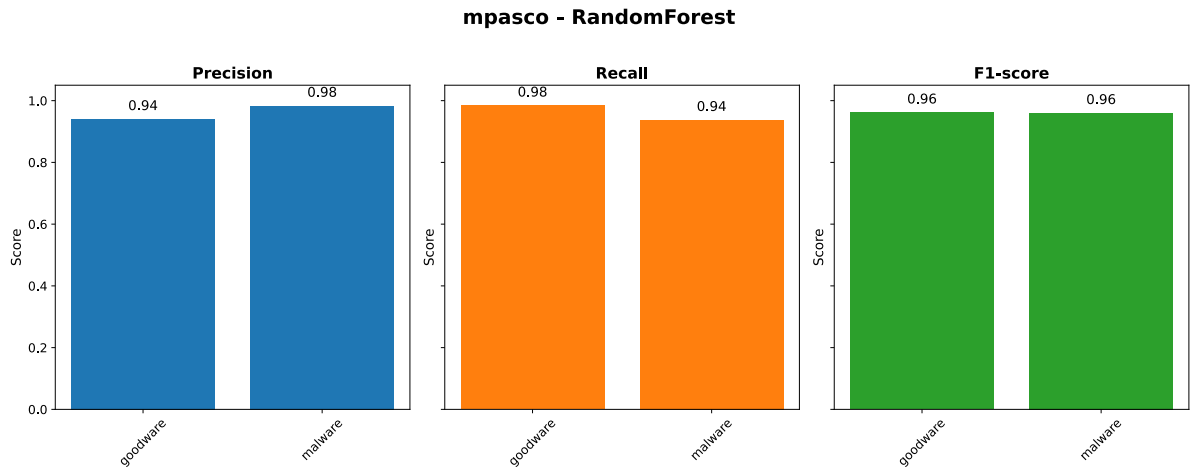

Figura 4.23: *Precision*, *Recall* e *F1-score* sul dataset **mpasco** con modello *RandomForest*

Nelle figure Figura 4.22 e Figura 4.23 si evidenzia l'ottima capacità dei modelli nel distinguere le due classi. Nello specifico, in termini di *F1-score*, il modello *XGBoost* raggiunge un valore di 0.96 sulla classe *Goodware* e di 0.94 sulla classe *Malware*. Il modello *RandomForest*, invece, mostra prestazioni leggermente superiori: mentre conferma lo stesso punteggio per la classe *Goodware* (0.96), migliora il risultato sulla classe *Malware*, portando l'*F1-score* a 0.96. Tale risultato indica che, su questo specifico dataset, il classificatore *RandomForest* è riuscito a massimizzare la combinazione di precisione e recupero in modo lievemente più efficace rispetto a *XGBoost*.

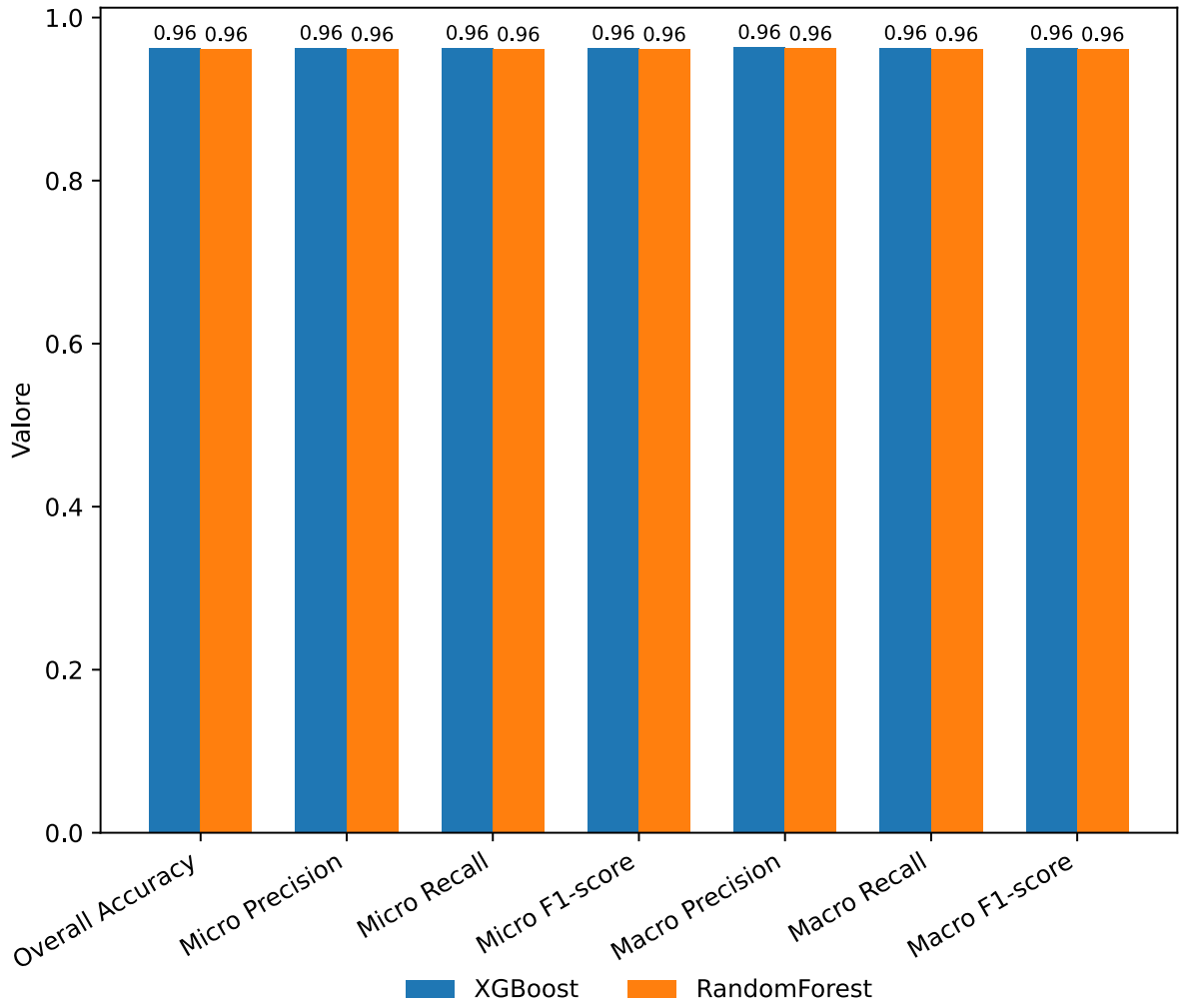

Figura 4.24: *Overall Accuracy*, *Micro* e *Macro* metriche sul dataset **mpasco**

I valori delle metriche globali illustrati in Figura 4.24 sintetizzano i risultati ottenuti per le singole classi. In particolare, essendo il dataset perfettamente bilanciato, non si osservano discrepanze significative tra l'*Overall Accuracy* e le metriche calcolate con media *Macro* o *Micro*: tutti gli indicatori convergono verso il valore di 0.96.

Le matrici di confusione ottenute per il dataset **quovadis** sono illustrate in Figura 4.25.

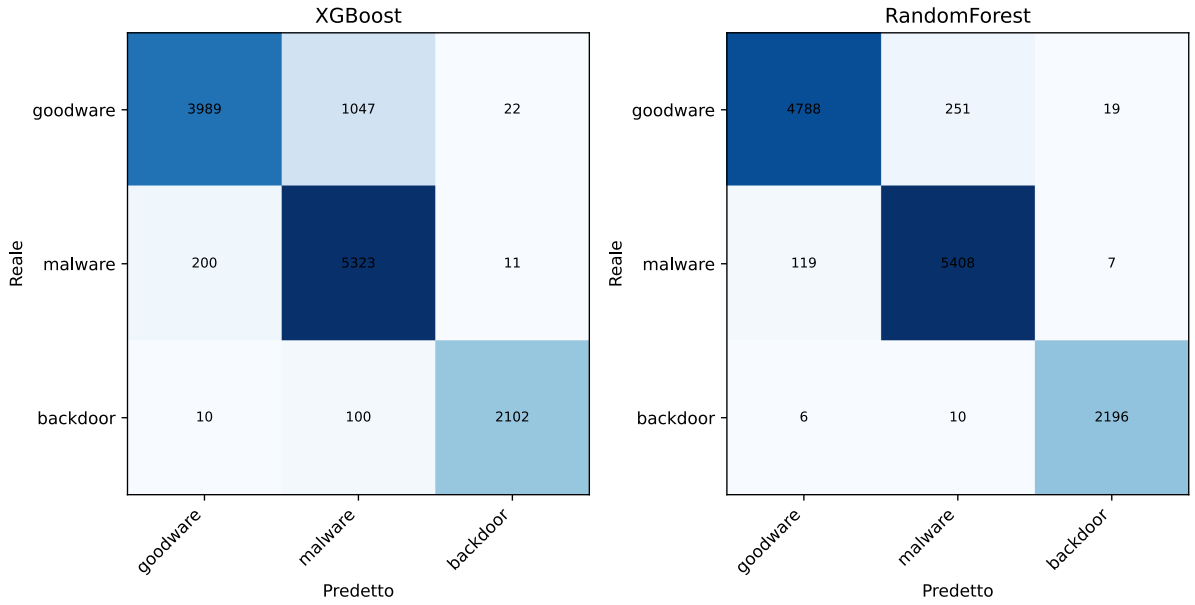

Figura 4.25: Matrici di confusione sul dataset **quovadis**

Il modello *XGBoost* mostra una buona capacità di classificare gli esempi della classe malware *Malware*, indicando un piccolo bias verso questa etichetta dovuto allo sbilanciamento del dataset. In linea generale, tale comportamento non sarebbe necessariamente negativo, poiché in un contesto di sicurezza è preferibile classificare erroneamente un file legittimo come sospetto piuttosto che il contrario. Tuttavia, in questo caso, tale bias non porta benefici: *XGBoost* tende infatti a confondere un numero maggiore di *Malware* e *Backdoor* come *Goodware* rispetto a *RandomForest*. Questo implica che il modello finisce per sottostimare la presenza di file malevoli, aumentando i falsi negativi e riducendo quindi l'efficacia complessiva nella rilevazione delle minacce.

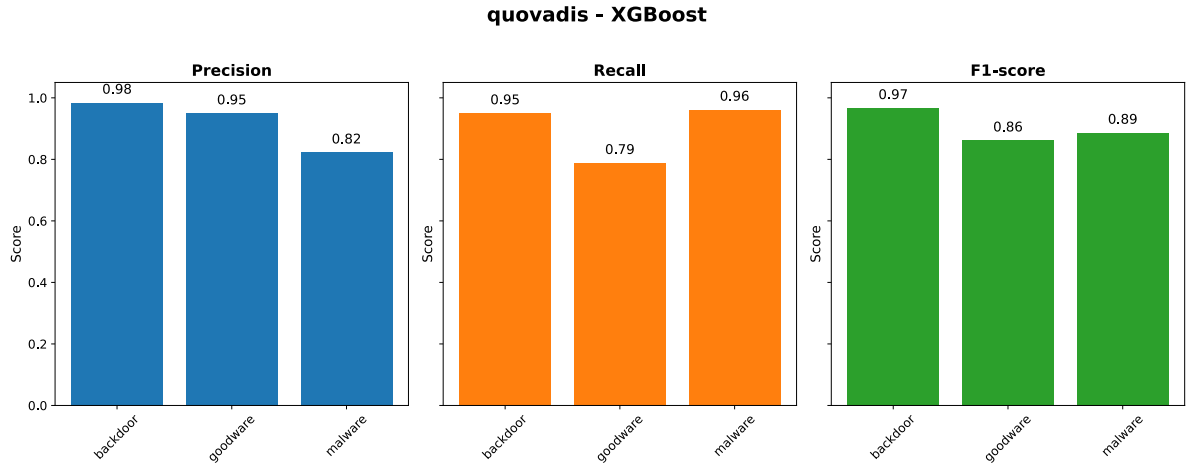

Figura 4.26: *Precision*, *Recall* e *F1-score* sul dataset **quovadis** con modello *XGBoost*

In Figura 4.26 sono riportate le metriche di valutazione per il modello *XGBoost* sul

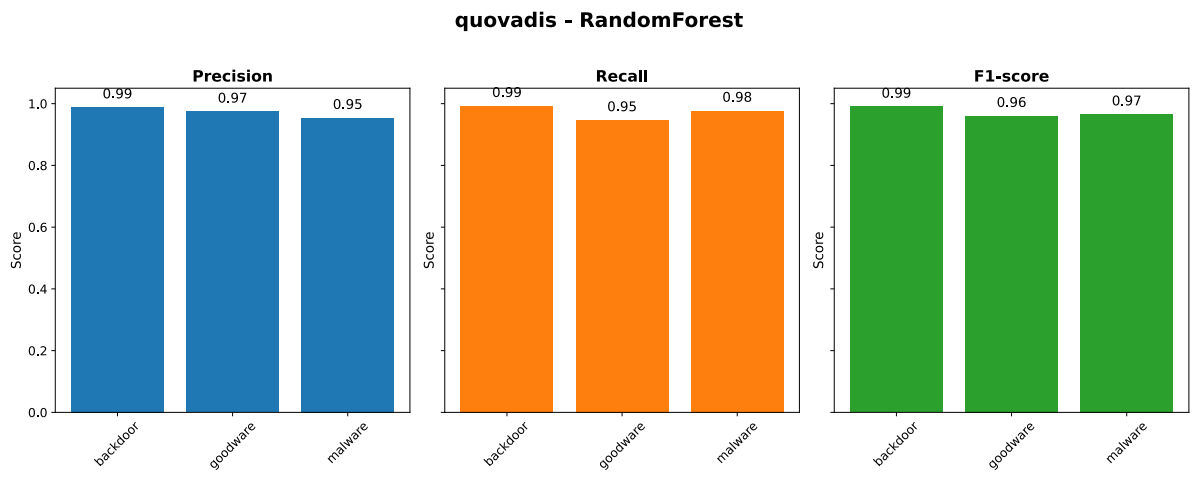

Figura 4.27: *Precision*, *Recall* e *F1-score* sul dataset **quovadis** con modello *RandomForest*

dataset `quovadis`. Il modello dimostra prestazioni eccellenti sulla classe *Backdoor*, raggiungendo nelle metriche *Precision*, *Recall* e *F1-score* valori superiori o uguali a 0.95. Sulla classe *Goodware* nonostante si abbia una *Precision* elevata (0.95), la *Recall* è pari a 0.79 indicando che il modello fallisce nell'identificare oltre il 20% dei campioni legitimi. Per la classe *Malware*, il valore di *Precision* è il più basso (0.82), suggerendo un tasso più elevato di falsi positivi rispetto alle altre due classi. Il modello *RandomForest*, illustrato in Figura 4.27, mostra invece prestazioni più alte rispetto all' *XGBoost* e più uniformi. Si osservi inoltre il notevole incremento prestazionale sulla classe *Malware*: se con *XGBoost* l'*F1-score* si attestava a 0.89, con *RandomForest* tale valore sale a 0.97. È interessante notare come, per entrambi i modelli, la classe *Goodware* registri i valori più bassi di *Recall* e *F1-score* rispetto alle altre classi. Analizzando la matrice di confusione, ciò indica che un numero significativo di campioni legittimi (*Goodware*) viene erroneamente classificato come malevolo, in particolare confondendolo con la classe generica *Malware*. Tuttavia, a differenza di *XGBoost* che subisce un crollo prestazionale su questa classe, *RandomForest* non mostra cali significativi, mantenendo tutte le metriche al di sopra della soglia di 0.95 per tutte le classi.

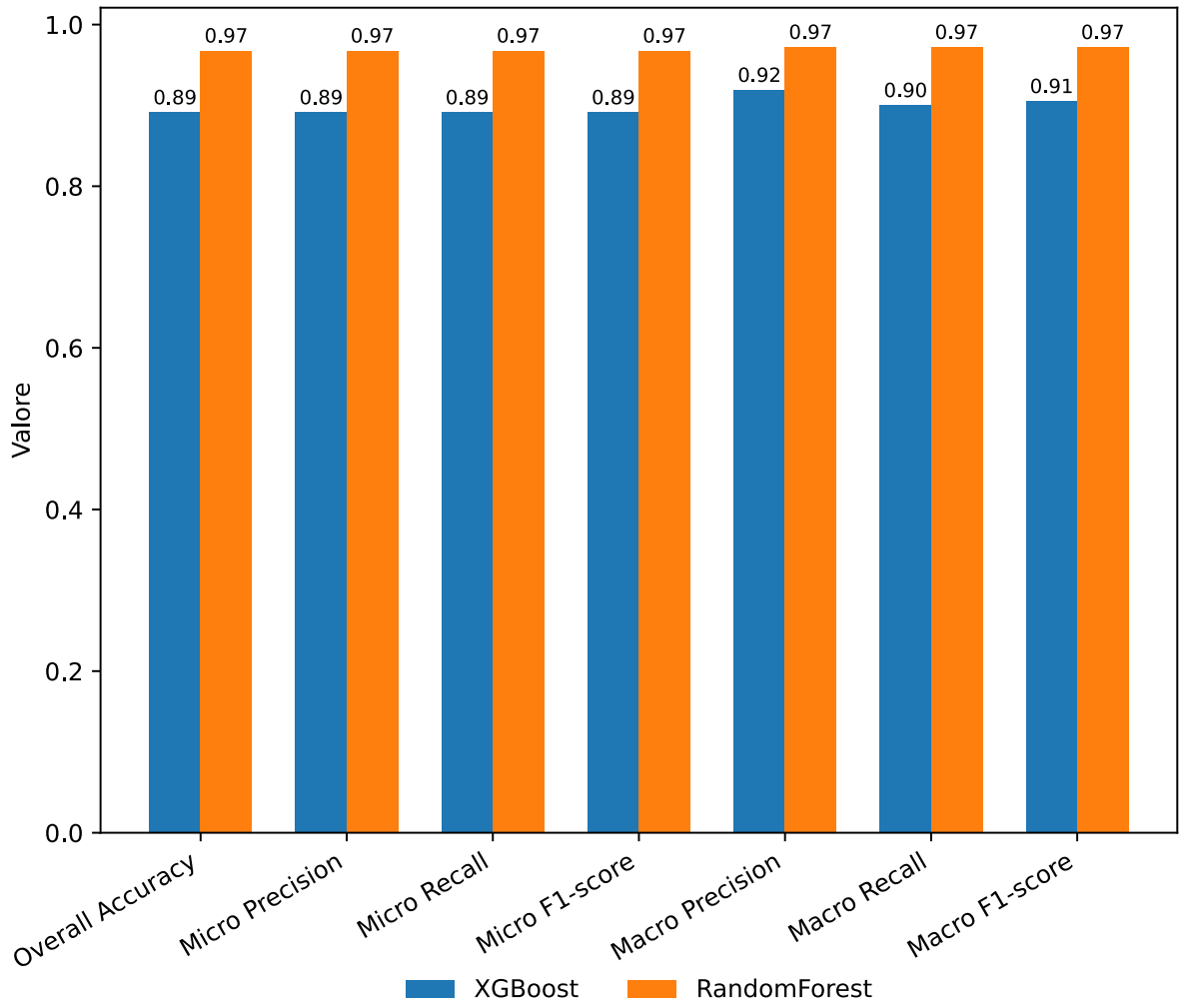

Figura 4.28: *Overall Accuracy*, *Micro* e *Macro* metriche sul dataset **quovadis**

Le metriche globali illustrate in Figura 4.28 rispecchiano le prestazioni sulle classi ribadendo che il classificatore *RandomForest* raggiunge prestazioni superiori a *XGBoost* in tutte le metriche calcolate.

In sintesi, l'analisi comparativa condotta sui quattro dataset evidenzia una maggiore efficacia del classificatore *RandomForest* rispetto a *XGBoost*. Sebbene entrambi i modelli abbiano raggiunto buone prestazioni, specialmente nella classificazione delle classi maggioritaria con qualche calo nella classificazione delle minoritarie, *RandomForest* ha dimostrato una superiore capacità di generalizzazione, garantendo una maggiore accuratezza e stabilità anche nelle classificazioni multiclasse più sbilanciate. Al contrario, *XGBoost* ha mostrato, in alcuni frangenti, una sensibilità maggiore alla distribuzione delle classi, introducendo bias verso le categorie maggioritarie. È rilevante notare notare che i risultati eccellenti sono stati ottenuti mantenendo le configurazioni di base degli algoritmi, ovvero utilizzando gli iperparametri predefiniti forniti dalla libreria *scikit-learn*. Sebbene i risultati conseguiti con la configurazione standard siano da considerarsi buoni, confermando la

validità delle feature estratte, un ulteriore processo di ottimizzazione degli iperparametri (*fine-tuning*) potrebbe portare a un incremento delle prestazioni. In particolare, tale affinamento risulterebbe utile sulle criticità delle classi minoritarie, permettendo ai modelli di specializzarsi maggiormente sui pattern meno frequenti e di migliorare il bilanciamento complessivo della classificazione.

Capitolo 5

Conclusioni e Sviluppi Futuri

In questo capitolo conclusivo vengono riassunti i risultati ottenuti dal lavoro di ricerca e sviluppo svolto, evidenziando il raggiungimento degli obbiettivi preposti. Successivamente, vengono delineate le possibili direzioni future per estendere e migliorare l'approccio proposto, con particolare attenzione all'integrazione dei dati e all'adozione di architetture di apprendimento più complesse.

5.1 Conclusioni

Il presente lavoro di tesi ha affrontato il problema della classificazione di *Malware* in ambiente *Windows* attraverso l'analisi dinamica e l'utilizzo di tecniche di Machine Learning. L'obiettivo principale era sviluppare un sistema in grado di distinguere non solo tra software benigno e malevolo, ma anche di categorizzare le diverse famiglie di *Malware*, basandosi sulle frequenze di chiamate API. L'approccio proposto ha previsto la standardizzazione di dataset eterogenei (*apimds*, *octack*, *mpasco*, *quovadis*) e l'implementazione dei classificatori *RandomForest* e *XGBoost*, sfruttando la rappresentazione *Bag-of-Words* per la trasformazione delle sequenze in vettori delle caratteristiche. La validazione empirica è stata condotta applicando i modelli ai diversi dataset e analizzando le matrici di confusione risultanti, dalle quali sono state calcolate le metriche per verificarne l'efficacia.

Dall'analisi dei risultati sperimentali, emerge l'efficacia dell'analisi dinamica: l'utilizzo delle sole frequenze delle chiamate API si è dimostrato sufficiente per ottenere buone accuratezze di classificazione, confermando l'ipotesi che i malware tendano a utilizzare sottoinsiemi specifici e ricorrenti di funzionalità del sistema operativo per raggiungere i propri scopi. Nel confronto diretto, *RandomForest* si è dimostrato complessivamente più accurato rispetto a *XGBoost*. Tale superiorità è risultata particolarmente evidente negli scenari multi-classe e in presenza di dataset sbilanciati (come *octack* e *apimds*), dove *RandomForest* ha garantito una migliore generalizzazione, mantenendo valori alti di *F1-score* anche sulle classi minoritarie, a differenza di *XGBoost*.

In sintesi, il sistema sviluppato ha dimostrato la capacità di identificare le minacce con precisione osservando il comportamento del software piuttosto che la sua struttura statica, superando così i limiti delle tecniche di offuscamento che affliggono l'analisi statica tradizionale.

5.2 Sviluppi Futuri

Un primo sviluppo naturale del lavoro svolto consiste nell'integrazione dei dataset analizzati in un unico corpus. Nel presente studio, sebbene i dati siano stati uniformati in un formato standard comune, le fasi di addestramento e validazione sono state condotte sui singoli dataset in modo indipendente. Una futura estensione prevede quindi la fusione effettiva di queste fonti con l'obiettivo di ottenere un **dataset monolitico**. Disporre di una base di dati così ampia e variegata permetterebbe di addestrare un modello "generalista", capace di riconoscere minacce provenienti da contesti molto diversi tra loro, simulando più fedelmente uno scenario reale in cui i malware presentano una grande variabilità. Tuttavia, l'unione di dataset eterogenei comporterebbe inevitabilmente un aumento esponenziale del numero di API distinte (il vocabolario), generando vettori delle caratteristiche di dimensioni enormi e contenenti per lo più valori nulli (sparsità). Gestire uno spazio così vasto può confondere algoritmi come *RandomForest* e *XGBoost*, riducendone l'efficienza e aumentando i tempi di calcolo. Per mitigare questo problema, sarà fondamentale implementare tecniche di **selezione delle caratteristiche** (*Feature Selection*) più avanzate. L'idea è quella di utilizzare metodi statistici (come l'*Information Gain* o la *PCA*) per identificare e mantenere solo quel sottoinsieme di API che è realmente discriminante per la classificazione, scartando tutto ciò che è rumore o ridondanza. Questo permetterebbe di mantenere il modello leggero e performante anche su scala molto più grande.

Infine, una linea di ricerca fondamentale riguarda il superamento dei limiti strutturali del modello *Bag-of-Words*. Come evidenziato dai risultati sperimentali, l'approccio basato sulle sole frequenze si è rivelato efficace per una classificazione ad alto livello, ma soffre di un limite intrinseco: la perdita dell'informazione sequenziale. Tale semplificazione impedisce di distinguere comportamenti che, pur utilizzando le stesse API, differiscono per l'ordine di esecuzione o per il contesto logico. A tal proposito, un naturale sviluppo del lavoro consiste nella validazione empirica sui dataset qui proposti delle metodologie avanzate già discusse nello Stato dell'Arte. In particolare, si suggerisce di confrontare l'attuale *baseline* con approcci che preservano la struttura temporale e semantica delle tracce. Una prima direzione prevede l'estensione al dominio multiclasse delle tecniche di analisi semantica e probabilistica introdotte da Amer e Zelinka[64]. Sebbene originariamente applicato alla classificazione binaria, questo approccio — che combina proiezioni vettoriali

(*Word2Vec*) con catene di Markov — offrirebbe un meccanismo robusto per modellare le transizioni di stato tra le chiamate API anche nel riconoscimento delle specifiche famiglie di malware. L'adattamento richiederebbe la costruzione di una matrice di transizione dedicata per ogni singola classe, e la classificazione finale verrebbe quindi determinata calcolando la verosimiglianza della sequenza osservata rispetto a ciascun modello e assegnando l'etichetta della classe che massimizza tale probabilità (*Maximum Likelihood Estimation*). Parallelamente, l'indagine dovrebbe estendersi verso il *Deep Learning* sequenziale, sperimentando reti neurali ricorrenti e convoluzionali (RNN/CNN) adattive, come nel modello di Gond e Mohapatra[44], capaci di gestire l'evoluzione temporale delle tracce e il fenomeno del *concept drift*. Infine, un'alternativa promettente risiede nella rappresentazione a grafo, modellando le sequenze come grafi direzionali elaborati tramite *Graph Convolutional Networks* (GCN) secondo la metodologia di Ma et al.[47]. L'obiettivo di tale estensione sarebbe verificare quantitativamente se la maggiore complessità computazionale richiesta da questi modelli si traduca in un incremento significativo delle prestazioni rispetto ai risultati già solidi ottenuti con *RandomForest*, specialmente nella rilevazione di varianti di malware complesse che eludono l'analisi frequenziale.

Capitolo 6

Appendice

Il presente capitolo raccoglie i contenuti tecnici e strumentali che, pur essendo funzionali alla realizzazione del progetto, sono stati separati dal corpo principale della tesi per preservarne la fluidità di lettura. Nello specifico, vengono forniti i dettagli relativi alla configurazione dell'ambiente di sviluppo e le convenzioni grafiche adottate per la modellazione UML, elementi essenziali per garantire la riproducibilità del lavoro e la corretta interpretazione degli schemi progettuali presentati.

6.1 Ambiente di Sviluppo

L'intero progetto è stato sviluppato con l'ausilio dell'editor di codice **Visual Studio Code** (VS Code) [65], usufruendo dell'integrazione offerta dall'estensione **Dev Containers** [66]. Tale estensione consente di utilizzare un container **Docker** [67] come ambiente di sviluppo completo, garantendo che il codice venga eseguito in un contesto isolato, coerente e perfettamente riproducibile indipendentemente dal sistema operativo della macchina ospitante.

La tecnologia Docker, su cui si basa questa architettura, introduce due concetti fondamentali:

- **Immagine Docker (Docker Image)**: È un modello (template) di sola lettura che contiene un set di istruzioni per creare un container. Essa include tutto il necessario per eseguire un'applicazione: il codice, il runtime, le librerie, le variabili d'ambiente e i file di configurazione. Nel contesto di questo progetto, l'immagine definisce il sistema operativo di base e le dipendenze Python necessarie.

- **Container Docker**: È l'istanza eseguibile di un'immagine. Rappresenta un ambiente isolato e leggero che condivide il kernel del sistema operativo host ma opera in uno spazio utente separato. Mentre l'immagine è statica, il container è dinamico e costituisce l'effettivo ambiente in cui avviene lo sviluppo e l'esecuzione del codice.

La configurazione dell'ambiente è centralizzata all'interno della directory `.devcontainer`, la quale ospita i file necessari all'orchestrazione del container. Il file `Dockerfile` definisce la costruzione dell'immagine personalizzata. Per questo progetto, si è partiti dall'immagine base ufficiale `python:3.13.5`. Oltre all'interprete Python e a strumenti essenziali come `git`, sono state installate a livello di sistema le dipendenze necessarie per il browser `chromium` e le relative librerie grafiche (come `libgbm1`, `libxrandr2`, ecc.). L'installazione di queste ultime si è resa necessaria per supportare le librerie di visualizzazione dati utilizzate nel progetto per il salvataggio e l'esportazione dei grafici generati.

Il file `devcontainer.json`, invece, orchestra l'integrazione con l'editor VS Code. Esso è stato configurato per:

- Automatizzare la costruzione dell'immagine basata sul `Dockerfile` citato.
- Eseguire comandi post-avvio (*postStartCommand*), automatizzando l'installazione delle dipendenze Python tramite il comando `make install`.
- Configurare l'editor, impostando il percorso dell'interprete Python e attivando strumenti di qualità del codice come il *linting* (`flake8`) e la formattazione automatica (`black`).
- Installare automaticamente le estensioni necessarie all'interno del container, tra cui `ms-python.python` per il supporto al linguaggio e `streetsidesoftware.code-spell-checker` per il controllo ortografico.

Ad ulteriore affinamento dell'ambiente, la directory `.vscode` contiene configurazioni specifiche per il workspace, come l'abilitazione del framework di testing `pytest` e l'aggiunta di termini specifici del dominio (es. "GOODWARE") al dizionario del correttore ortografico, evitando falsi positivi durante la stesura del codice e della documentazione.

L'adozione di questa architettura disaccoppia l'ambiente di sviluppo dall'hardware locale. Ciò rende il progetto immediatamente utilizzabile su qualsiasi macchina dotata di Docker in pochi semplici passaggi, eliminando le problematiche tipiche legate alle discordanze tra versioni di librerie o sistemi operativi differenti. Inoltre, grazie alla compatibilità nativa con lo standard Dev Container [68], il repository può essere aperto direttamente via browser tramite **GitHub Codespaces** [69], permettendo di sviluppare ed eseguire il codice in un ambiente cloud pre-configurato senza alcuna installazione locale.

6.2 Convenzioni e Notazione UML

Per la corretta interpretazione del Diagramma delle Classi presentato, definiamo le principali convenzioni di notazione UML adottate, relative alla struttura delle classi, alla visibilità e ai tipi di relazioni.

6.2.1 Classe & Visibilità

La classe è l'elemento fondamentale del Diagramma delle Classi. Come mostrato in Figura 6.1, è rappresentata da un rettangolo suddiviso in tre compartimenti che ne definiscono l'identità e il comportamento:

1. **Nome** (Superiore): Contiene il nome della classe.

2. **Attributi** (Centrale): Elenca le proprietà della classe.

3. **Metodi** (Inferiore): Elenca le operazioni della classe.

Il costrutto tra parentesi angolari (≪`nome`≫) viene chiamato *stereotipo* e serve ad estendere il vocabolario del UML, aggiungendo ulteriori significati (es. ≪`constructor`≫ per indicare che è il metodo dedicato alla costruzione dell'oggetto).

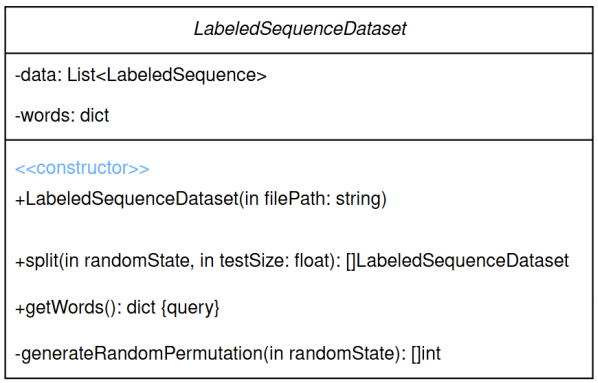

Figura 6.1: UML - Classe

6.2.2 Definizione di Attributi e Metodi

La notazione UML stabilisce formati rigorosi per la dichiarazione di attributi e metodi, specificando dettagli come la molteplicità e le proprietà comportamentali.

La definizione di un attributo segue il formato:

$$[\langle visibilità \rangle] \ \langle nome \rangle [\langle molteplicità \rangle] : \langle tipo \rangle \ [= \langle valore \rangle] [proprietà]$$

Nota: gli elementi tra parentesi quadre [ ] sono opzionali. La visibilità, se non specificata, va intesa come Protetta.

La dichiarazione di un metodo o di un attributo è preceduta da un simbolo che ne specifica la **visibilità**, ovvero il livello di accesso consentito ad altri elementi del sistema. In Tabella 6.1 sono riportati i simboli e il loro significato.

Tabella 6.1: Simboli di visibilità in UML

Simbolo	Tipo di visibilità	Descrizione
+	Pubblica	Qualsiasi elemento può accedere
-	Privata	Solo la classe stessa ne ha accesso
#	Protetto	Solo la classe e le sue sottoclassi ne hanno accesso
~	Package	Solo gli elementi dello stesso package ne hanno accesso

La **molteplicità** indica quante istanze[1] sono presenti di quell'attributo; se non esplicitata si intende 1. Può essere un range nel formato *min..max*, dove *max* può assumere il valore * per indicare l'assenza di un limite.

I valori assumibili dal campo *proprietà* per gli attributi includono:

- **changeable**: non vi sono restrizione sulla modificabilità dell'attributo.
- **addOnly**: valido per gli attributi con molteplicità maggiore di 1, valori possono essere aggiunti ma non rimossi.
- **frozen**: una volta assegnato un valore alla inizializzazione dell'oggetto, non è possibile modificarlo.

La definizione di un metodo segue invece questo formato:

$$[\langle visibilità \rangle] \ \langle nome \rangle (\langle lista\ parametri \rangle) : \langle valore\ di\ ritorno \rangle [\langle proprietà \rangle]$$

I valori possibili per le proprietà di un metodo includono:

- **isQuery**: il metodo non modifica lo stato del sistema.
- **leaf**: il metodo non può essere ulteriormente specializzato.
- **sequential**: i chiamati del metodo devono organizzarsi affinche le chiamate vengono esaudite in modo sequenziale.
- **guarded**: come **sequential** ma l'oggetto chiamato si occupa della logica per essere eseguito in modo sequenziale.
- **concurrent**: anche a chiamate multiple, il sistema risulta integro.

6.2.3 Relazioni Tra Classi

Le classi nel diagramma UML sono collegate da **associazioni** che definiscono le loro interazioni. Tre tipi di relazioni sono fondamentali in questo progetto:

[1] Un **istanza** è una instanzazione di una classe, ovvero un oggetto in esecuzione.

Composizione (Aggregazione Forte) La Composizione (Figura 6.2) è una forma forte di associazione che modella la relazione "parte-intero" ed è rappresentata da un **rombo pieno** attaccato alla classe "intero" (o contenitore). Questa relazione implica che l'esistenza delle "parti" dipende strettamente dall'esistenza dell'oggetto "intero" e che le parti non possono essere condivise da altre istanze del oggetto contenitore. L'esempio mostra come la classe *Sequence* sia composta dalle *ApiCall*, indicando che una sequenza non esiste senza le sue chiamate.

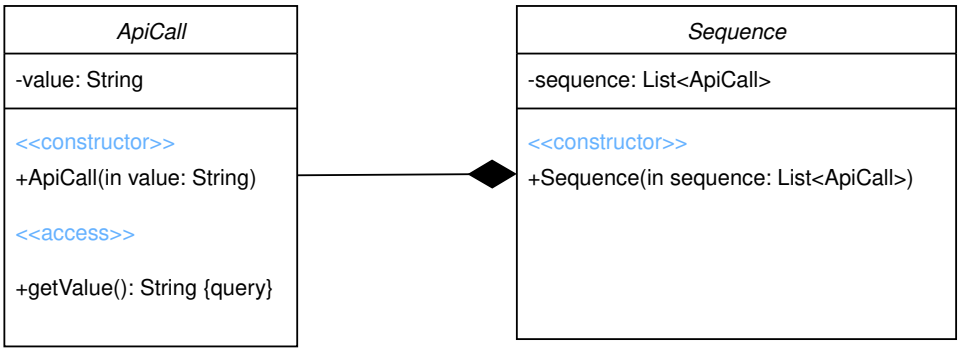

Figura 6.2: UML - Composizione

Use La relazione *use* è formalmente una **dipendenza** ed è rappresentata da una linea tratteggiata con freccia. Essa indica una **dipendenza implementativa** a breve termine: la classe Cliente ha bisogno della classe Fornitore per la sua realizzazione funzionale (e.g., come parametro di un metodo) senza conservarne un riferimento strutturale permanente.

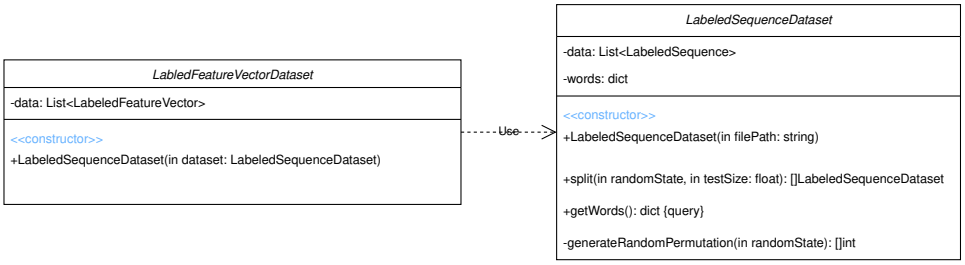

Figura 6.3: UML - Use

Extends La relazione *extends* (Figura 6.4) è fondamentale per l'OOP. Permette alla sottoclasse (quella che genera la freccia) di ereditare tutti gli attributi e i metodi della classe padre (quella puntata). Questo facilita il riuso del codice e la specializzazione tramite *variazione funzionale* dei comportamenti ereditati.

Implements La relazione *Implements* (Figura 6.5), rappresentata da una linea tratteggiata con freccia triangolare, descrive la **realizzazione di un'interfaccia**. La classe che genera la freccia si impegna a implementare tutti i metodi astratti definiti nell'interfaccia puntata.

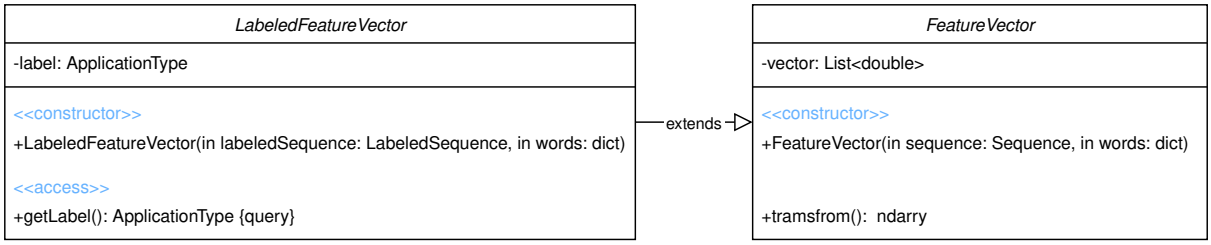

Figura 6.4: UML - Extends (Ereditarietà)

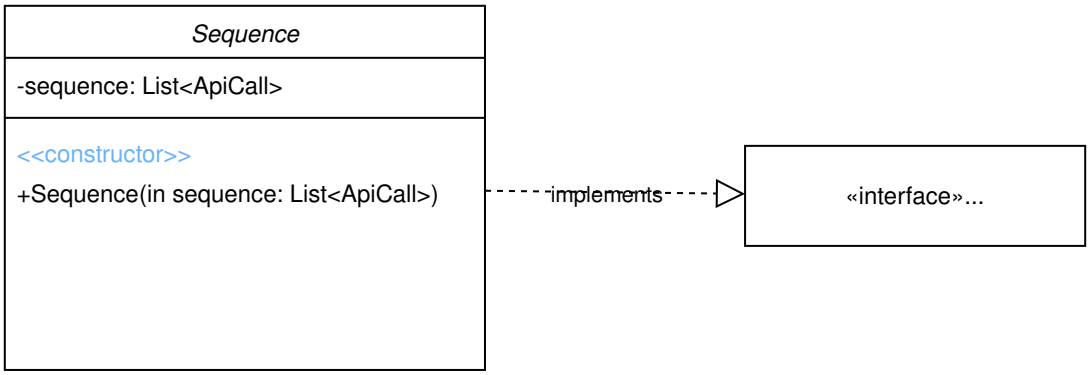

Figura 6.5: UML - Implements (Realizzazione)

Package I **package** in UML (Figura 6.6) sono utilizzati per raggruppare logicamente le classi correlate, migliorando l'organizzazione visiva e concettuale del diagramma delle classi.

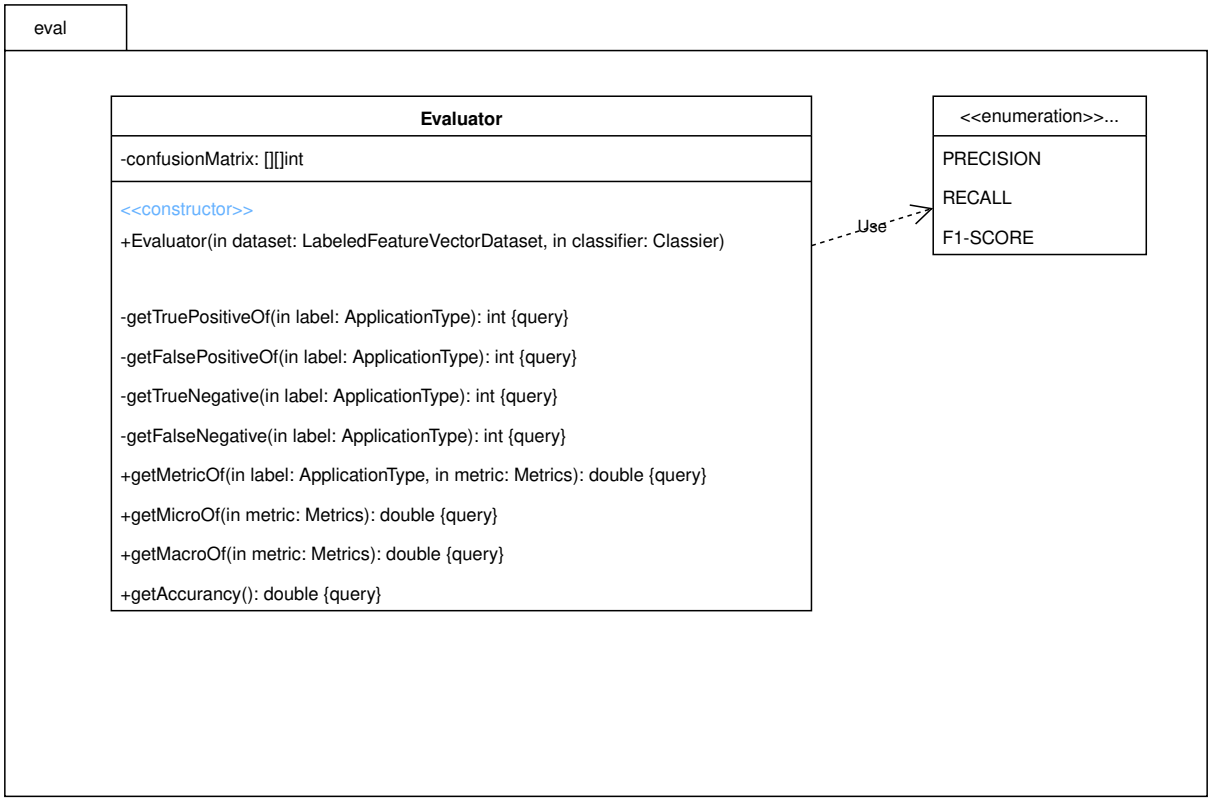

Figura 6.6: UML - Package

Bibliografia

[1] L. Zannella, M. Zanella, and M. Rizzo, "Report Cittadini E ICT," Istat, Report/Study, 2025. [Online]. Available: https://www.istat.it/wp-content/uploads/2025/04/REPORT_CITTADINI-E-ICT_2024.pdf

[2] C. , "© Clusit - Rapporto 2025 sulla Cybersecurity in Italia e nel mondo," Clusit, Tech. Rep., Mar. 2025. [Online]. Available: https://clusit.it/wp-content/uploads/download/Rapporto_Clusit_03-2025_web.pdf

[3] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, no. 2, pp. 32–46, 2007. [Online]. Available: https://profsandhu.com/cs5323_s17/im_2007.pdf

[4] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on api call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 659101, 2015. [Online]. Available: https://doi.org/10.1155/2015/659101

[5] Microsoft, "Introduction to APIs." [Online]. Available: https://learn.microsoft.com/en-us/xandr/industry-reference/intro-to-apis

[6] Y. Qiao, Y. Yang, L. Ji, and J. He, "Analyzing malware by abstracting the frequent itemsets in api call sequences," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, July 2013, pp. 265–270. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6680850

[7] Microsoft, "Formato pe - win32 apps." [Online]. Available: https://learn.microsoft.com/it-it/windows/win32/debug/pe-format

[8] ——, "Libreria di collegamento dinamico." [Online]. Available: https://learn.microsoft.com/it-it/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library

[9] Kaspersky, "Trojan droppers." [Online]. Available: https://encyclopedia.kaspersky.com/glossary/backdoor/

[10] Cisco, "What is a malware." [Online]. Available: https://www.cisco.com/site/us/en/learn/topics/security/what-is-malware.html

[11] Kaspersky, "Trojan droppers." [Online]. Available: https://encyclopedia.kaspersky.com/glossary/trojan-droppers/

[12] ——, "Multipacked." [Online]. Available: https://encyclopedia.kaspersky.com/knowledge/multipacked/

[13] Microsoft, "Indice di api windows - win32 apps." [Online]. Available: https://learn.microsoft.com/it-it/windows/win32/apiindex/windows-api-list

[14] ——, "Getkeyboardstate - win32 apps." [Online]. Available: https://learn.microsoft.com/it-it/windows/win32/api/winuser/nf-winuser-getkeyboardstate

[15] D. Heath, S. Kasif, and S. Salzberg, "k-dt: A multi-tree learning method," in *Proceedings of the Second International Workshop on Multistrategy Learning*, 1993, pp. 138–149.

[16] T. Chen. (2016) Xgboost: A scalable tree boosting system. Archived on 2016-08-07. [Online]. Available: https://web.archive.org/web/20160817055008/http://arxiv.org/abs/1603.02754

[17] M. Christodorescu and S. Jha, *Static Analysis of Executables to Detect Malicious Patterns **. USENIX Association, 2003. [Online]. Available: https://pages.cs.wisc.edu/~jha/jha-papers/security/usenix_2003.pdf

[18] S. Cesare and Y. Xiang, *Software similarity and classification*, 2012th ed. Guildford, England: Springer, 2012.

[19] "Detours: Detours is a software package for monitoring and instrumenting API calls on windows. it is distributed in source code form." [Online]. Available: https://github.com/microsoft/Detours

[20] DynamoRIO Project, "Dynamorio: Dynamic binary instrumentation tool," https://dynamorio.org/, 2025, accessed: 2025-09-20.

[21] wtrace.net, "Using withdll and detours to trace win api calls," https://wtrace.net/guides/using-withdll-and-detours-to-trace-winapi/, Nov. 2023, accessed: 2025-09-20.

[22] IBM, "Che cos'è il machine learning (ml)?" Mar. 2025, accessed: 2025-09-21. [Online]. Available: https://www.ibm.com/it-it/think/topics/machine-learning

[23] Google, "Che cos'è il machine learning?" accessed: 2025-09-21. [Online]. Available: https://developers.google.com/machine-learning/intro-to-ml/what-is-ml?hl=it

[24] ——, "Glossario del machine learning," accessed: 2025-09-21. [Online]. Available: https://developers.google.com/machine-learning/glossary?hl=it#model

[25] ——, "Glossario del machine learning," accessed: 2025-09-21. [Online]. Available: https://developers.google.com/machine-learning/glossary?hl=it#training

[26] ——, "Glossario del machine learning," accessed: 2025-09-21. [Online]. Available: https://developers.google.com/machine-learning/glossary?hl=it#example

[27] ——, "Glossario del machine learning," accessed: 2025-09-21. [Online]. Available: https://developers.google.com/machine-learning/glossary?hl=it#feature

[28] ——, "Glossario del machine learning," accessed: 2025-09-21. [Online]. Available: https://developers.google.com/machine-learning/glossary?hl=it#class

[29] I. Belcic, "What is classification in machine learning?" Sep. 2025. [Online]. Available: https://www.ibm.com/think/topics/classification-machine-learning

[30] T. Mucci, "Cosa sono l'overfitting e l'underfitting?" Mar. 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/overfitting-vs-underfitting

[31] "Cos'è il bagging?" Jun. 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/bagging

[32] "Alberi decisionali (decision trees)." [Online]. Available: https://developers.google.com/machine-learning/decision-forests/decision-trees?hl=it

[33] "Che cos'è la foresta casuale," Feb. 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/random-forest

[34] "Che cos'è il boosting," Nov. 2024. [Online]. Available: https://www.ibm.com/it-it/topics/boosting

[35] "Che cos'è la discesa del gradiente?" Jan. 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/gradient-descent

[36] E. Kavlakoglu and E. Russi, "Che cos'è xgboost?" Jan. 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/xgboost

[37] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[38] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on api call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 659101, 2015. [Online]. Available: https://doi.org/10.1155/2015/659101

[39] C. Stryker and J. Holdsworth. (2025) Cos'è l'nlp (elaborazione del linguaggio naturale)? Data di consultazione: 25 settembre 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/natural-language-processing

[40] W. contributors, "Word2vec," data di consultazione: 25 settembre 2025. [Online]. Available: https://it.wikipedia.org/w/index.php?title=Word2vec&oldid=145522864

[41] E. Kavlakoglu and V. Winland. (2025, feb) Cos'è il k-means clustering? Data di consultazione: 25 settembre 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/k-means-clustering

[42] Wikipedia contributors, "N-gramma," https://it.wikipedia.org/w/index.php?title=N-gramma&oldid=140364382, 2024, wikipedia, The Free Encyclopedia.

[43] IBM, "What are convolutional neural networks?" https://www.ibm.com/think/topics/convolutional-neural-networks, Sep. 2025, iBM.com.

[44] B. P. Gond and D. Mohapatra, "Deep learning-driven malware classification with api call sequence analysis and concept drift handling," *arXiv*, vol. abs/2502.08679, 2025. [Online]. Available: https://arxiv.org/abs/2502.08679

[45] V. Garg and R. K. Yadav, "Malware detection based on api calls frequency," in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, 2019, pp. 400–404. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9036219/authors#authors

[46] IBM, "What is support vector machine?" June 2025. [Online]. Available: https://www.ibm.com/think/topics/support-vector-machine

[47] C. Ma, Z. Li, H. Long, A. Bilal, and X. Liu, "A malware classification method based on directed api call relationships," *PLOS ONE*, vol. 20, no. 3, pp. 1–26, 03 2025. [Online]. Available: https://doi.org/10.1371/journal.pone.0299706

[48] G. Gaudio, "Acpamc: Api call pattern analysis for malware classification," https://github.com/ThePino/ACPAMC, 2025, repository GitHub.

[49] Y. Shafranovich, "Common format and mime type for comma-separated values (csv) files," Internet Engineering Task Force (IETF), RFC 4180, 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4180

[50] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, T. Bray, Ed. RFC Editor, 2017.

[51] Wikipedia contributors, "Bag-of-words model," https://en.wikipedia.org/w/index.php?title=Bag-of-words_model&oldid=1289978794, May 2025, accessed: 2025-09-26.

[52] ——, "Campionamento stratificato," 2025, ultimo accesso: 26 settembre 2025. [Online]. Available: https://it.wikipedia.org/w/index.php?title=Campionamento_stratificato&oldid=141667417

[53] Scikit-learn developers, "Scikit-learn: Machine learning in python," https://scikit-learn.org/stable/index.html, 2025, accessed: 2025-09-26.

[54] B. Wagner, "Programmazione object-oriented in c#," 2023, accessed: 2025-09-26. [Online]. Available: https://learn.microsoft.com/it-it/dotnet/csharp/fundamentals/tutorials/oop

[55] IBM Developer, "The class diagram," 2023, accessed: 2025-09-27. [Online]. Available: https://developer.ibm.com/articles/the-class-diagram/

[56] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on api call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 2015, pp. Article ID 659 101, 9 pages, 2015. [Online]. Available: https://ocslab.hksecurity.net/apimds-dataset

[57] F. O. Catak, J. Ahmed, K. Sahinbas, and Z. H. Khand, "Data augmentation based malware detection using convolutional neural networks," *PeerJ Computer Science*, vol. 7, p. e346, Jan. 2021. [Online]. Available: https://doi.org/10.7717/peerj-cs.346

[58] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury, "Api-maldetect: Automated malware detection framework for windows based on api calls and deep learning

techniques," *Journal of Network and Computer Applications*, p. 103704, 2023.

[59] D. Trizna, "Quo vadis: Hybrid machine learning meta-model based on contextual and behavioral malware representations," in *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*, ser. AISec'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 127–136. [Online]. Available: https://doi.org/10.1145/3560830.3563726

[60] F. O. Catak, "malware_api_class: Malware dataset for security researchers and data scientists," 2020, public malware dataset generated by Cuckoo Sandbox based on Windows OS API call analysis for cybersecurity research. [Online]. Available: https://github.com/ocatak/malware_api_class/tree/master

[61] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury, "Malbehavd-v1: Public dataset of malware and benign executable files (windows exe files)," 2023, dataset for cybersecurity research, suitable for training and testing machine learning and deep learning algorithms. [Online]. Available: https://github.com/mpasco/MalbehavD-V1/tree/main

[62] D. Trizna, "quo.vadis: Hybrid machine learning model for malware detection based on windows kernel emulation," 2022, hybrid machine learning model for malware detection based on Windows kernel emulation. [Online]. Available: https://github.com/dtrizna/quo.vadis/tree/main

[63] J. Murel and E. Kavlakoglu, "Che cos'è una matrice di confusione?" Feb. 2025. [Online]. Available: https://www.ibm.com/it-it/think/topics/confusion-matrix

[64] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence," *Computers & Security*, vol. 92, no. 101760, p. 101760, 2020. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2020.101760

[65] "The open source AI code editor," https://code.visualstudio.com/, accessed: 2025-11-24.

[66] "Dev containers - visual studio marketplace," https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-containers, accessed: 2025-11-24.

[67] S. Ratliff, "Docker: Accelerated container application development," https://www.docker.com/, Apr. 2025, accessed: 2025-11-24.

[68] "Development Container specification." [Online]. Available: https://containers.dev/implementors/spec/#:~:text=A%20development%20container%20is%20a,scale%20to%20large%20development%20groups.

[69] "GitHub Codespaces," 2025. [Online]. Available: https://github.com/features/codespaces