UNIVERSITÀ
DEGLI STUDI
DI PALERMO

NDS
NETWORKING AND DISTRIBUTED SYSTEMS
RESEARCH GROUP

Progetto e sviluppo di un sistema di analisi dinamica di malware per dispositivi mobili

Tesi di Laurea Magistrale in Ingegneria Informatica

Emiliano Oddo

Relatore: Prof. Giuseppe Lo Re
Correlatore: Ing. Marco Morana

Progetto e sviluppo di un sistema di analisi dinamica di malware per dispositivi mobili

Tesi di laurea di:
Emiliano Oddo

Relatore:
Ch.mo Prof. Giuseppe Lo Re

Controrelatore:
Ing. Daniele Peri

Correlatore:
Ing. Marco Morana

Sommario

Gli *smartphones* e i dispositivi mobili in genere stanno rapidamente diventando indispensabili per molti utenti. Sfortunatamente, a causa della crescente popolarità, diventano anche campo fertile per gli *hacker*, permettendo loro di sviluppare e immettere *malware* nei *marketplace* di applicazioni per dispositivi mobili, soprattutto quelli per sistemi operativi Android. Grazie alla sua architettura *open* e al notevole numero di utenti, il sistema Android fornisce agli sviluppatori un accesso semplice al codice ed una grande facilità di distribuzione di *software* malevolo. Pertanto, vi è la necessità di riconoscere i *malwares*, classificandoli come tali. La presente tesi descrive la progettazione e realizzazione di un sistema che sfrutta l'analisi dinamica dei *malware* su dispositivi mobili con sistema operativo Android, effettuando una classificazione binaria e distinguendo il software malevolo da quello benigno. Per l'attività di analisi, è stato utilizzato un software di tipo *sandbox* in cui eseguire le applicazioni senza rischiare di infettare il dispositivo in cui viene eseguita l'applicazione stessa. Sono state estratte diverse caratteristiche per descrivere il comportamento del software analizzato e classificarlo, utilizzando vari algoritmi di apprendimento. Il sistema utilizza un'architettura *client-server* per potervi interagire da remoto, permettendo ad un possibile *client* di usufruire del servizio di analisi fornito dal *server*. Per verificare l'efficienza del sistema proposto, sono stati condotti degli esperimenti per confrontare quanto si è realizzato con altre tecniche allo stato dell'arte.

INDICE

INTRODUZIONE

Lo sviluppo di applicazioni *Android* ed i *marketplace* di applicazioni Android si sono espansi drasticamente negli ultimi anni. Uno dei principali motivi di crescita è la sua facilità di sviluppo e distribuzione delle applicazioni.
Google Play Store e molti altri *market* di applicazioni *Android* in tutto il mondo consentono agli sviluppatori di caricare e distribuire le loro applicazioni in modo facile e veloce. La pubblicazione e gli aggiornamenti delle applicazioni sono molto semplici e richiedono una revisione manuale minima. Mentre, da un lato, la facilità di distribuzione ha aiutato gli sviluppatori, dall'altro, ha reso la distribuzione di applicazioni dannose considerevolmente semplice. Questo *software* dannoso viene più comunemente indicato con il nome di *Malware*, che indica un qualsiasi programma informatico usato per disturbare le operazioni svolte da un computer, rubare informazioni sensibili, accedere a sistemi informatici privati, ecc. A causa dell'enorme volume e dell'aumento dei caricamenti delle applicazioni *Android*, è molto difficile e costoso effettuare una revisione manuale del *software* che di volta in volta viene caricato sui *marketplace*. Queste condizioni hanno portato alla creazione e distribuzione di *malware Android* in modo molto semplice.
A causa delle limitazioni sulla revisione manuale delle applicazioni, molti *market Android* hanno necessità di un sistema di analisi dei *malware* automatizzato rapido ed efficiente. I principali *market* di applicazioni, come *Google Play Store*, hanno sviluppato le proprie soluzioni automatizzate di analisi del *malware* e hanno eliminato il *malware* esistente, ma gli archivi applicativi di piccole dimensioni dipendono ancora in gran parte dalle tradizionali tecniche di scansione basate sull'*hash*.

L'obiettivo di questo lavoro di tesi è sviluppare un sistema di classificazione di *malware* su sistema *Android*, basato sull'apprendimento automatico, a partire dall'analisi dinamica degli applicativi effettuata con *CuckooDroid Sandbox*. L'analisi dinamica fornirà le informazioni sul comportamento dei *malware*, che verranno utilizzate successivamente come caratteristiche per gli algoritmi di

Machine Learning. Per la classificazione dei *malware* e dei file benigni, si fa uso di diversi algoritmi, come Random Forest, Decision Tree, le reti Bayesiane e le reti neurali. L'obiettivo è quello di determinare il metodo di classificazione migliore, che sia in grado di distinguere i file malevoli da quelli benigni con la maggiore precisione.

L'elaborato è organizzato come segue: nel capitolo 2 vi è una breve descrizione delle tipologie di analisi dei *malware*, mostrando le varie differenze che le caratterizzano; nel capitolo 3 viene mostrato l'ambiente di sviluppo, illustrando le varie componenti utilizzate nel sistema, in particolare viene descritto il software CuckooDroid, unità portante di questo lavoro; nel capitolo 4 vengono illustrate le scelte progettuali a livello di architettura e configurazione; nel capitolo 5 ci si concentra sull'analisi dei dati ottenuti e le estrazioni dell'informazioni utili ai fini del progetto; nel capitolo 6 vengono mostrate e descritte le tecniche di *Machine Learning* utilizzate per la fase di classificazione; nel capitolo 7 vengono mostrati i risultati sperimentali ottenuti e, infine, nel capitolo 8 sono presenti alcune conclusioni sul lavoro effettuato.

CAPITOLO 1: Lo Stato dell'Arte

I *malwares* per il sistema operativo Android sono in forte aumento negli ultimi anni a causa della crescente popolarità di Android e della proliferazione di *marketplace* di applicazioni di terze parti. Nel 2018, sono quasi 4,57 miliardi i dispositivi mobili posseduti dagli utenti, di cui l'88% dei dispositivi esegue il sistema operativo Android. Le famiglie di malware Android emergenti stanno adottando tecniche sempre più sofisticate per evitare il rilevamento e ciò richiede approcci più efficaci per riuscire a riconoscere e distinguere i malware Android dalle applicazioni benigne, o *goodware*. I *malware* in Android non sono più soltanto *Trojan* SMS o *Spyware*, ma sono in aumento anche diverse categorie di *malware*, tra cui *Ransomware*, *Hostile Downloaders*, *Botnet Trojan*.
Google classifica i malware Android in 17 diverse categorie, come mostrato nella **Tabella 1**:

Backdoors	Commercial Spyware	Phishing Applications
Billing Fraud	Data Collection	Trojan
SMS Fraud	Denial of Service	Ransomware
Call Fraud	Hostile Downloaders	Rooting Apps
Spam	Spyware	Non-Android Threat
Wireless Access Protocol (WAP) Fraud	Privilege Escalation Apps	

Tabella 1: *Categorie di malware Android per Google*

I due principali approcci comunemente utilizzati per il rilevamento dei malware sono l'analisi statica e l'analisi dinamica, che si basano sulla raccolta dei dati effettuando l'esecuzione o meno dell'applicazione.

1.1 - Analisi statica

L'analisi statica si concentra principalmente sui dati che possono essere raccolti da un file .apk, eseguibile dell'applicazione *Android*. Ciò su cui si focalizza l'analisi statica è il codice e le possibili stringhe e *sub-routine* nel codice dell'applicazione.

I sistemi di rilevamento delle intrusioni di solito utilizzano questo tipo approccio, come, ad esempio, nei sistemi basati su *host* che fanno uso della tecnica *Misuse-Based*, in cui vengono utilizzate le caratteristiche strutturali e di sintassi delle applicazioni e le loro informazioni statiche per rilevare le attività di intrusione e il loro codice. In questo modo, si cerca di determinare le attività di intrusione del programma e il suo codice prima che l'applicazione sia eseguita [34].

L'analisi statica si concentra sulla creazione di firme *hash* e sui *pattern* di *byte-code* per il rilevamento di applicazioni malevoli. Il disassemblaggio e le revisioni manuali del codice sono compiti fondamentali nell'analisi statica [35].

Uno degli approcci più comuni, tipico dei sistemi di intrusione *Knowledge-Based*, consiste nell'utilizzare dei software che eseguono la scansione del *malware* confrontando i risultati con le informazioni presenti in un database di riferimento, che vengono chiamate *firme* [34]. Anche se con l'analisi statica vengono rivelate molte informazioni, non sono sufficienti soltanto questi dati per rivelare le funzionalità e le caratteristiche di qualsiasi *malware*. L'analisi dinamica d'altra parte valuta il comportamento dell'applicazione durante la sua esecuzione.

Diversi sono stati gli approcci di tipo statico, come quelli, ad esempio, dei lavori descritti in [21] e in [28], come il sistema descritto in [29] e quello degli autori di [9] e di [7].

Ad esempio, gli autori di [28] hanno studiato un approccio di analisi statica per il rilevamento di malware Android basato su 179 caratteristiche derivate da chiamate API, intenti, permessi e comandi che sono stati combinati con l'apprendimento dell'insieme. Il loro approccio è stato valutato su un *dataset* di 2925 campioni di *malware* e 3938 campioni benigni. Vari approcci analoghi al rilevamento di malware statico hanno utilizzato caratteristiche derivate similmente,

ma con diversi classificatori come Support Vector Machine (SVM) nel lavoro [2], Naïve Bayes in [27] e k-Nearest Neighbor (KNN) in [21].

Sono stati anche proposti approcci di rilevamento di malware Android basati sull'apprendimento automatico che utilizzano funzionalità statiche derivate esclusivamente dai permessi.

Gli autori di [14] hanno proposto uno schema di rilevamento di *malware* Android basato su permessi a 2 livelli. Nel loro sistema, ogni fase utilizza l'algoritmo J48 per identificare applicazioni malevoli. Il primo livello utilizza i permessi richiesti e le coppie di permessi richieste, mentre il secondo livello utilizza le "coppie di permessi usate" estratte dai file eseguibili Dalvik. Un semplice algoritmo a tre fasi viene utilizzato per ottenere la decisione di classificazione generale.

Nel sistema descritto in [19] sono state valutate semplici logistiche, usando classificatori Naïve Bayes, reti Bayesiane, SVM, IB*k* (KNN), J48, Random Tree e Random Forest usando i permessi come caratteristiche. È stato usato un *dataset* di 249 campioni di malware e 357 campioni benigni e ottenuto i migliori risultati con il classificatore Random Forest (AUC = 0.92).

In [16] , gli autori hanno presentato risultati sperimentali su 2338 campioni benigni e 1446 campioni di *malware* utilizzando i permessi. Hanno studiato 4 diversi metodi di selezione delle caratteristiche con classificatori Naïve Bayes, Classification and Regression Tree (CART), J48, SVM e Random Forest. I loro risultati hanno anche dimostrato che il classificatore Random Forest mostra la migliore precisione di classificazione (94,9%).

Il sistema APK Auditor descritto in [23] utilizza i permessi con un approccio di calcolo statistico per rilevare applicazioni Android malevoli. In contrasto con gli approcci menzionati, che si basano su funzioni predefinite di alto livello, come

autorizzazioni o chiamate API, il rilevamento del malware basato su *n-gram* utilizza come caratteristiche le sequenze di *opcode* di basso livello.

Le caratteristiche *n-gram* possono essere utilizzate per addestrare un classificatore per distinguere i *malware* dalle applicazioni benigne, come nel lavoro [10], o per classificare un malware in famiglie diverse, come mostrato in [11]. Forse, sorprendentemente, anche le funzionalità basate su *1-gram*, che sono semplicemente un istogramma del numero di volte in cui viene utilizzato ogni *opcode*, possono essere utili per distinguere il *malware* da applicazioni benigne [5]. La lunghezza del *n-gram* usato [10] e il numero di *n-gram* usati nella classificazione [5] possono entrambi avere un effetto sulla precisione del classificatore.

Gli autori di [10] hanno utilizzato sequenze di *opcode* con l'apprendimento automatico e hanno sperimentato con funzionalità binarie a 2, 3, 4 e 5 *grams*. Il posizionamento delle caratteristiche e la selezione sono state fatte basandosi sul calcolo del *information gain* di ciascuna sequenza di *opcode* delle sequenze trovate nei *training set*. Gli autori hanno valutato il loro approccio sui campioni malevoli dell'Android Malware Genome Project (AMGP) e campioni benigni ottenuti dal Google Play Store. Sono stati in grado di ottenere le migliori prestazioni di classificazione con le caratteristiche da 5 *grams* (F-measure medie globali vicine a 0.9771) utilizzando un classificatore SVM lineare. Il documento ha esaminato solo la classificazione del *malware* in benigni o sospetti e non ha considerato la classificazione della famiglia di *malware*. Inoltre, l'approccio si basa su occorrenze binarie di *opcode* degli *n-gram* in ogni applicazione e non considera alcuna informazione di frequenza associata agli *opcode* degli *n gram* presenti in ciascuna applicazione.

A differenza del lavoro presentato in [10], gli autori di [4] hanno presentato risultati basati sulla classificazione della famiglia di *malware* per testare il loro approccio utilizzando classificatori SVM (Gaussiano) e Random Forest. Hanno presentato risultati utilizzando un set di applicazioni attendibili e anche 10 famiglie di *malware* Android: FakeInstaller, Plankton, DroidKungFu, GinMaster,

BaseBridge, ADRD, Kmin, Gemini, DroidDream e Opfake. Inoltre, i loro esperimenti coprivano intervalli di *n* da 1 a 5, cioè fino a un massimo di *opcode* di 5 *grams* e fino a 2000 caratteristiche di *n-gram* per schema dopo aver applicato un algoritmo di selezione delle caratteristiche. Il miglior risultato riportato nel documento è il 96,88% di precisione usando il classificatore SVM con 1000 caratteristiche da *2 grams*.

Si noti che, sebbene i lavori [11], [4], [17], [25] e [6] utilizzavano funzioni *opcode* statiche, non erano basate su *n-gram*. Gli *n-gram* sono stati applicati alle sequenze di chiamate di sistema ottenute da applicazioni Android come descritto nel lavoro [15]. Tuttavia, le sequenze statiche di *opcode* hanno un carico computazionale molto inferiore rispetto alle sequenze delle chiamate di sistema, poiché queste ultime sono basate sull'analisi dinamica.

A differenza dei precedenti lavori citati, gli autori di [12] esaminano un approccio basato su caratteristiche di opcode statico ad *n-gram*, che consente l'uso di *n-grams* più lunghi analizzando fino a 10 grammi, mentre i lavori precedentemente riportati utilizzano fino a *5 grams* [10] [4] . Il sistema utilizza l'apprendimento automatico per indentificare e classificare i *malware* Android. Inoltre, a differenza del lavoro [10] che si basava solo su informazioni binarie, gli autori di [12] sfruttano sia la frequenza che le informazioni binarie, consentendo una maggiore copertura delle informazioni. Gli esperimenti sono stati effettuati su un *dataset* di 2520 campioni ed hanno mostrato una precisione del 98% utilizzando l'approccio basato su opcode ad *n-gram*.

Infine, il lavoro di tesi del collega Castronovo, che ha progettato e sviluppato un sistema di analisi statica di malware su dispositivi Android in grado di rilevare e classificare, con l'ausilio di algoritmi di machine learning come le reti neurali, le reti bayesiane e l'algoritmo Random Forest, il software malevolo da quello benigno.

1.2 - Analisi dinamica

L'analisi dinamica si concentra sul comportamento a *run-time* dell'applicazione. L'analisi implica l'esecuzione e il monitoraggio dell'attività svolta dall'applicazione, la quale viene solitamente eseguita in un ambiente sicuro e protetto, definito "*sandbox*".

Una *sandbox* identifica normalmente un ambiente estraneo, in cui possono essere fatte sperimentazioni per testare dei *software* in fase di sviluppo, senza comportare problemi al sistema ospitante. Il monitoraggio delle attività e la raccolta dei dati vengono effettuati a livello di sistema, a livello di rete ed a livello di memoria. I dati a livello di sistema includono le chiamate di sistema effettuate dalle applicazioni. I dati a livello di rete includono le richieste DNS, *payloads* scaricati, domini e indirizzi IP contattati dall'applicazione. I dati di rete rivelano, inoltre, dettagli sul protocollo e le richieste HTTP effettuate dall'applicazione. I dati a livello di memoria si ottengono di solito durante il debug dell'applicazione, che coinvolge i *breakpoint* a livello di codice e il monitoraggio o i registri, i dati di *stack* e *heap*.

L'analisi dinamica a volte è molto più potente dell'analisi statica in quanto rivela il comportamento del *malware* e funzionalità intenzionalmente nascoste, anche se alcuni *malware* possono accorgersi di essere eseguiti in una sandbox e possono nascondere parte del proprio comportamento malevolo.

Approcci basati sull'apprendimento che utilizzano features prestabilite sono stati applicati ampiamente anche al rilevamento di malware dinamico, come i lavori [30] e [20], oppure come il lavoro [3] e quello degli autori di [22] e di [8] ed il sistema descritto in [1].

Gli autori di [1] presentano un sistema per identificare dinamicamente se un'applicazione Android è malevola o meno, in base all'apprendimento automatico e alle funzionalità estratte dalle *API call* Android e dalle chiamate di sistema. Il sistema è stato valutato con 7.520 app, di cui 3.780 come *training set* e 3.740 come *test set*, ottenendo una precisione del 96.66%.

Gli autori di [7] hanno estratto alcune *API call* predefinite categorizzate in *API call* relative alla *privacy*, relative alla rete, legate agli SMS, relative al *Wi-Fi* e ai componenti. Hanno combinato queste caratteristiche con i permessi Android ed hanno addestrato 7 classificatori: Naïve Bayes, SVM, Radial Basis Function (RBF), Multi-Layer Perceptron (MLP), Liblinear, Decision Tree e Random Forest. Hanno scoperto che la migliore precisione di classificazione è stata ottenuta dal classificatore *Random Forest* quando le *API call* sono state utilizzate con i permessi rispetto all'utilizzo dei soli permessi. I loro esperimenti sono stati eseguiti su un *dataset* di 796 campioni costituiti da 621 campioni benigni e 175 *malware*.

Gli autori di [22] presentano un *framework* di protezione per *smartphone* che consente ai *market-place* Android ufficiali e alternativi di rilevare applicazioni malevoli tra le nuove applicazioni inviate per il rilascio nei vari *store*. Il sistema è costituito da *server* che funzionano su *cloud,* dove gli sviluppatori che desiderano rilasciare le loro nuove applicazioni possono caricare il loro *software* a scopo di verifica. Il *server* di verifica utilizza per prima cosa le statistiche sulle chiamate di sistema per identificare potenziali applicazioni malevoli; gli utenti che eseguono le nuove applicazioni possono richiamare uno strumento di monitoraggio del traffico di rete (NTM), che attiva l'acquisizione del traffico di rete dopo aver rilevato alcuni comportamenti sospetti. I risultati sperimentali utilizzando 120 applicazioni di test (che consistono di 50 malware e 70 applicazioni normali) indicano una precisione del 94,2% e del 99,2% con i classificatori, rispettivamente, J.48 e Random Forest, utilizzando il framework.

Gli autori di [18] hanno presentato PMBS, un sistema di rilevamento di malware basato sui permessi. PMBS estrae i permessi richiesti come indicatori di comportamento e crea dei classificatori di apprendimento automatico per identificare automaticamente comportamenti potenzialmente malevoli. Hanno considerato C4.5, lazy algorithm (instance-based learner), ripetute potature incrementali per produrre la riduzione dell'errore (RIPPER) e classificatori Naïve Bayes, utilizzando un *dataset* con 1500 campioni di malware benigno e 1450.

Hanno anche applicato la tecnica di potenziamento del meta-apprendimento sui classificatori C4.5, RIPPER e Naïve Bayes. I loro risultati hanno mostrato che il C4.5 "potenziato" ha ottenuto risultati migliori con un'accuratezza del 95,22%.

1.3 - Analisi ibrida

Il modo migliore per raccogliere tutti i dati rilevanti dell'analisi è di seguire l'approccio dell'analisi ibrida, cioè utilizzando insieme l'analisi statica e dinamica. Questo approccio potrebbe richiedere molte risorse e molto tempo, ma fornisce risultati più significativi e accurati a causa della completezza dei dati.

L'analisi statica è generalmente efficiente in termini di tempo e l'analisi dinamica è più orientata alla profondità. L'utilizzo dell'analisi ibrida consente la creazione di firme robuste e l'estrazione di funzionalità di rilevamento dinamico. Consente di raccogliere il numero massimo di funzionalità che consente un rilevamento più accurato.

Vi sono diversi approcci basati sul rilevamento di malware dinamico, come [13] e [24], ma anche il sistema mostrato in [26] e quello mostrato in [45].

Per quanto riguarda il lavoro mostrato in [46], realizzato nel laboratorio NdsLab dell'Università degli Studi di Palermo, svolto dai gruppi di ricerca coordinati dai Proff. Gaglio e Lo Re [45], e relativo ad un sistema di rilevato di *malware* su *Big Data*, si basa sull'utilizzo di un approccio di analisi ibrida. In particolare, il sottosistema di analisi statica, che si basa su una rete profonda, elabora una piccola parte del file attraverso un processo leggero e fornisce anche la probabilità che il file analizzato sia un *malware*, fornendo così una misura del suo grado di incertezza. Se l'incertezza supera una determinata soglia, viene utilizzato il sottosistema di analisi dinamica. In questo modo, viene data la priorità al sistema di analisi statica, data la sua maggiore efficienza e poco onere a livello di risorse, e, eventualmente, utilizzare il sistema di analisi dinamica quando viene superato un certo valore di incertezza.

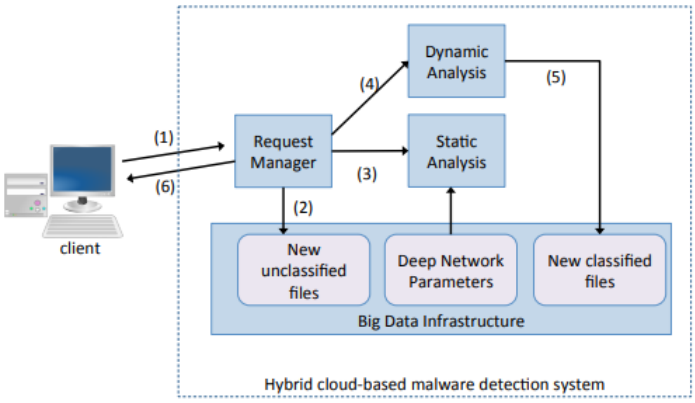

Figura 1: *Esempio di sistema di analisi ibrida*

Per quanto riguarda il lavoro descritto in [13], viene mostrato MARVIN, un sistema che combina l'analisi statica con quella dinamica e che sfrutta le tecniche di apprendimento automatico per valutare il rischio associato ad applicazioni Android sconosciute assegnando a ciascuna di esse un punteggio di malignità. MARVIN esegue analisi statiche e dinamiche per rappresentare le proprietà e gli aspetti comportamentali di un'applicazione attraverso varie di funzionalità.

Per le funzioni di analisi statica è stato analizzato il *file manifest*, che deve essere fornito con ogni applicazione. Contiene informazioni essenziali di cui il sistema operativo Android ha bisogno per installare ed eseguire un'applicazione. Tra i dati contenuti in un *file manifest*, vengono estratte alcune informazioni, come il nome del pacchetto Java che identifica in modo univoco l'applicazione, le autorizzazioni richieste, ecc. Per quanto riguarda l'aspetto dinamico, durante l'analisi vengono monitorati gli eventi relativi alle operazioni sui file, alle operazioni di rete, eventi relativi al comparto telefonico, perdite di dati, ecc.

CAPITOLO 2: Tecniche di classificazione

In questo capitolo, verranno mostrate le tecniche ed algoritmi di Machine Learning utilizzati nel sistema per la classificazione dei *malware*, illustrando brevemente le loro caratteristiche ed il loro funzionamento. Esistono molte tecniche per la classificazione, ma il problema è che i classificatori non sono paragonabili in generale. Dal momento che nessun classificatore è strettamente migliore degli altri, si è pensato di utilizzarne alcuni per effettuare un confronto sui risultati. Per questo progetto, sono stati utilizzati l'algoritmo Random Forest, J48, le Reti Neurali e le Reti Bayesiane, poiché questi sono i classificatori più usati in letteratura per questo scopo.

2.1 - Random Forest

Random Forest è un algoritmo di machine learning semplice e molto utilizzato in letteratura, producendo molto spesso ottimi risultati. È noto sia per la sua semplicità, sia perché può essere usato per effettuare classificazione ma anche per regressione. Si tratta di un algoritmo basato sui *Decision Tree* che, al contrario di un normale algoritmo con alberi di decisione, come J48, utilizza un insieme di alberi per ridurre la varianza nella classificazione e, quindi, migliorare l'accuratezza. L'algoritmo Random Forest è stato implementato da Leo Breiman, un famoso statista statunitense. L'algoritmo combina più alberi decisionali per ogni campione usato, effettuando una classificazione basata sul *majority vote* di tutti gli alberi. Gli alberi sono generati usando ciò che viene chiamato *Bagging* (by Breiman) o *Bootstrapping*. Il *bootstrap* funziona generando un sottoinsieme casuale del set di campioni totale, ovvero il *training data* per ogni albero. Ciò significa che ogni albero, quando esso viene generato, è basato su diversi campioni. Per questo motivo, si presuppone che gli alberi generati siano soggetti a rumore e non distorti l'uno con l'altro, con conseguente elevata varianza e decorrelazione tra loro.
L'algoritmo può essere espresso con le seguenti fasi:

1. Viene creata una quantità T di alberi di decisione e per ogni albero:
    a. Si genera un campione di bootstrap **Z*** di dimensione N dall'intero spazio campione (*training data*);
    b. Si fa crescere l'albero T_i basandosi sulla ripetizione ricorsiva di quanto segue in ciascun nodo:
        i. Vengono selezionate casualmente le caratteristiche da un sottoinsieme di caratteristiche, del campione di bootstrap, come candidati per la divisione dei dati;
        ii. Viene scelta la migliore suddivisione delle caratteristiche tra le m caratteristiche casuali basate sul più alto IG (*Information Gain*);
        iii. Vengono creati due nodi figli.
2. In output si ha l'insieme degli alberi.

In particolare, $m = \mathbf{int} \log_2((M) + 1)$ dove M è il numero massimo di features in input. Dal momento che le caratteristiche vengono scelte casualmente, si creerà un'alta varianza tra gli alberi, che generalmente conduce a risultati migliori. Al termine della fase di *training*, i dati di *testing* sono dati in input ai *decision tree*, ognuno dei quali propone un'etichetta per ogni istanza. Alla fine, per ogni istanza viene effettuato un *majority voting* e viene asgnata un'etichetta di conseguenza.

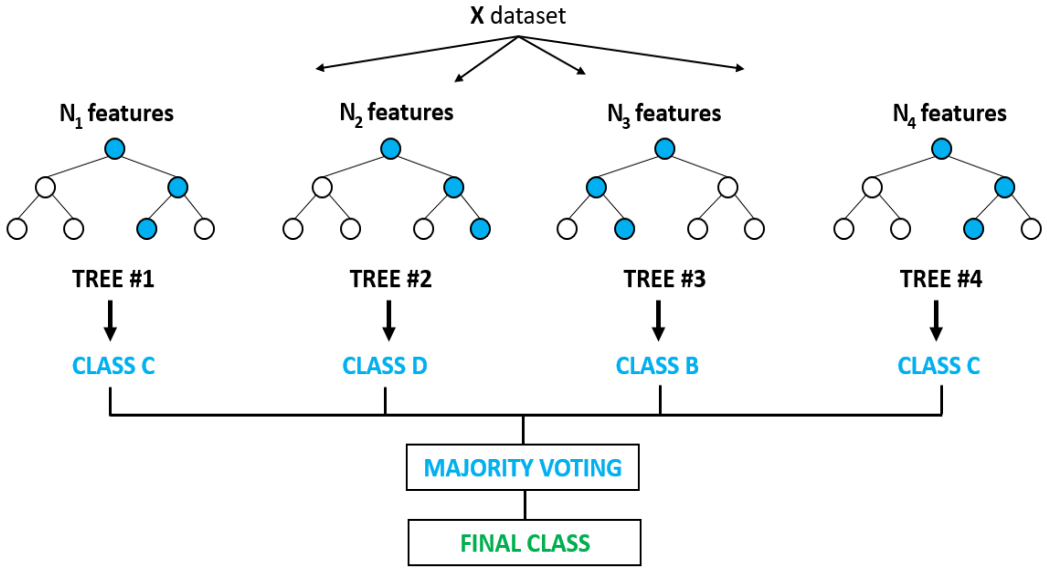

Figura 18: *Esempio dell'algoritmo di Random Forest*

2.2 - J48 Decision Tree

J48 è anch'esso un algoritmo di learning supervisionato basato sull'utilizzo dei Decision Tree. Nello strumento di data mining WEKA, J48 è un'implementazione Java open-source dell'algoritmo C4.5. Considerando la struttura del Decision Tree, l'albero è costituito dai nodi decisionali e dai nodi foglia, ed i primi sono connessi a diversi nodi foglia tramite rami. I nodi foglia rappresentano le decisioni o le classificazioni. Il nodo iniziale più in alto rappresenta il nodo radice.

L'algoritmo comune per i Decision Tree è ID3 (Dichotomiser Iterative 3), il quale si basa sui concetti di entropia e di *information gain*. Le funzionalità aggiuntive di J48 tengono conto di valori mancanti, potatura dei decision tree, intervalli di valori degli attributi continui, derivazione di regole, ecc.
In breve, l'algoritmo ID3 può essere descritto come segue: partendo dal nodo radice, in ogni fase si vuole suddividere i dati in un insieme di dati omogeneo (simile nella loro struttura). Nello specifico, si vuole trovare l'attributo che determinerebbe il più alto *information gain*, ovvero restituire i rami più omogenei:

1. Calcolo dell'entropia del target:

$$E(T,X) = \sum_{c \in X} P(c)\, E(c)$$

$$E(S) = -\sum_{i=1}^{c} p_i \log_2 p_i$$

2. Dividere il set di dati e calcolare l'entropia di ciascun ramo. Quindi, calcolare il guadagno di informazioni, cioè la differenza dell'entropia iniziale e la somma proporzionale delle entropie dei rami.

$$Gain(T,X) = Entropy(T) - Entropy(T,X)$$

3. L'attributo con il valore di guadagno più alto viene selezionato come nodo decisionale.
4. Se uno dei rami del nodo decisionale selezionato ha un'entropia pari a 0, diventa il nodo foglia. Altri rami richiedono ulteriori suddivisioni.
5. L'algoritmo viene eseguito in modo ricorsivo finché non vi è più nulla da suddividere.

Il metodo dei *decision tree* ha raggiunto la sua popolarità grazie alla sua semplicità. Può gestire bene dataset di grandi dimensioni e può gestire molto bene il rumore nei dati. Un altro vantaggio è che, a differenza di altri algoritmi, come SVM o KNN, gli alberi decisionali operano in una "*white box*", il che significa che possiamo vedere chiaramente come si ottiene il risultato e quali decisioni lo hanno portato.

Inoltre, viene utilizzata la tecnica di *Pruning*, ovvero la rimozione di rami non promettenti per il processo di decisione.
La classificazione viene eseguita sulle istanze del *training set* e viene creato l'albero di decisione. Il *pruning* viene eseguito diminuendo gli errori di classificazione che vengono prodotti da specializzazioni nel *training set*, in modo da favorire la generalizzazione dell'albero ed evitare il problema dell'*overfitting*.

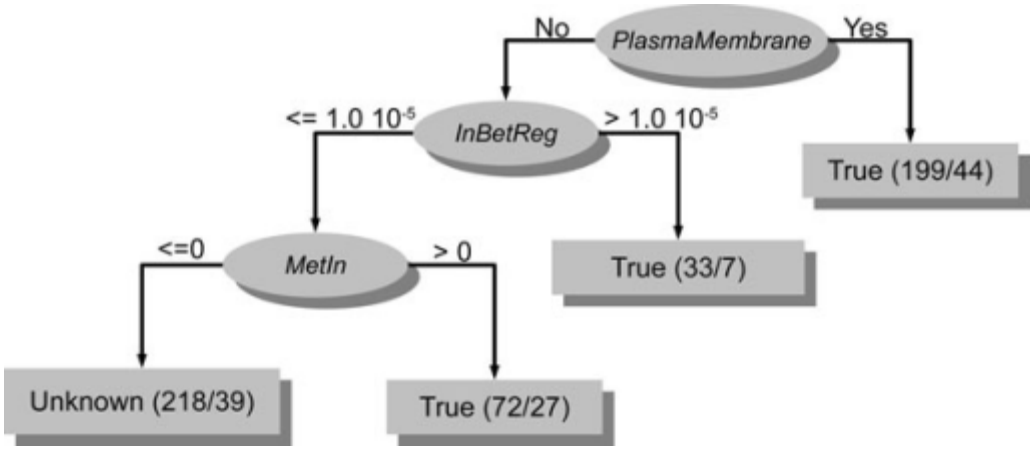

Figura 19: *Esempio di Decision Tree con l'algoritmo J48*

2.3 - Reti Neurali

Le reti neurali sono modelli di apprendimento costruiti tramite interconnessioni tra un insieme di unità logiche secondo una particolare topologia, che ne determina il comportamento.
Queste strutture sono in grado di apprendere, tramite degli esempi, le relazioni che vi sono tra le informazioni in input $(x_1, x_2, \ldots, x_n)$ e quelle in output y , basandosi anche su dati non completi o soggetti a rumore, e di adattarle ad un nuovo e più ampio insieme di dati in input, calcolando le relative uscite. Le informazioni provenienti dall'input vengono ponderate in base a dei pesi $(w_1, w_2, \ldots, w_n)$.

Per introdurre una non linearità nel modello e/o per assicurarsi che determinati segnali rimangano all'interno di specifici intervalli, viene utilizzata una funzione di attivazione. La funzione di attivazione non lineare più comunemente usata è la sigmoide: $f(z) = \frac{1}{1+e^{-z}}$. Successivamente, l'output y_k del neurone è uguale a : $y_k = \frac{1}{1+e^{(-\sum_j w_j x_j)}}$.

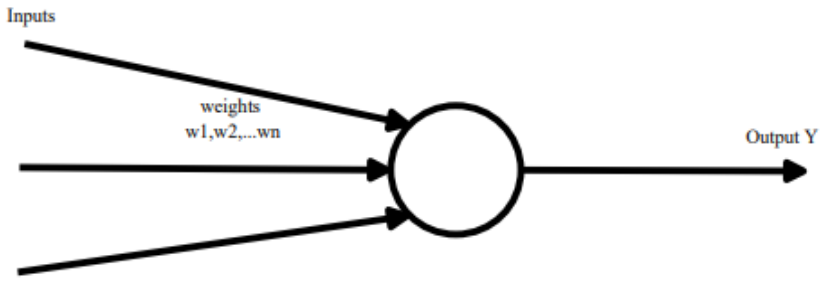

Figura 20: *Esempio di neurone artificiale di una rete neurale*

Per costruire una rete neurale, i neuroni sono collegati tra loro usando degli *edges* orientati. La maggior parte dei modelli limita le reti ad essere acicliche per un facile apprendimento. Le reti possono essere divise in livelli in base alla distanza dall'input. Ci sono due livelli principali: il livello di input e il livello di output. Il livello di input è quello in cui vanno messi gli elementi d'ingresso e il livello di output è quello in cui si ottengono i risultati. Il resto dei livelli sono chiamati livelli nascosti, come mostrato nella **Figura 21**.

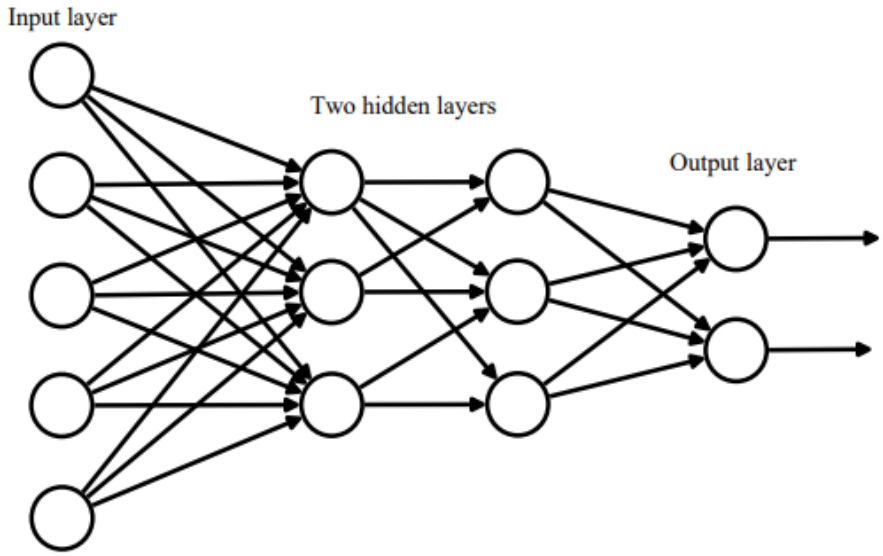

Figura 21: *Esempio di rete neurale con due livelli nascosti*

Nel processo di apprendimento modifichiamo iterativamente i pesi della rete, al fine di minimizzare una funzione di errore E (come $E = \sum_{i=1}^{n}(d_i - y_i)^2$, dove d_i è l'uscita desiderata del neurone i , y_i è l'uscita effettiva del neurone i , e n è il numero di neuroni nello livello di output). I pesi sono ottimizzati da un algoritmo di discesa del gradiente, cioè viene sottratto (un piccolo multiplo di) il gradiente della funzione di errore al peso w_i dal peso w_i in ciascun passo.

$$W_i^{(j)} \leftarrow W_i^{(j)} + c_i^{(j)} \delta_i^{(j)} X^{(j-1)}$$

Questo approccio è chiamato *backpropagation*.

Il compito di addestrare la rete consiste nella selezione di una topologia appropriata e nella messa a punto dei pesi delle sinapsi. Questo compito è molto complesso, poiché esiste un infinito numero di possibili strutture per la rete. Di conseguenza, la struttura viene solitamente decisa in modo arbitrario e l'algoritmo di apprendimento assegna solo i pesi effettivi.

I vantaggi offerti dalle reti neurali sono molteplici: possono essere aggiornate semplicemente e automaticamente per tenere il passo con l'evoluzione dei comportamenti del sistema, imparano relazioni non lineari tra i dati che sarebbero difficili da esprimere altrimenti, e sono uno strumento ampiamente accettato e compreso; tuttavia, per grandi moli di dati, l'algoritmo può essere soggetto a problemi a livello computazionale.

2.4 - Reti Bayesiane

Le reti di Bayes sono uno strumento che modella l'incertezza in un sistema rappresentato come un insieme di variabili aleatorie correlate tra loro. In un Grafo Orientato Aciclico, cioè un grafo dotato di archi orientati e privo di cicli diretti, ad ogni nodo è associata una variabile casuale e la sua Tabella delle Probabilità Condizionate, che esprime il legame di dipendenza condizionale nei confronti delle variabili rappresentate dai nodi genitori. Gli archi rappresentano le relazioni di dipendenza condizionale tra le variabili.

A partire dai nodi di evidenza, si calcolano le probabilità che certe ipotesi di interesse siano verificate, adottando un processo inferenziale che ricalca il ragionamento diagnostico. Quindi, la distribuzione congiunta di probabilità di un insieme di variabili aleatorie $X = \{X_1, \dots, X_n\}$ è rappresentata come il prodotto delle distribuzioni di probabilità:

$$p(X) = \prod_{i=1}^{n} p\,(X_i | Pa_i)$$

dove $p(X_i|Pa_i)$ è la distribuzione locale di probabilità associata alla variabile aleatoria X_i e condizionata dalle variabili Pa_icorrispondenti ai nodi genitori.
La mancanza di un arco tra due nodi riflette la loro indipendenza condizionale. Al contrario, la presenza di un arco dal nodo X_i al nodo X_j può essere interpretata come il fatto che X_i sia causa diretta di X_j .

Al fine di specificare completamente la rete bayesiana, è necessario specificare ulteriormente per ciascun nodo X_i la distribuzione di probabilità per X_i condizionata ai suoi genitori. In questo modo è possibile utilizzare una rete bayesiana per eseguire qualsiasi inferenza probabilistica sulle variabili del dominio.

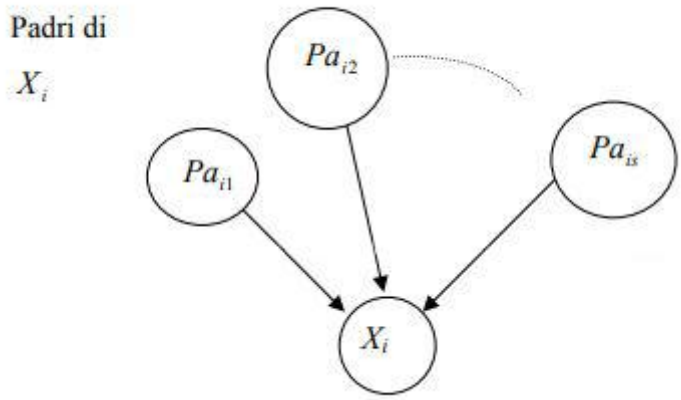

Figura 22: *Esempio di Grafo Orientato Aciclico*

Il processo di learning delle reti bayesiane comprende:

- **learning structure** (o structural learning): apprendere la struttura della rete, ovvero le relazioni fra le variabili;
- **learning parameters**: apprendere i parametri.

Lo scenario dell'apprendimento può variare a seconda che:

1. le informazioni sulla struttura del modello siano o meno disponibili:
    a. **known structure**: la struttura della rete è nota, per esempio è fornita da un esperto;

    b. **unknown structure**: bisogna apprendere prima la struttura della rete e dopo è possibile apprendere i parametri.
2. il database delle informazioni sia:
    a. completo;
    b. con dati mancanti.

A seconda che i dati siano completi o incompleti e che la struttura sia nota si utilizza l'algoritmo più appropriato.

Le reti bayesiane sono uno strumento sempre più flessibile ed adatto per il *problem solving* e diversi sono i motivi che portano alla scelta delle stesse per estrarre informazioni dai dati:

1. Esse permettono di apprendere le relazioni causali. L'apprendimento delle relazioni causali è importante perché aumenta il grado di comprensione del dominio di un problema e permette di fare predizioni. L'uso di reti Bayesiane permette di risolvere problemi simili anche quando non è disponibile nessun esperimento a riguardo.

2. Le reti Bayesiane, in congiunzione con le tecniche statistiche di tipo Bayesiano, facilitano l'associazione fra la rappresentazione del dominio, la conoscenza a priori ed i dati. La conoscenza a priori del dominio è importante, specialmente quando i dati sono scarsi o costosi. Nelle reti Bayesiane, la semantica causale (rappresentazione grafica delle relazioni) e la possibilità di apprendere le probabilità condizionate rendono la codifica della conoscenza a priori particolarmente chiara.

3. I metodi Bayesiani, in congiunzione con le reti Bayesiane, offrono un approccio efficiente per evitare l'*overfitting* dei dati.

CAPITOLO 3: Architettura del Sistema

In questo capitolo verrà mostrata l'architettura del sistema che è stata progettata e le scelte progettuali effettuate per rendere più performante il sistema e per garantire la sicurezza del sistema di analisi.

In particolare, verrà illustrata l'architettura *Client-Server* che è stata progettata *ad hoc* per garantire un servizio di analisi dinamica di *malware* su un *Server*, a cui un possibile *Client* possa accedere in remoto per usufruirne.

Un altro aspetto è la configurazione del *firewall* del sistema, che è stata effettuata per garantire la sicurezza dell'esecuzione della macchina virtuale, evitando possibili infezioni.

Per effettuare le analisi, è stato utilizzato il *software* CuckooDroid, che ha permesso di estrarre le informazioni necessarie per descrivere il comportamento dei *malware*.

*** OMISSIS ***

CAPITOLO 4: Implementazione

In questo capitolo, verrà mostrata la parte riguardante l'analisi dei risultati ottenuti. In particolare, verranno mostrate le strutture dei due file risultanti dalle analisi, effettuate con il *software* CuckooDroid. La parte di analisi dei file "*report.json"* e "*droidmon.log"* è stata una parte fondamentale del progetto, poiché ha permesso di comprendere le informazioni caratterizzanti dei *malware* e, successivamente, poterle rappresentare per descriverne il comportamento.

4.1 - Configurazione della Sandbox

Per garantire un corretto funzionamento, ma, soprattutto, per scongiurare problemi di sicurezza del sistema, è stato necessario configurare appositamente il *firewall*, andando a modificare le *Iptables rules*.

Iptables è il firewall, da linea di comando, installato in modo predefinito su Ubuntu. Iptables raggruppa tutti i controlli che può fare sul traffico in entrata, nella cosiddetta *Chain INPUT*. I controlli sul traffico in uscita sono, invece, raggruppati nella *Chain OUTPUT*. La *Chain FORWARD* serve, per esempio, quando il traffico di dati non è indirizzato ad un destinatario, ma passa comunque per il suo computer.

Per gli scopi del progetto, è stato necessario isolare dalla rete esterna sia la macchina *host* che la macchina virtuale, per evitare che queste potessero instaurare

connessioni verso l'esterno, a causa dei comportamenti assunti dai *malware* durante la loro esecuzione all'interno della *sandbox*.

Sono state modificate soltanto le *rules* relative al traffico in entrata e a quello in uscita, con i seguenti comandi:

sudo iptables -A INPUT -p tcp --dport *5554* -j ACCEPT # *porta emulatore*
sudo iptables -A OUTPUT -p tcp --sport *5554* -j ACCEPT
sudo iptables -A INPUT -s *127.0.0.1/24* -p tcp --dport *27017* -j ACCEPT # *MongoDB*
sudo iptables -A OUTPUT -d *127.0.0.1/24* -p tcp --sport *27017* -j ACCEPT
sudo iptables -A INPUT -s *127.0.0.1* -j ACCEPT # *IP localhost*
sudo iptables -A OUTPUT -d *127.0.0.1* -j ACCEPT
sudo iptables -A INPUT -s *147.163.5.233* -j ACCEPT # *IP macchina host*
sudo iptables -A OUTPUT -d *147.163.5.233* -j ACCEPT
sudo iptables -A INPUT -p tcp --dport *8001* -j ACCEPT # *porta webserver*
sudo iptables -A OUTPUT -p tcp --sport *8001* -j ACCEPT
sudo iptables -A INPUT -p tcp --dport *2042* -j ACCEPT # *port ResultServer*
sudo iptables -A INPUT -p tcp --sport *2042* -j ACCEPT
sudo iptables -A INPUT -s *10.0.2.2* -j ACCEPT # *IP ResulServer*
sudo iptables -A OUTPUT -d *10.0.2.2* -j ACCEPT
sudo iptables -P INPUT DROP # *drop in ingresso*
sudo iptables -P OUTPUT DROP # *drop in uscita*

In un primo momento, la configurazione *network* prevista dalla guida ufficiale del *software CuckooDroid* non garantiva la massima sicurezza del sistema, poiché le applicazioni che erano in esecuzione nella *sandbox* riuscivano ad effettuare e portare a termine le connessioni in ingresso ed in uscita, compromettendo la sicurezza. Con le dovute accortezze e le corrette eccezioni, è stato possibile utilizzare il software CuckooDroid senza problemi di comunicazione, concedendo le connessioni agli indirizzi IP ed alle varie porte utilizzate dai vari componenti del software, indicati in precedenza. Infine, sono state bloccate tutte le restanti connessioni in ingresso ed in uscita, per garantire la

sicurezza dell'esecuzione della macchina virtuale. In questo modo, l'ambiente virtuale della *sandbox* viene mantenuto isolato dall'esterno, senza alcuna possibilità di portare a termine eventuali richieste di connessioni, sia in entrata che in uscita.

*** OMISSIS ***

4.4 - Ambiente di sviluppo

In questo paragrafo, verranno illustrati i *software* che sono stati utilizzati in questo progetto per effettuare l'analisi dei *malware* in ambiente *Android*. Oltre al *software* "CuckooDroid", utilizzato per l'analisi delle applicazioni .apk, che è stato già mostrato nel dettaglio nel Capitolo 3, è stato utilizzato il *software* "Android Studio", fondamentale per emulare l'ambiente protetto in cui eseguire i *malware*, quale la *sandbox*, e il *software* "Weka", utilizzato per l'apprendimento automatico.
Le estrazioni delle caratteristiche, le analisi e le classificazioni dei *malware* sono stati effettuati in ambiente Linux, con distribuzione Ubuntu 18.04.

4.4.1 - Android Studio

Android Studio è un ambiente di sviluppo integrato (IDE), progettato specificamente per lo sviluppo di applicazioni *Android*. Oltre a tutti gli strumenti di cui è dotato per la progettazione e sviluppo di applicativi per sistemi operativi *Android*, esso permette la creazione di emulatori di dispositivi *Android*, andando a scegliere sia i requisiti a livello *hardware*, sia la versione del sistema operativo, la risoluzione e tutti i vari settaggi in modo da simulare perfettamente un dispositivo reale.

L'utilizzo di questo *software* è stato necessario per il corretto funzionamento di CuckooDroid, poiché ha permesso la creazione ed utilizzo della *sandbox* in cui mandare in esecuzione i *malware* per effettuare l'analisi dinamica.

È stato, dunque, possibile creare una macchina virtuale utilizzando il dispositivo virtuale *Android*, il quale funziona come un comune dispositivo fisico e, quindi, il comportamento del *malware* risulta generalmente identico a quello che avrebbe in un sistema reale.

Gli applicativi, sia benigni che malevoli, si presentano nel tipico formato Android, identificato dall'estensione .APK, che sta per ***A**ndroid **P**ac**K**age*. Questo formato di file è una variante del formato .JAR, tipico formato per gli archivi dati compressi.

Per gli scopi del progetto, è stato utilizzato un dispositivo virtuale con un sistema operativo Android 4.1.2 Jelly Bean. La scelta della versione del sistema operativo è stata fatta principalmente in base alla compatibilità con il *software* CuckooDroid, ma anche perché si tratta di un sistema più leggero rispetto alle successive versioni, ma al tempo stesso funzionale per gli scopi.

Per effettuare le analisi e rendere la *sandbox* riutilizzabile, CuckooDroid sfrutta una caratteristica di Android Studio relativa alla gestione dei dispositivi virtuali, ovvero quella di creare una *snapshot* del sistema virtualizzato. La *snapshot* consiste in un'immagine del sistema virtualizzato relativa all'istante in cui è stata estratta e viene creata una sola volta durante la configurazione del sistema virtualizzato ed utilizzata successivamente per ripristinare lo stato iniziale della macchina e preparare il sistema per un'altra analisi. Ogni volta che viene avviata un'analisi, viene eseguito il malware all'interno della macchina virtuale. Non appena il sistema termina di analizzare file, Android Studio carica la *snapshot,* ripristinando lo stato del sistema e preparando la macchina virtuale per l'analisi successiva.

4.4.2 - Weka

Weka, acronimo di *Waikato Environment for Knowledge Analysis*, è un software open source per l'apprendimento automatico sviluppato nell'Università di Waikato in Nuova Zelanda. Weka è un ambiente software interamente scritto in Java che permette di applicare dei metodi di apprendimento automatico ad un set di dati (*dataset*), ed analizzarne il risultato.
Attraverso questi metodi, è possibile avere, quindi, una previsione dei nuovi comportamenti dei dati.

Weka accetta come *dataset* di *input* una tabella in cui le istanze corrispondono alle righe e gli attributi selezionati alle colonne. Il formato utilizzato in Weka per la lettura dei dataset è il formato ARFF (*Attribute Relationship File Format*), simile al formato CSV (*Comma Separated Value*), il quale è equivalente alla tabella di un *database* relazionale. Weka raccoglie diversi algoritmi di apprendimento automatico e contiene strumenti per il *pre-processing*, la classificazione, la regressione, il *clustering* e il *mining* dei dati.

È possibile utilizzare diverse metodologie per la classificazione, come la tecnica dello Split Percentuale, in cui viene scelta, appunto, una percentuale per scindere il *dataset* in una porzione per il *Training Set* e la restante parte per *Test Set*, oppure è possibile utilizzare un apposito *dataset* come *Training Set*, oppure come *Test Set* per un modello. Inoltre, è possibile utilizzare la tecnica *k-cross-validation*, dove *k* sono il numero di *folds*. La tecnica *k-cross-validation* consiste nella suddivisione del *dataset* in *k* parti di uguale dimensione dove, ad ogni passo, la *k*-esima parte del *dataset* viene utilizzata come insieme di validazione, mentre la restante parte costituisce l'insieme di addestramento. In questo modo, il modello viene addestrato per ognuno dei sottoinsiemi del *dataset*, evitando, quindi, problemi di *overfitting*. In questo modo si esclude iterativamente un gruppo alla volta e lo si cerca di predire con i gruppi non esclusi.

CAPITOLO 5: Risultati Sperimentali

*** OMISSIS ***

5.1 - Metriche

In questo paragrafo, viene descritta la metrica utilizzata dal *software* Weka per la rappresentazione dei risultati della classificazione.

Al termine della classificazione il software mostra la bontà del processo attraverso diversi risultati.
Nei risultati vengono indicati il numero di campioni classificati correttamente e il numero di campioni classificati in modo scorretto. Grazie a questi due risultati, il software calcola la precisione e l'errore, in percentuali. Weka, inoltre, utilizza diverse metriche quali l'errore quadratico medio, l'errore assoluto medio.

Errore assoluto medio (MAE)
Il MAE misura la grandezza media degli errori in un insieme di previsioni, senza considerare la loro direzione. Essa misura l'accuratezza per variabili continue.

MAE rappresenta la media sul campione di verifica dei valori assoluti delle differenze tra la previsione e l'osservazione corrispondente. Il MAE è un punteggio lineare, il che significa che tutte le differenze individuali sono ponderate equamente nella media.

Errore quadratico medio (RMSE)

Il RMSE è una regola di punteggio quadratica che misura la grandezza media dell'errore. Essa rappresenta la differenza tra la previsione e i corrispondenti valori osservati, calcolati su ogni quadrato e poi calcolati sulla media del campione. Infine, viene presa la radice quadrata della media. Poiché gli errori vengono elevati al quadrato prima di essere calcolati come media, il RMSE attribuisce un peso relativamente elevato a errori di grandi dimensioni. Ciò significa che RMSE è più utile quando errori di grandi dimensioni sono particolarmente indesiderati [68].

MAE e RMSE possono essere usati insieme per diagnosticare la variazione degli errori in un insieme di previsioni.

Il RMSE sarà sempre più grande o uguale al MAE; maggiore è la differenza tra loro, maggiore è la varianza nei singoli errori nel campione. Se RMSE = MAE, allora tutti gli errori hanno la stessa grandezza.

Considerati i seguenti valori di *output* di Weka:

TP = true positives: numero di esempi predetti positivi che sono veramente positivi
FP = false positives: numero di esempi predetti positivi che sono veramente negativi
TN = true negatives: numero di esempi predetti negativi che sono veramente negativi
FN = false negatives: numero di esempi predetti negativi che sono veramente positivi

$$\textbf{Precisione} = \frac{TP}{TP+FP}$$

$$\textbf{Accuratezza} = \frac{TP+TN}{TP+TN+FP+FN}$$

Il software, inoltre, utilizza un'ulteriore metrica chiamata *Kappa statistic.*

Kappa statistic è una metrica che confronta l'accuratezza osservata con la precisione prevista. Essa rappresenta il grado di accuratezza e affidabilità in una classificazione statistica [68].

Per questo lavoro, sono stati valutati e confrontati i vari approcci scegliendo la metrica della Precisione, considerando, quindi, quanti campioni sono stati classificati correttamente e quanti in modo scorretto, con valori in percentuale.

*** OMISSIS ***

CAPITOLO 6: Conclusioni

In questo progetto è stato presentato un sistema per l'analisi e riconoscimento di *malware* che fa uso di una sandbox per la registrazione del comportamento dei files tramite la loro esecuzione ed utilizza delle tecniche di *machine learning* per la classificazione.

*** OMISSIS ***

INDICE DELLE FIGURE

INDICE DELLE TABELLE

BIBLIOGRAFIA

[1] Afonso, V., Amorim, M., Gregio, A., Junquera, G., Geus, P. (2015). Identifying Android malware using dynamically obtained features.

[2] Arp, D., Sprientzenbarth, M., Hubner, M., Gascon, H., & Reick, K. (2014) DREBIN: Effective and explainable detection of Android malware in your pocket.

[3] Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for Android.

[4] Canfora, G., Lorenzo, A., Medvet, E., Mercaldo, F., & Vissagio, C. (2015a). Effectiveness of opcode n-grams for detection of multi-family Android malware.

[5] Canfora, G., Mercaldo, F., & Vissagio, C. (2015b). Mobile malware detection using op-code frequency histograms.

[6] Canfora, G., Mercaldo, F., & Visaggio, C. (2015c). Evaluating op-code frequency histograms in malware and third-party mobile applications.

[7] Chan, P. & Song, W. (2014). Static detection of Android malware by using permissions and API calls.

[8] Dini, G., Martinelli, F., Saracino, A., & Sgandurra, D. (2012). MADAM: a multilevel anomaly detector for Android malware.

[9] Dong-Jie, W., Ching-Hao, M., Te-En, W., Hahn-Ming, L., & Kuo-Ping, W. (2012). DroidMat: Android malware detection through manifest and API calls tracing.

[10] Jerome, Q., Allix, K., State, R., & Engel, T. (2014). Using opcode sequences to detect malicious Android applications.

[11] Kang, B., Kang, B., Kim, J., & Im, E. (2013) Android malware classification method: Dalvik bytecode frequency analysis.

[12] Kang, B., Yerima, S. Y., Sezer, S., & McLaughlin, K. (2016). N-gram Opcode Analysis for Android Malware Detection.

[13] Lindorfer, M., Neugschwandtner, M., & Platzer, C. (2015). MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis.

[14] Liu, X. & Liu, J. (2014). A two-layered permission-based Android malware detection scheme.

[15] Mas'ud, M., Sahib, S., Abdollah, M., Selamat, S., & Yusof, R. (2016). An evaluation of n-gram system call sequence in mobile malware detection.

[16] Pehlivan, U., Baltaci, N., Acartürk, C. & Baykal, N. (2014). The analysis of feature selection methods and classification algorithms in permission based Android malware detection.

[17] Puerta, J., Sanz, B., Santos, I., & Bringas, P. (2015). Using Dalvik opcodes for malware detection on Android.

[18] Rovelli, P. & Vigfusson, Y. (2014). PMBS: Permission-based malware detection system.

[19] Sanz, B., Santos, I., Laorden, C., Ugarte-Pedro, X., Bringas, P., & Alvarez, G. (2012) PUMA: Permission usage to detect malware in Android.

[20] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). Andromaly: A behavioral malware detection framework for android devices.

[21] Sharma, A. & Dash, S. (2014). Mining API calls and permissions for Android malware detection. Cryptology and Network Security.

[22] Su, X., Chuah, M., & Tan, G. (2012). Smartphone dual defence protection framework: detecting malicious applications in Andoid markets.

[23] Talha, K., Alper, D., & Aydin, C. (2015). APK Auditor: permission-based Android malware detection system.

[24] Titze, D., Stephanow, P., & Schütte, J. (2013). App-ray: User-driven and fully automated android app security assessment.

[25] Varsha, M., Vinod, P., & Dhanya, K. (2016). Identification of malicious android app using manifest and opcode features.

[26] Wei, X., Gomez, L., Neamtiu, I.m & Faloutsos, M. (2012). ProfileDroid: multi-layer profiling of android applications.

[27] Yerima, S., Sezer, S., & McWilliams, G. (2014). Analysis of Bayesian classification approaches for Android malware detection.

[28] Yerima, S., Sezer, S., & Muttik, I. (2015a) High accuracy Android malware detection using ensemble learning.

[29] Yerima, S., Sezer, S., & Muttik, I. (2015b). Android malware detection: An eigenspace analysis approach.

[30] Zhao, M., Ge, F., Zhang, T. & Yuan, Z. (2011). AntiMalDroid: An efficient SVM-based malware detection framework for android.

[31] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, Haojin Zhu. (2016) StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware.

[32] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, Bo Li. (2017). Automated Poisoning Attacks and Defenses in Malware Detection Systems: An Adversarial Machine Learning Approach

[33] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein and Yves Le Traon. (2016). AndroZoo: Collecting Millions of Android Apps for the Research Community

[34] Kopelo Letou, Dhruwajita Devi, Y. Jayanta Singh. (2013) Host-based Intrusion Detection and Prevention System (HIDPS)

[35] Suyash Jadhav. (2016) Advance Android PHAs/Malware Detection Techniques by Utilizing Signature Data, Behavioral Patterns and Machine Learning

[36] Steven Strandlund Hansen, Thor Mark Tampus Larsen. (2015) Dynamic Malware Analysis: Detection and Family Classification using Machine Learning

[37] Tina R. Patil, Mrs. S. S. Sherekar. (2013) Performance Analysis of Naive Bayes and J48 Classification Algorithm for Data Classification

[38] Gaganjot Kaur, Amit Chhabra. (2014) Improved J48 Classification Algorithm for the Prediction of Diabetes

[39] Nils J. Nilsson. (2019). Artificial Intelligence: A New Synthesis

[40] M. Nielsen. (2015) Neural Networks and Deep Learning

[41] J.C. Principe, N.R. Euliano, W.C. Lefebvre. (2017) Neural and Adaptive Systems: Fundamentals through Simulations

[42] Trevor Hastie, Robert Tibshirani, Jerome Friedman. (2001) The Elements of Statistical Learning Data Mining, Inference, and Prediction

[43] Tom M. (1997) Mitchell. Machine Learning

[44] Hans Hauska, Philip Swain. (1975) The Decision Tree Classifier: Design and Potential

[45] A. De Paola, S. Gaglio, G. Lo Re, M, Morana. (2018) A Hybrid System for Malware Detection on Big Data.

[46] A. De Paola, S. Favaloro, S. Gaglio, G. Lo Re, M, Morana. (2018) Malware detection through low-level features and stacked denoising autoencoders

[47] S. Gaglio and G. Lo Re, Advances onto the Internet of Things: How ontologies make the Internet of Things meaningful. Springer, 2014

SITOGRAFIA

[47] https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[48] https://source.android.com/security/reports/Google_Android_Security_PHA_classifications.pdf

[49] https://repo.xposed.info/module/de.robv.android.xposed.installer

[50] https://www.appelmo.com/2015/05/08/malware-android-le-6-tipologie-piu-diffuse-ed-i-metodi-per-contrastarli

[51] https://www.antimalware.news/some-common-and-popular-types-of-android-mobile-malware

[52] https://nakedsecurity.sophos.com

[53] https://www.kaspersky.com/resource-center/threats/mobile

[54] https://en.wikipedia.org/wiki/Mobile_malware

[55] https://github.com/ArtemKushnerov/az

[56] https://cuckoo-droid.readthedocs.io/en/latest/

[57] https://github.com/idanr1986/cuckoodroid-2.0

[58] https://it.wikipedia.org/wiki/Ubuntu

[59] https://en.wikipedia.org/wiki/Weka_(machine_learning)

[60] https://developer.android.com/studio/

[61] https://github.com/jgamblin/isthisipbad/blob/master/isthisipbad.py

[62] https://virusshare.com/

[63] https://nsec.sjtu.edu.cn/kuafuDet/download.html

[64] http://www.malgenomeproject.org/policy.html

[65] https://www.sec.cs.tu-bs.de/~danarp/drebin/

[66] http://www.pwnzen.com/

[67] http://contagiodump.blogspot.com/

[68] https://androzoo.uni.lu/

[69] https://spu.fem.uniag.sk