Università di Perugia
Dipartimento di Matematica e Informatica

Tesi triennale in ...

Attacchi Adversarial Machine Learning contro Sistemi di Malware Detection in Android

Relatore/i
Valentina Poggioni
Alina Elena Baia

Laureando
Nicolò Vescera

Anno Accademico 2020-2021

Indice

Immagini

Tabelle

Listings

Abstract

In questo documento andremo ad analizzare vari esempi, presenti in letteratura, di Attacchi effettuati a sistemi di Malware Detection per Android tramite tecniche di Adversarial Machine Learning. Vedremo le principali metodologie, sia quelle tradizionali che quelle baste su Machine Learning, per l'individuazione di Malware (Malware Detection System) evidenziandone i punti di forza e le debolezze. Esamineremo il concetto di Adversarial Machine Learning con le tipologie di attacco più comuni. Introdurremo il concetto di codice Malevolo (Malware) con alcuni cenni di storia, le varie forme che può assumere ed alcune delle principali tecniche per la sua generazione. Parleremo anche del sistema operativo Android descrivendone l'architettura con i vari livelli che la compongono, la storia con le varie caratteristiche che lo hanno reso celebre e i suoi software principali: le App.

A year spent in artificial intelligence is enough to make one believe in God.

Alan Perlis

1 Android

> As an open system, Android is not under the tight control of its creator, Google.
>
> Steven Levy

1.1 Introduzione

Il sistema operativo *Android* fu introdotto in un periodo dove il mercato era colmo di valide alternative ampiamente utilizzate: Nokia con *Symbian*, Apple con *iOS*, BlackBarry con *BlackBarry OS* e Microsoft con *Windows Mobile*. Cosa lo ha reso quindi il Sistema Operativo più adottato (al mondo) [9] su smartphone, dal 2011, e su tablet dal 2013 ? La *libertà*. Android è un software libero (*open source*), a differenza dei suoi rivali, e ciò permette alle case produttrici di utilizzarlo e modificarlo senza dover pagare *royalitis* [7]. Si basa su *kernel Linux* [2] rendendolo facilmente adattabile a moltissimi dispositivi ed hardware differenti. Ovviamente non è l'unico sistema ad essere *open source* e basato su *Linux*, un esempio è *Ubuntu Mobile* che non è riuscito a riscuotere lo stesso successo per via della sua *User Interface* e semplicità di utilizzo. Quest'ultima caratteristica, insieme alla potente SDK con una documentazione molto dettagliata, hanno nettamente influito sulla sua diffusione.

1.2 Storia

Questo sistema operativo fu inizialmente creato dalla *Android inc.* ma venne quasi subito acquistato da *Google* e rilasciato col nome di *Android Open Source Project* (AOSP) nel 2007 [2]. Parallelamente a questo evento venne fondata la *Open Handset Alliance* (OHA), un consorzio dedicato allo sviluppo e distribuzione di Android. Il software fu rilasciato sotto licenza *Apache* (gratis, libera e open source). La OHA è formata da diverse case produttrici di hardware, software e telecomunicazioni come *Google*, *Intel*, *NVIDIA*, *Qualcomm*, *Motorola*, *HTC*, *T-Mobile*, ecc. [2] Il suo obbiettivo è quello di sviluppare tecnologie che permettono di abbassare nettamente i tempi e i costi della creazione e distribuzione di dispositivi mobili e relativi servizi.

Android crebbe in modo molto rapido ed ancora oggi è supportato con frequenti aggiornamenti di sicurezza e nuove versioni. Quasi tutte hanno il nome

di un cioccolatino, caramella o di un alimento inerente al mondo dei dolci e le iniziali di ognuna sono ordinate alfabeticamente in modo crescente. Nelle ultime versioni, come si può vedere dalla seguente tabella, questa particolarità si è persa.

Versione	*Nome*
1.5	Cupcake
1.6	Donut
2.0	Éclair
2.2	Froyo
2.3	Gingerbread
4.0	Ice Cream Sandwich
4.1	Jelly Bean
4.4	KitKat
5.0	Lollipop
6.0	Marshmallow
7.0	Nougat
8.0	Oreo
9	Pie
10	10
11	11

Table 1: Tabella riassuntiva di tutte le versioni di Android

Il primo dispositivo commercializzato che utilizzava Android fu l'*HTC Dream* (2008) [9], noto anche con il nome di *T-Mobile G1* (negli Stati Uniti e in alcune parti dell'Europa) o Era G1 (in Polonia). Montava un processore Qualcomm *MSM7201A ARM11* a 528 MHz, 256 MB di memoria interna, espandibile fino a 16 GB con microSD, 192 MB di RAM e una batteria a ioni di litio (rimovibile) da 1150 mAh. Era dotato di uno schermo da 3,2 pollici LED con touchscreen capacitivo ad una risoluzione di 320x480 pixel. La scocca portante era completamente in plastica ed aveva una tastiera QWERTY fisica.

Successivamente, divenne una pratica comune per Google utilizzare il lancio dei suoi nuovi dispositivi come base per le nuove versioni di Android: un esempio è il *Nexsus 5* che fu introdotto nel 2013 [9] con *KitKat* (la allora nuovissima versione di Android), il *Nexsus 6* con *Lollipop* o il *Pixel* con *Nougat 7.1*.

Figure 1: HTC Dream

Questa immagine [9] mostra un esemplare di HTC Dream nella sua colorazione bianca con tastiera AZERTY.

1.3 Architettura

L'architettura del Sistema Operativo Android è suddivisa in 4 livelli principali [2]:

- Linux Kernel
- Libraries
- Application Framework
- Applications

Come visto nelle sezioni precedenti, la base di Android è il *kernel Linux*, un insieme di software (detti *driver*) che servono per interfacciarsi con l'hardware, permettono quindi la gestione (a basso libello) di elementi come: schermo, fotocamera, memoria, connettività, ecc.

Per accedere a tali funzionalità non lo si fa in modo diretto, ma sono presenti delle librerie, scritte in C, che operano come un intermediario (*astrazione* ad un livello superiore). È proprio questo lo scopo del livello *Libraries*. Per

semplificarne ancora di più l'utilizzo al programmatore, a queste librerie si accede tramite un *API* Java, il linguaggio principale utilizzato per scrivere applicazioni Android. Perciò, nel sistema è integrata una speciale *Java Virtual Machine* (JVM) chiamata *Dalvik*, creata da Google per poter funzionare anche su dispositivi dotati di risorse molto limitate. *Dalvik*, a differenza della JVM di *SUN* (ora parte di Oracle Corporation) e simili, esegue archivi di tipo *.dex*, un formato che risulta più compatto ed ottimizzato a differenza del *.class*. Non si ha accesso a tutte le API messe a disposizione da JavaSE o JavaME, ma è presente un loro sottoinsieme chiamato *Core Libraries*.
Questi due elementi fanno parte di un sottoinsieme del livello *Libraries* chiamato *Android Runtime*.

La parte più importante del livello *Application Framework* è l'*Activity Manager*, un particolare componente che si occupa della gestione del ciclo di vita delle applicazioni. È importante notare che ogni applicazione Android viene eseguita da un'istanza della Dalvik VM all'interno di una *sandbox*: un particolare spazio isolato dal resto del sistema in modo da aggiungere uno strato di sicurezza in più.

Infine, nell'ultimo livello risiedono le varie applicazioni installate nel sistema.

1.3.1 Dalvik VM e ART

Come scritto in precedenza, il linguaggio con cui vengono programmate le Applicazioni Android è Java e quindi si è introdotta la Dalvik VM all'interno del sistema. Questa non utilizza lo standard *bytecode* delle normali JVM ma ne ha uno proprio ideato apposta per le esigenze di Android. Questo è un sistema *multitasking* e permette alle applicazioni di essere *multithreaded* (di utilizzare più processori), in aggiunta, ogni app viene eseguita in una propria sandbox con la propria istanza della Dalvik VM. Sono proprio questi motivi che impongono alla Dalvik VM di essere leggera e di dover avere un basso *overhead*.

Questa VM compila i programmi Java quasi come tutte le altre, ha solo una differenza: invece di dare come risultato file *.class*, che poi verranno compressi in archivi *.jar*, crea file in formato *.dex* che saranno poi raggruppati in archivi *.apk* (il formato standard della applicazioni Android). Il formato *.dex* per risultare leggero contiene solo *informazioni uniche* [2]: se, per esempio, diversi file hanno in comune alcune stringhe, queste appariranno in un solo file .dex, mentre negli altri sarà presente il loro riferimento (*puntatore*). Lo stesso meccanismo è applicato per costanti, oggetti e funzioni.

Dalla versione *5.0 Lollipop* di Android la Dalvik VM è stata rimpiazzata con l'*Android Runtime* (ART) [2]. La sostanziale differenza tra le due è

quando effettuano la conversione da *bytecode* a *codice macchina*: la vecchia Dalvik utilizza una compilazione *Just In Time* (JIT) [9], durante l'esecuzione dell'applicazione effettua la conversione quando è necessario; ART usa una compilazione *ahead-of-time* (AOT) [9] cioè tutto il bytecode viene convertito durante la fase di installazione dell'applicazione. Ciò garantisce la necessità di impiegare meno risorse per l'esecuzione delle applicazioni e assicura una maggior fluidità e responsività del sistema.

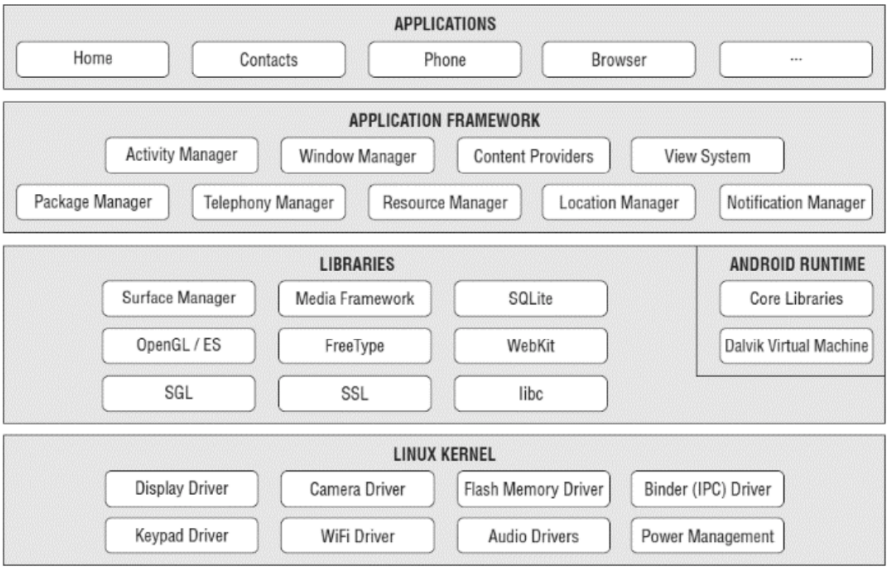


Figure 2: Architettura di Android

Questo è uno schema riassuntivo dell'architettura di Android prima della versione *Lollipop 5.0*

1.4 Componenti delle Applicazioni

Ogni applicazione Android è un archivio *.apk* che contiene vari file in formato *dex*. Ognuna di queste può svolgere innumerevoli funzioni, interagire con il sistema ed accedere a particolari tipi di dati condivisi. Di seguito andremo ad elencare ed analizzare i componenti principali che costituiscono ogni applicazione e che gli permettono di effettuare queste azioni.

1.4.1 Activities

L'*Activity* è l'elemento principale di un'applicazione, rappresenta ogni schermata dell'app [7], ne gestisce la logica di funzionamento e le interazioni con

l'utente. Saranno presenti tante *Activity* quante schermate. Queste non sono altro che un file Java.

È possibile definire l'aspetto di un'*Activity* (i colori, gli elementi presenti, la loro posizione, ecc.) tramite un apposito file *XML* (unico per ognuna) chiamato *Layout* o tramite apposite funzioni e oggetti all'interno del codice Java.

1.4.2 Intents

Un *Intent* è un meccanismo che serve per descrivere un'azione [7]: passare da un'Activity ad un'altra, scattare una foto, scegliere un'immagine dalla galleria, accedere alla posizione del GPS, ecc. La creazione e l'utilizzo di un *Intent* avvengono tramite apposite funzioni ed oggetti messi a disposizione dall'SDK.

```
1     Intent intent = new Intent(
2         this,
3         DisplayMessageActivity.class
4     );
5     startActivity(intent);
```

Listing 1: Esempio di codice Java per l'avvio di un'Activity tramite Intent

1.4.3 Services

In Android è possibile visualizzare ed interagire con una sola app alla volta, il che pone un grande limite nell'utilizzo. Per risolvere questo problema è possibile creare, per ogni applicazione, un *Service* [7]: una specie di *Unix Daemon*, privo di user interface, che rimane in esecuzione in background anche quando l'applicazione non è visibile o se ne sta utilizzando un'altra.

Un esempio è il lettore musicale che riesce a riprodurre file audio anche quando il telefono è bloccato o si utilizzano altre applicazioni.

1.4.4 Content Providers

Il *Content Provider* è una parte di un'applicazione che si occupa di gestire un determinato set di dati e di regolarne gli accessi dalle altre app [7].

Un esempio è il *Content Provider* dell'app *Contatti* che gestisce gli accessi, provenienti da altre applicazioni, ai dati salvati nella rubrica del telefono.

```
1       Cursor cursor = getContentResolver().query(
2           UserDictionary.Words.CONTENT_URI,
3           projection,
4           selectionClause,
5           selectionArgs,
6           sortOrder
7       );
```

Listing 2: Esempio di codice scritto in Java per una richiesta al Content Provider di User Dictionary

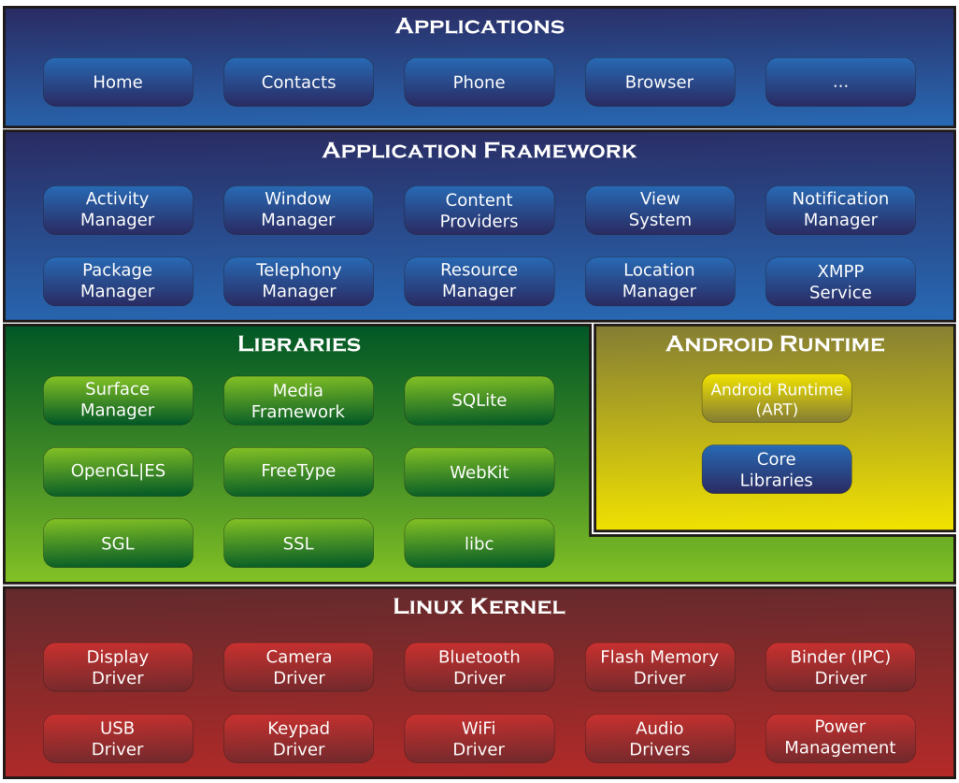


Figure 3: Nuova architettura di Android

Questo è uno schema riassuntivo della nuova architettura di Android con la nuova ART.

1.5 Struttura di un Progetto Android

Android Studio è l'IDE che Google fornisce per la creazione di app per Android. Ogni progetto ha una struttura di cartelle e file ben precisa e complessa che, tramite Android Studio, viene semplificata e mostrata all'utente

raggruppando i vari elementi in *moduli* [3]. I principali moduli sono:

- **manifest**: qui è presente il file *AndroidManifest.xml* al cui interno sono presenti dati importanti come il nome di ogni Activity, il nome del progetto, la dichiarazione di vari Intent, gli elementi hardware necessari al funzionamento, i vari permessi a cui l'app deve avere accesso, ecc.
- **java**: questo modulo contiene tutti i file Java raggruppati nei vari *package* (se presenti).
- **res**: in questa parte sono contenuti tutti i file che non presentano codice al proprio interno: layout XML, immagini, set di colori, icone e altri elementi simili.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk
    /res/android"
3     package="com.example.toggletest"
4     android:versionCode="1"
5     android:versionName="1.0" >
6
7     <application
8         android:allowBackup="true"
9         android:icon="@drawable/ic_launcher"
10        android:label="@string/app_name"
11        android:theme="@style/AppTheme" >
12        <activity
13            android:name="com.example.toggletest.
    MainActivity"
14            android:label="@string/app_name" >
15            <intent-filter>
16                <action android:name="android.intent.
    action.MAIN" />
17
18                <category android:name="android.intent.
    category.LAUNCHER" />
19            </intent-filter>
20        </activity>
21    </application>
22
23    <uses-permission android:name="android.permission.
    ACCESS_NETWORK_STATE" />
```

```
24      <uses-permission android:name="android.permission.
        ACCESS_WIFI_STATE" />
25
26 </manifest>
```

Listing 3: Esempio di AndroidManifest.xml

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <LinearLayout xmlns:android="http://schemas.android.com
       /apk/res/android"
 3                 android:layout_width="match_parent"
 4                 android:layout_height="match_parent"
 5                 android:orientation="vertical" >
 6     <TextView android:id="@+id/text"
 7                 android:layout_width="wrap_content"
 8                 android:layout_height="wrap_content"
 9                 android:text="Hello, I am a TextView" />
10     <Button android:id="@+id/button"
11                 android:layout_width="wrap_content"
12                 android:layout_height="wrap_content"
13                 android:text="Hello, I am a Button" />
14 </LinearLayout>
```

Listing 4: Esempio di Layout XML

```
 1 package com.example.myproject
 2
 3 import android.widget.TextView
 4
 5 public class MainActivity extends Activity {
 6     private TextView myTextView;
 7
 8     @Override
 9     public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_main);
12
13         myTextView = findViewById(R.id.mytexview);
14
15         myTextView.setText("Hello World !");
16     }
17 }
```

Listing 5: Esempio di codice di un'Activity scritto in Java

2 Malwares

> The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards.
>
> Gene Spafford

2.1 Introduzione

"malware continues to grow" (*i malware continuano ad aumentare*), è ciò che afferma un recente report di McAfee [6]. Nell'ultimo periodo, grazie all'aumento dei dispositivi elettronici (computer, smartphone, ecc.) e soprattutto di servizi online come *smart working* e *home banking*, il numero di malware è sempre in aumento. Si stanno dunque cercando di studiare e sviluppare nuovi sistemi robusti ed efficaci per contrastare queste minacce.

In questa sezione andremo ad analizzare cos'è un Malware, le varie tipologie, i classici Sistemi anti-malware, i nuovi basati su Machine Learning e i loro punti deboli con possibili attacchi presenti in letteratura.

2.2 Definizione di Malware

Il termine "Malware" deriva dall'unione di due parole [6]: "malicious" e "software" (*software malevolo*) e sta ad indicare qualunque tipo di software indesiderato. Un'altra definizione di Malware, creata da G. McGraw e G. Morrisett [6] è: "qualunque pezzo di codice aggiunto, modificato o rimosso da un software, in modo da danneggiare o alterare il funzionamento del sistema".

Ogni malware ha le seguenti caratteristiche:

- **Riproduzione**: questa è la caratteristica più importante per un malware, dato che gli permette di riprodursi e continuare la sua vita. In alcuni casi, l'eccessiva riproduzione arriva a saturare le risorse della macchina (come RAM e Hard Disk).
- **Invisibilità**: tramite questa proprietà i malware riescono a sfuggire ai sistemi di individuazione delle minacce. Può essere realizzata tramite tecniche *polimorfiche* o *metamorfiche*.

- **Propagazione**: un malware deve essere in grado di propagarsi da un dispositivo ad un altro per cercare di infettare più macchine possibili. Un modo molto comune che hanno di spostarsi da un dispositivo infetto ad un altro è tramite la rete (locale o globale) o tramite filesystem locali o remoti.

- **Auto-esecuzione**: una volta raggiunto il sistema bersaglio, un malware deve essere in grado di auto avviarsi per poter dare inizio al suo ciclo vitale.

- **Corruzione del sistema**: un software di questo tipo può anche andare a minare l'integrità, l'accessibilità e la riservatezza dei dati presenti nel dispositivo attaccato.

2.3 Creazione di un Malware

Per creare un Malware si possono utilizzare varie tecniche [6], da quelle più semplici che consistono nell'inserimento di una speciale parte di codice all'interno di un programma, fino a quelle più complesse che utilizzano algoritmi sofisticati per generare minacce di tipo *Obfuscated* o *Polymorphic*.

Di seguito le andremo ad analizzare brevemente.

2.3.1 Metodi Basilari

Come detto prima, queste tecniche consistono nell'inserimento ti parti di codice malevolo all'interno di un altro software. I Malware creati in questo modo possono essere facilmente individuati estraendo dal codice alcune caratteristiche univoche che prendono il nome di "signature".

2.3.2 Polymorphic

Questo metodo consiste nel riuscire a creare programmi che riescono a mutare, ad ogni nuova infezione, la sintassi della parte di codice malevola, lasciandone inalterata la semantica. La crittografia è la tecnica più utilizzata per generare questo tipo di minacce.

2.3.3 Obfuscation

Infine, i Malware prodotti con questa tecnica riescono a trasformare la forma del codice in modo da lasciare inalterate le sue funzionalità ma rendendolo

molto complesso da leggere e interpretare. Alcuni modi per realizzare la Obfuscation sono:

- **Dead-code**: inserimenti di codice totalmente inutile per rendere difficile e incomprensibile la lettura
- **Code Trasportation**: aggiunge salti incondizionati al codice, rendendolo molto complesso, ma lasciandone il flusso di esecuzione originale invariato
- **Register Renaming**
- **Toolkit Paradigm**

2.4 Tipologie di Malware

Esistono svariati metodi per la classificazione dei Malware, il cui scopo è quello di aiutare il tracciamento della paternità, correlazione e identificare la nascita di nuove varianti. In questo documento non utilizzeremo tali classificazioni ma faremo la distinzione tra malware che utilizzano la *Rete Internet* come mezzo di propagazione o di esecuzione e quelli che non la utilizzano. Chiameremo i primi *Network-based Malware*[6] e gli altri *Ordinary Malware*[6].

2.4.1 Network-based Malware

Spyware
Sono software installati in segreto nella macchina di un utente utilizzati per raccogliere dati a sua insaputa. Anche aziende come Microsoft e Google utilizzano spyware per raccogliere informazioni in segreto[6].

Adware
Questo tipo di software apre in automatico pubblicità e annunci senza il consenso dell'utente. Generalmente sono innocui e puntano solo a far guadagnare il proprietario tramite la pubblicità (che molto spesso si presenta come un pop-up che interrompe l'attività dell'utente), ma ne esistono alcune versioni contenenti spyware, come keylogger, che minacciano la privacy della vittima.

Cookies
I *Cookies* sono informazioni che il browser salva nella macchina dell'utente per vari motivi: salvare le preferenze per i vari siti, mantenere dati per le sessioni server-based, autenticare l'utente, ecc. Di norma questi non sono

dannosi e soprattutto non possono essere eseguiti, ma potrebbero rimanere per sempre nel computer della vittima. Esistono dei Malware che rubano e utilizzano le informazioni presenti all'interno di questi Cookies.

Backdoors
Le *Backdoors*, anche chiamate *Trap Doors*, sono del codice malevolo inserito all'interno di un'applicazione o un sistema operativo che garantiscono all'attaccante l'accesso al sistema della vittima senza dover passare per i consueti metodi di autenticazione.

Trojan Horse
All'apparenza sembra software utile, ma in realtà il suo scopo è quello di rubare informazioni dell'utente o corrompere i dati.

Sniffers
Gli *Sniffer* sono programmi che riescono ad intercettare e salvare il traffico che avviene all'interno di una rete. Catturano ogni pacchetto trasmesso o ricevuto, ne analizzano il contenuto e ricavano ogni dato e informazione sensibili presenti al loro interno. Generalmente l'utilizzo degli Sniffer è il primo passo di un *Intrusion Attack*.

Spam
È uno speciale software che invia a numerosi utenti email contenenti lo stesso messaggio con lo scopo di intasare la casella di posta della vittima, propinare truffe di vario tipo e, a volte, favorire la propagazione di altri malware.

Botnet
Le *Botnet* sono un insieme di computer infettati (contengono un software chiamato "bot") che possono essere controllati da remoto dall'attaccante. Generalmente vengono utilizzate per performare attacchi di tipo DoS (*Denial of Service*).

2.4.2 Ordinary Malware

Virus
I *Virus* sono software in grado di riprodursi da soli, durante la fase di infezione, ed annidarsi in altre applicazioni o documenti. Hanno lo scopo di danneggiare il dispositivo che li ospita (possono arrivare fino alla distruzione dell'intero sistema) e di infettare le periferiche di archiviazione con cui entra in contatto. Un virus viene inserito all'interno di un file tramite tre tecniche:

1. **Pre-pending**
2. **Embedding**
3. **Post-pending**

Un esempio di ciò è la modifica del file *Autorun.inf*. Questo file risiede in ogni periferica di memoria che contiene musica e serve a riprodurla in automatico quando questa viene collegata al computer. Quando la periferica viene inserita, il sistema operativo va a cercare il file Autorun.inf e lo esegue. Aggiungendo virus all'intero di questo file, l'infezione è garantita. Questo attacco può essere individuato tramite un sistema "signature-based" (se le signature sono fornite).

Worm
Questi software sono in grado di auto-replicarsi, cioè, creare molteplici copie di se stessi e nasconderle all'interno del computer della vittima. Gli *AV Scanner*, sfruttando questa caratteristica, possono riuscire ad individuarli: se sono presenti numerosi file dalle caratteristiche molti simili, allora questo è sintomo di un infezione da worm.

Logic Bomb
I malware di tipo *Logic Bomb* sono programmi che rimangono inattivi fino quando una determinata condizione non si verifica. Solitamente è la data e l'ora. Questi codici la controllano periodicamente, tramite il sistema operativo, per capire se è il momento di attivarsi. Quando il dato evento sopraggiunge, riescono ad eseguire il proprio codice in automatico.

2.5 Malware Detection Systems

Un *Malware Detection System* è un programma in grado di individuare malware presenti in un sistema, qualora ce ne fossero. Una definizione formale di Malware Detection System è la seguente [6]:

Un programma di Malware Detection D è una funzione che lavora su un dominio P che contiene applicazioni *benigne* e *maligne*. La funzione D analizza i programmi $p \in P$ classificandoli in *benigni*, se sono programmi normali oppure *maligni* se contengono un malware. Più in breve:

$$D(p) = \begin{cases} maligno, & \text{se } p \text{ contiene codice malevolo} \\ benigno, & \text{altrimenti} \end{cases}$$

La precedente funzione di Malware Detection potrebbe risultare in problemi generando *falsi negativi*, *falsi positivi* o *indecisioni* dipendentemente dalla sua efficienza. Possiamo dunque riscriverla come:

$$D(p) = \begin{cases} maligno, & \text{se } p \text{ contiene codice malevolo} \\ benigno, & \text{se } p \text{ non contiene codice malevolo} \\ indeciso, & \text{se } D \text{ non riesce a classificare } p \end{cases}$$

La classe *indeciso* viene assegnata quando non si riesce a capire la natura del programma, generalmente accade quando si incontrano malware nuovi che non sono stati ancora scoperti. Un *falso positivo* è un file innocuo che viene catalogato come malware, infine un *falso negativo* è un malware che viene catalogato come programma benigno.

2.5.1 Storia dei Sistemi di Malware Detection

Il primo programma anti-malware fu "Flushot Plus" creato da Ross Greenberg nel 1987 [6]. Il suo scopo era quello di impedire ai vari malware di modificare il contenuto dei file.

Nel 1989, John McAfee rilasciò "VirusScan™", un software in grado di individuare ed eliminare diversi virus contemporaneamente.

Successivamente ci fù "Wisdom & Sense (W&S)", un *Statistic-based Anomaly Detector* creato al Los Alamos National Laboratory.

Poi vennero anche "Time-based Inductive Machine (TIM)", "Network Security Monito (NSM)", "Information Security Officer's Assistant (IOSA)".

2.5.2 Tecniche e Metodologie

Generalmente, i produttori di Malware Detection Systems, tengono traccia dei nuovi programmi, li analizzano e li aggiungono in apposite liste [6]:

- **White List**: qui ci vanno i software innocui
- **Black List**: questo è il posto per i malware
- **Gray List**: qui aggiungono i software che generano *indecisioni*

Per questi ultimi, quando un software entra a farvi parte, viene fatto eseguire in un ambiente controllato e isolato per poterlo classificare con più precisione. Qualora uno di questi programmi venisse rilevato come malware, i produttori del sistema rilascerebbero un aggiornamento permettendo così agli utenti di scaricarlo ed avere sempre un database delle minacce aggiornato.

Qui di seguito andremo ad analizzare le più comuni tecniche di Malware Detection.

Signature-based & Anomaly-based
Tutti i malware scanner utilizzano tecniche *signature-based* e *anomaly-based* per andare ad identificare la natura di un programma. Queste possono essere applicate in maniera *dinamica* o *statica*. La prima necessita di far eseguire il programma e ricava informazioni a *run time*, l'altra estrapola le informazioni senza la necessità di avviare il file. Infine, ne esiste un'altra, chiamata *ibrida*, che utilizza una combinazione del metodo *statico* e di quello *dinamico*.

La tecnica *Signature-based* consiste nel confrontare l'impronta del file, chiamata anche *signature* (firma), con altre impronte di malware presenti in un database. Il suo vantaggio è una buona efficacia, ma ha il grande problema che non riesce ad individuare nuovi malware non ancora presenti all'interno di questi database.

I sistemi che utilizzano la metodologia *Anomaly-based* invece, catalogano i programmi in base al loro comportamento: analizzano le varie azioni che compiono cercando di individuare procedure sospette che trascendono dal normale utilizzo della macchina.

Heuristic-based
L'*Intelligenza Artificiale* (AI) è stata molto utilizzata insieme alle tecniche Signature-based e Anomaly-based per aumentare la loro efficienza. Vengono utilizzate *Reti Neurali* (NN) soprattutto per la loro caratteristica di adattarsi bene ai cambiamenti e per l'abilità di effettuare predizioni. Anche gli *Algoritmi Genetici* sono impiegati nei problemi di malware detection, per derivare

regole di classificazione e per la scelta di features e parametri. Questi, per funzionare, applicano principi della biologia evolutiva quali: *Ereditarietà*, *Mutazioni*, *Selezione* e *Combinazione*.
Modelli *Statistici e Matematici* vengono impiegati anche in questo campo, applicandoli ad informazioni del sistema come: connessioni attive, bandwidth, utilizzo della memoria, chiamate di sistema, ecc.

2.5.3 Tecnologie

Host-based
I sistemi di detection Host-based, monitorano costantemente e dinamicamente lo stato e il comportamento del sistema, controllando se ci sono attività interne o esterne che mettono a rischio l'integrità della macchina. Questi vengono anche chiamati "in-the-box" perché risiedono all'interno dello stesso dispositivo che proteggono. Il vantaggio di questi sistemi è che sono in grado di proteggere l'host, in maniera molto efficacie, dall'interno, ma sono deboli a contrastare attacchi esterni.

Network-based
Queste tecnologie prendono il nome di *Network-based Intrusion Detection Systems* (IDS) e vengono utilizzate per "sniffare" tutti i pacchetti da e per ogni nodo della rete ed analizzarli. Possono esserci moduli di questo tipo per ogni segmento di rete (che monitorano il traffico di quel pezzo) oppure un modulo per ogni nodo. Questi vengono anche chiamati "out-of-the-box" dato che risiedono fuori dal dispositivo che proteggono. Il loro vantaggio è di riuscire a prevenire attacchi dall'esterno, ma non riescono a proteggere da attacchi interni.

Hybrid-based
Un approccio misto tra Host-based e Network-based può essere utilizzato. Ogni nodo della rete ha un particolare modulo che raccoglie informazioni e le manda ad un dispositivo principale che ne effettua l'analisi e la classificazione. L'approccio ibrido riesce a sopperire ai difetti delle singole tecnologie utilizzate.

Virtual Machines
Le *Virtual Machine* (VM) possono essere utilizzate nell'ambito della malware detection in quanto permettono di avere una copia della macchina reale in grado di essere isolata da quest'ultima e garantiscono un'elevata sicurezza, efficienza di esecuzione ed il pieno controllo delle risorse. Queste servono come ambiente sicuro dove far eseguire programmi per poi analizzarli e catalogarli.

Alcune tipologie di macchine virtuali sono:

- **Sandbox**: un ambiente controllato dove le varie risorse sono accessibili solo tramite tramite API fornite dalla VM. Qui sono fatti eseguire i vari software sospetti che vengono monitorati, analizzati ed infine viene stilato un report finale.
- **Emulation**: viene simulato l'intero sistema del computer (*emulation*) per poterci far eseguire un altro sistema operativo (*guest*) che risulta isolato dal sistema originale (*host*).

Web-based
Esistono siti web che hanno la capacità di effettuare scansioni dell'intero sistema di un computer, di hard disk, aree del sistema a rischio, cartelle e file. Questa è una buona soluzione per chi non vuole installare anti-malware nel proprio dispositivo dato che questi possono essere disabilitati da alcuni tipi di software malevoli.

Application Protocol-based
Application Protocol-based Intrusion Detection System (APIDS) è uno speciale sistema di monitoraggio che analizza uno specifico protocollo applicativo. Questo sistema è formato da un *agent* posizionato tra diversi gruppi di server che monitora costantemente lo stato di un determinato protocollo applicativo utilizzato da questi. Un esempio è un APIDS posto tra un Web Server ed un Database Management System che analizza il traffico del protocollo SQL.

3 Adversarial Machine Learning

> "The potential benefits of artificial intelligence are huge, so are the dangers.
>
> Dave Waters

3.1 Introduzione

Negli ultimi anni l'uso del *Machine Learning* si sta diffondendo sempre di più: dai classificatori che riescono a distinguere un cane da un gatto (*Image Classification*), fino ad importanti sistemi di sicurezza come identificazione facciale (*Face Detection/Recognition*) e individuazione di minacce (*Malware Detection*).

La sicurezza e la robustezza di questi algoritmi viene messa a rischio da attacchi effettuati tramite l'*Adversarial Machine Learning* [4]: una particolare tipologia di Machine Learning che mira a violare il corretto funzionamento di uno (o più) di questi meccanismi.

Il funzionamento può essere descritto come segue: sia M un sistema di Machine Learning e sia C un input, che chiameremo "clean example", assumiamo che C sia classificato correttamente dal sistema: $M(C) = y_{true}$. È possibile costruire un input A, che chiameremo "adversarial example", praticamente identico a C ma che viene classificato in modo errato: $M(A) \neq y_{true}$. L'obbiettivo dell'Adversarial Machine Learning è quindi quello di riuscire a creare l'input A per portare ad una classificazione errata il modello M.

3.2 Metodologie di Attacco

Generalmente l'Adversarial Machine Learning si concentra sulla modifica e alterazione di immagini. Ci sono 3 principali metodologie di attacco [4]:

1. **Poisoning**
2. **Evasion**
3. **Model Extraction**

Ognuna di queste va ad agire sulle varie debolezze del modello. Di seguito una breve descrizione di queste.

3.2.1 Poisoning

Questo tipo di attacco mira a violare la fase di *training* (addestramento) del modello introducendo un "adversarial input" nel dataset in modo tale da ridurre il più possibile l'*accuracy* per tutti i possibili input.

3.2.2 Evasion

L'obiettivo di questo attacco è quello di violare l'integrità di un modello: l'attaccante va a modificare un input cercando di ottenere come output una classe diversa dalla classe di appartenenza. In alternativa, si può anche tentare di ridurre la confidenza del modello per quell'input.

In modo più formale, possiamo descrivere questo attacco con il seguente problema di ottimizzazione:

$$argmin||\delta_X||s.tF(X + \delta_X) = Y^*$$

dove δ_X è un vettore che contiene la perturbazione da aggiungere all'input X, F è la funzione approssimata dall'algoritmo e Y^* è la classe obiettivo che l'attaccante vuole ottenere per X.

3.2.3 Model Extraction

A differenza dei precedenti attacchi, il *Model Extraction* (o anche *Model Stealing*) viene utilizzato per violare la "*confidentiality*" del modello: è possibile clonare un determinato modello, tramite un numero finito di interrogazioni e utilizzando gli output per addestrare un nuovo modello clone, o recuperare informazioni private e sensibili riguardanti il dataset utilizzato durante la fase di training.

3.3 Attacchi

La maggior parte degli attacchi di Adversarial Machine Learning appartengono alla tipologia numero 2 [4] vista in precedenza, *Evasion*, e vengono effettuate su immagini per cercare di violare sistemi di Image Classification e Object Detection. Di seguito andremo ad analizzare i principali tipi di attacchi appartenenti a questa categoria:

3.3.1 Fast Gradient Sign Method

Il *Fast Gradient Sign Method* (FGSM) è stato proposto da Goodfellow et al. (2014) come un modo semplice per generare "adversarial examples":

$$X^{adv} = X + \epsilon sign(\nabla_X J(X, y_{true}))$$

Il suo obbiettivo è quello di trovare una perturbazione/rumore che, aggiunto all'immagine, aumenti il valore della *loss function*. Questo metodo è più efficiente rispetto alla complessità computazionale se confrontato con metodi più complessi come il L-BFGS, ma generalmente ha una percentuale di successo inferiore.

3.3.2 One-step target class methods

Questo attacco si basa sul massimizzare la probabilità $p(y_{target}|X)$ di una specifica classe y_{target}, che risulterà improbabile essere la classe corretta per una data immagine.
Per una rete neurale che ha come *loss function* la *cross-entropy* questo porterà ad avere la seguente formula:

$$X^{adv} = X - \epsilon sign(\nabla_X J(X, y_{target}))$$

Come classe target possiamo usare la classe che ha minor probabilità di essere predetta dalla rete: $y_{LL} = argmin_y\{p(y|X)\}$. In questo caso l'attacco avrà il nome di *One-step Least Likely Class* ("step l.l."). In alternativa possiamo utilizzare anche una classe scelta in maniera casuale. Questo prenderà l nome di "step rnd."

3.3.3 Basic Iterative Method

Questa è una derivazione del metodo FGSM che consiste nell'applicarlo più volte con step di dimensioni inferiori:

$$X_0^{adv} = X, \quad X_{N+1}^{adv} = Clip_{X,\epsilon}\{X_N^{adv} + \alpha sign(\nabla_X J(X_N^{adv}, y_{true}))\}$$

3.3.4 Iterative least-likley class Method

Applicando più volte il metodo "step l.l" possiamo ottenere "adversarial examples" che vengono classificati in modo errato più del 99% delle volte:

$$X_0^{adv} = X, \quad X_{N+1}^{adv} = Clip_{X,\epsilon}\{X_N^{adv} - \alpha sign(\nabla_X J(X_N^{adv}, y_{LL}))\}$$

4 Attacchi ad Android Malware Detection Systems

> Adversarial examples for machine learning based detection are very much like the HIV which progres- sively disables human beings' immune system
>
> *Xiao Chen et al.* [1]

4.1 Introduzione

Android è il sistema operativo più utilizzato negli smartphone, nel 2018 ha raggiunto una quota di mercato del 84.8% [8], rendendolo così un bersaglio molto interessante per i malware.

Come visto nelle sezioni precedenti, sistemi di *Machine Learning* sono stati utilizzati per migliorare le prestazioni di software per la Malware Detection. Il loro principale svantaggio è l'inaffidabilità nella catalogazioni [8] di nuovi campioni mai visti prima. Questo li espone ad una grande vulnerabilità: gli *Adversarial Samples*, modifiche appositamente studiate ed applicate al codice di un malware per renderlo "invisibile" ai controlli.

La creazione di questi Adversarial Samples è stata ampiamente studiata e, di solito, viene applicata alle immagini: i pixel di una foto vengono distorti in modo tale da non permettere ad un occhio umano di percepire la modifica e riuscendo allo stesso tempo ad ingannare il classificatore. Analogamente, nell'ambito della malware detection, si punta ad inserire una perturbazione nel file malevolo in modo da farlo apparire come un file benigno. Questa modifica deve avere le seguenti tre caratteristiche [8]:

- Deve essere applicabile in modo veloce ed a basso costo (*low cost* e *low effort*)
- Non deve eliminare o modificare il comportamento originale del malware
- Si possono solo aggiungere o togliere elementi (*features*) al codice

4.2 Android Malware Detection con Machine Learning

Di solito, per il nostro problema, vengono impiegati algoritmi di Classificazione [8] per distinguere un malware da un normale software. Il Classificatore prende in input un file (*sample*) e lo trasforma in un *feature vector* che utilizzerà poi per effettuare la classificazione. Ci sono due modi per rappresentare un software come *feature vector*:

1. **Modelli Dinamici**: eseguono il codice, osservano il suo comportamento e ne estraggono le informazioni
2. **Modelli Statici**: analizzano il software trattandolo come un file binario. Questi sono nettamente più impiegati e studiati rispetto ai Modelli Dinamici.

Una volta estratte queste informazioni (*features*), ogni *sample* può essere rappresentato come un vettore $x \in X$ che ha come classe $y \in Y$ (per esempio 0 per un software benigno e 1 per uno maligno) ed utilizzati nel processi di *training* (per derivare la funzione $f : X \to Y$) e di *testing*.

I Classificatori che rappresentano lo stato dell'arte sono:

- Drebin
- Stormdroid
- MaMaDroid

4.3 Esempi di Adversarial Machine Learning

In questa sezione andremo a vedere alcuni esempi presenti in letteratura di *Adversarial Attacks* contro sistemi di Machine Learning per malware detection in Android.

Il metodo proposto da Papernot et al., *Jacobian-based Saliency Map Attack* (JSMA) [8], utilizza le *forward derivatives* del modello di classificazione per trovare la perturbazione ottima per la creazione di un adversarial sample. Grosse et al. lo hanno applicato al campo di Malware Detection riuscendo a raggiungere un *misclassification rate* dell'84%.

Carlini & Wanger hanno creato un modello di attacco *optimization-based*, conosciuto anche come *CW Attack*[8], che riesce a degradare le performance di classificazione di un modello raggiungendo un *success rate* anche del *100%*.

Chen et al. hanno ideato un attacco contro *MaMadroid* e *Drebin* basato su *CW* e *JSMA*. Il loro attacco genera *adversarial sample* aggiungendo *feature* binarie al codice del malware, raggiungendo un *success rate* del 100% quando si ha a disposizione il modello ed i suoi parametri (*withe box attack*).

Goodfellow et al. introdussero [8] l'utilizzo delle *Generative Adversarial Networks* (GANs) per generare *adversarial sample* partendo da *noise vector*. La loro idea fu estesa da Hu et al. per generare *adversarial sample* contro malware detector di Android (*black box attack*). Questa metodologia raggiunge il 100% di *success rate*, senza però considerare una soglia massima di modifiche applicabili al codice del malware.

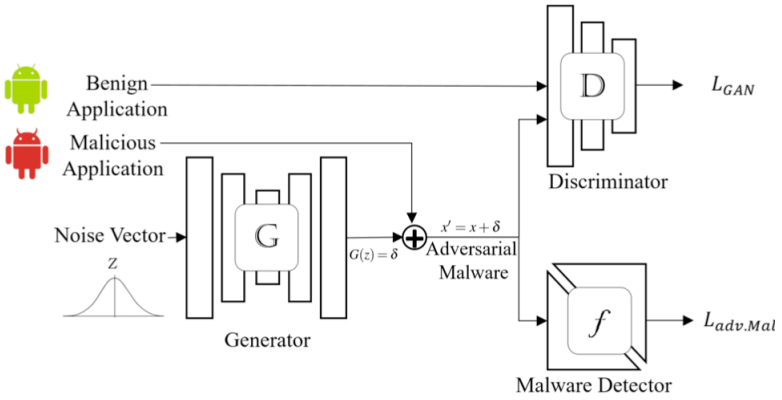
Shahpasand et al.[8] migliorano l'idea di Goodfellow et al., introducendo una soglia (*threshold*) per il livello massimo di distorsione applicabile al codice del malware originale.

4.3.1 Metodo di Shahpasand et al.

Il metodo proposto da Shahpasand et al.[8] si basa sulla tecnica dell'*evasion* per cercare di eludere il Classificatore e indurlo a catalogare Malware come file benigni. Il loro obbiettivo è quindi quello di generare *adversarial sample*, partendo dal codice originale di un malware, per ingannare il Malware Detector. In modo più formale:

Sia $X = \{0,1\}^m$ il *feature space*, con m il numero delle *feature*, dove $x \in X$ indica la presenza dell'*i-esima feature* in una parte di codice. Il classificatore deduce $f : X \to Y$ dal dominio X alle etichette *benigno* o *maligno* $y \in \{0,1\}$. Dato un malware x con la sua *true lable* $f(x) = 1$, l'attaccante punta a generare un *adversarial sample* $x' = x + \delta$ che risulti avere come etichetta $f(x') = 0$.

Utilizzano quindi una *GAN* per cercare la perturbazione δ ottimale da aggiungere al malware. L'architettura generale viene illustrata nella seguente figura:



Il *Generator* G prende in input un *noise vector* z e genera la perturbazione ottima $G(z) = \delta$. Il *Discriminator* usa come input un *sample Benigno* e il *sample* del malware distorto per poter diversificare il sample benigno dall'*adversarial malware*. Il *Discriminator* migliora la perturbazione aumentando la *loss* del *Generator* se l'*adversarial malware* è distinguibile dal *sample benigno*.
Questi due componenti sono *feed-forward neural network* che vengono allenate in maniera alternata in modo da ottimizzare la *min-max loss function*. Utilizzano una loro *loss function* formata da due elementi principali:

- **L_{GAN} function**[8], definita come:

  $$L_{GAN} = {}_{x\sim P_{Benign}} log D(x) + {}_{x\sim P_{Malware},\ z\sim P_z(z)} log(1-D(x+G(z)))\ \ s.t. |G(z)| < c$$

  dove $D(x)$ è la probabilità che il *sample* x derivi da un software benigno, L_{GAN} indica la necessità di avere un *Adversarial Malware* il più simile possibile ad uno *sample* benigno, avendo una distorsione che non superi una certa soglia c. $|G(z)|$ è il numero di *feature* aggiunte dalla perturbazione, se questo è troppo elevato rende immediatamente evidente l'attacco e potrebbe essere molto dispendioso applicare le modifiche al codice (in termini di costi e tempo).

- **$L_{adv.Mal}$ function**[8], definita come:

  $$L_{adv.Mal} = {}_{x,z} l_f(x + G(z), 0)$$

  dove la classe *Benigna*, con etichetta 0, è scelta come bersaglio per un *adversarial sample* $x + G(z)$ contro il Malware Detector f allenato con la *loss function* l_f.

La funzione utilizzata da Shahpasand et al. è quindi:

$$L = \alpha L_{GAN} + (1 - \alpha) L_{adv.Mal}$$

dove il primo termine indica una funzione che forza la GAN a generare un *adversarial sample* simile ad un *benign sample*; il secondo è un *adversarial malware sample* con un alto *misclassification rate*.

Utilizzano il dataset *Drebin Android* che contiene 129.013 applicazioni, in formato *APK*, di cui 5.560 sono malware. Ogni applicazione Android include il *Manifest* ed il *dexcode*, che sono le principali risorse da cui vengono estratte le *feature*. Queste vengono catalogate in 8 *feature set* che saranno poi utilizzate per la classificazione.

manifest	*dexcode*
Hardware components	Restricted API calls
Requested permissons	Used permission
Application components	Suspicious API calls
Filtered intents	Network addresses

Table 2: Feature Set del dataset Drebin Android.

Hanno allenato la *GAN* contro le seguenti tipologie di classificatori (tutti allenati con le impostazioni di default proposte dalla libreria "scikit-learn" di Python: 80% del dataset come *training set*):

- Support Vector Machine (SVM)
- Neural Networks (NN)
- Random Forest (RF)
- Logic Regression (LR)

Il risultato è stato che quasi tutti i classificatori sono molto vulnerabili a questo tipo di attacco: quasi il 99% di *success rate* contro le Neural Network e Logic Regression e fino all'87% contro le Support Vector Machine. Le Random Forest sono state le uniche più resistenti, con un *success rate* non superiore al 52%.

4.3.2 Metodo di Xiao Chen et al.

Xiao Chen et al.[1] propongono una metodologia di attacco *black-box*, basata su *CW* e *JSMA*, contro due diverse tipologie di Malware Detector che rappresentano lo *state-of-the-art* nel campo: *MaMaDroid*, basato sulle *Random Forest* e Drebin, basato su *Support Vector Machine*. Il meccanismo su cui si basa il loro attacco consiste nel trovare la perturbazione ottima da applicare all'archivio *APK* per portare i classificatori ad etichettare file maligni come benigni. Propongono anche un *framework* che, in automatico senza alcun intervento umano, è in grado di decomprimere l'*APK*, aggiungere le modifiche ai file *.dex* e ricompattare il tutto formando un nuovo *APK*. Dai test che hanno effettuato è possibile notare come i loro *Adversarial Sample* riducono la precisione dal 96% al 1% per MaMaDroid[1] e dal 97% al 1% per Drebin[1].

Attacco a MaMaDroid
La creazione di *Adversarial Sample* contro MaMaDroid avviene con l'aggiunta, il meno invasiva possibile, di chiamate API al codice *smali*. Il codice *smali* è il risultato di un operazione effettuata su un file *.dex*, un file binario, per renderlo più facilmente interpretabile e modificabile.

```
1    int x = 42;          // Java
2    13 00 2A 00          // .dex
3    const/16 v0, 42      // smali
```

Listing 6: Esempio di formati dex e smali. La riga 1 rappresenta il codice originale scritto in Java. La riga 2 è il contenuto dello stesso codice compilato in formato dex. La riga 3 è il codice smali ottenuto dalla conversione del file dex.

Utilizzano delle versioni appositamente modificate degli algoritmi *CW* e *JSMA* per calcolare il numero di perturbazioni (chiamate a funzioni) da aggiungere al codice. Il tutto si basa su come MaMaDroid calcola il *feature space* dell'app che deve analizzare. Aggiungendo un determinato numero di chiamate a API che partono da una specifica funzione chiamata *caller* ad un'altra chiamata *callee*, riescono ad alterare il valore del *feature space* inducendo il classificatore in una predizione errata. Propongono due versioni di questa procedura, una chiamata *Semplice* e l'altra *Sofisticata*. Di seguito una breve descrizione della *Semplice*.

La strategia *Semplice* inganna MaMaDroid aggiungendo specifiche classi nell'archivio *APK* e invocandone i metodi all'interno di *onCreate()* nella

MainActivity, per fargli estrarre un errato *feature vector*. Questo viene calcolato prendendo tutte le chiamate API dal file *classes.dex* e le converte nelle relative famiglie o pacchetti. Per esempio, una classe *MyClass* (da noi definita) contenuta all'interno di un pacchetto *android.os.mypack* (da noi definito) e la classe di sistema *StorageManager* presente all'interno del pacchetto di sistema *android.os.storage*, verranno ambedue convertite come famiglia *android* o come pacchetto *android.os*. Xiao Chen et al. sfruttano questa debolezza per indurre MaMaDroid ad effettuare classificazioni errate. È bene notare che l'aggiunta di queste chiamate non comporta alcuna modifica alle funzionalità originali del malware ed è possibile aggiungerne un numero arbitrario tramite l'utilizzo di un semplice script.

```
1    package android.os.mypack
2
3    public class Myclass {
4        public static void callee() {}
5        public static void caller() {
6              callee();
7              callee();
8        }
9    }
```

Listing 7: Esempio di classe ideata da Xiao Chen et al.[1] per sfruttare la debolezza del calcolo del *feature vector* di MaMaDroid

```
1    .class public Landroid/os/mypack/Myclass;
2    .source "Myclass.java"
3
4    .method public static callee()V
5        .locals 0
6        return-void .end method
7
8    .method public static caller()V
9        .locals 0
10       .line 6 invoke-static {},
11           Landroid/os/mypack/Myclass;->callee()V
12       return-void
13   .end method
```

Listing 8: Equivalente del codice di *Listing 7* in smail

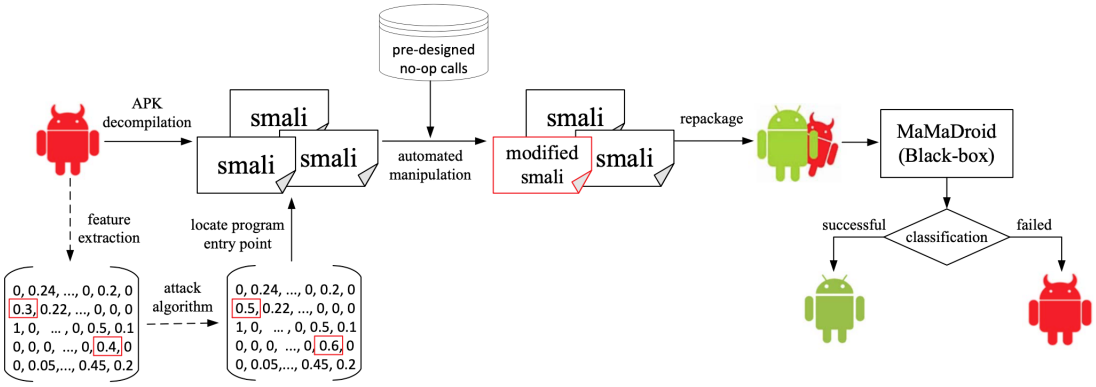


Figure 4: Xaio Chen et al. vs MaMaDroid
Questo diagramma mostra il funzionamento del metodo di Xiao Chen et al. contro MaMadroid. Le linee tratteggiate indicano il processo di attacco, le altre linee indicano il processo di manipolazione del file APK

Per la fase di testing hanno utilizzato MaMaDroid con le impostazioni di default fornite dagli autori e come dataset lo stesso utilizzato per MaMaDroid: un set di 5.879 software benigni e 5.560 software maligni. Dagli esperimenti che hanno condotto si evince che questo attacco ha un *success rate* che va dal 56%[1] fino al 96%[1].

Attacco a Drebin
Anche per attaccare Drebin sfruttano il modo in cui calcola il proprio *feature vector*. Per ottenerlo, Drebin effettua una "scansione lineare" sui file *smali* e *AndroidManifest.xml* andando a cercare solo la presenza di stringhe particolari, come il nome di chiamate API, senza andare a controllare se effettivamente le chiamate vengono eseguite. La loro strategia è quindi di andare ad aggiungere il codice (perturbazione) ai file *smali*, sotto forma di metodi o funzioni, senza però inserire l'istruzione per l'invocazione o esecuzione.

```
1     .method private addSuspiciousApiFeature()V .locals
      1
2       const-string v0, "phone" .line 17
3       invoke-virtual {p0, v0},
4             La/test/com/myapp/MainActivity;->
5             getSystemService(Ljava/lang/String;)
6             Ljava/lang/Object;
7         move-result-object v0
8         check-cast v0,
```

```
9       Landroid/telephony/TelephonyManager; return-void
10      .end method
```

Listing 9: Esempio di perturbazione che verrà aggiunta nei file smali.

Per trovare la perturbazione ottima utilizzano l'attacco JSMA che, tramite la matrice Jacobiana, riesce ad individuare la *feature* più influente da modificare per ogni iterazione. Il processo si ferma quando l'*adversarial sample* creato viene *misclassificato* oppure non si raggiunge la soglia massima di modifiche permesse. La matrice Jacobiana viene calcolata come:

$$J_F(X) = [\frac{\partial F(x)}{\partial X}] = [\frac{\partial F_j(X)}{\partial x_i}]_{i \in 1...n,\ j \in 0,1}$$

dove X è il *feature vector binario* per *Drebin* ed i è il risultato della classificazione ($i = 1$ per indicare un malware).

Per condurre gli esperimenti hanno utilizzato lo stesso dataset dell'attacco a MaMaDroid e dai risultati si può notare come questa metodologia raggiunge un *success rate* fino al 99%[1].

4.3.3 Metodo di Xiaolei Liu et al.

Xiaolei et al.[5] propongono una metodologia di attacco *black-box* con utilizzo di *Algoritmi Genetici* contro Malware Detection per sistemi *IoT* basati su Android, chiamato *TLAMD*.

Il loro obbiettivo è sempre quello di portare un Malware Detector a classificare un file maligno come benigno. L'approccio che utilizzano è quello di andare ad inserire *permission feature*[5] solo all'interno del file *AndroidManifest.xml* per evitare di alterare il funzionamento del codice del malware stesso e rendere più semplice applicare la modifica. Per aggiungere una sicurezza in più nel non alterare il corretto funzionamento del software andranno solo ad aggiungere elementi e mai a toglierli. Il numero di *feature* aggiunte rimane comunque importante: minore sarà la perturbazione meglio è, ovvero, minori saranno le modifiche al *file manifest* meglio è. Per far ciò applicano un algoritmo genetico che gli permette di trovare il minor numero di perturbazioni δ da applicare.

```
1   Require: Popluation Size pop_size
2       δ ← initialization()
3       for i=0 → pop_size do
4           Pi ← Crossover_Operator()
```

```
5               Pi ← Mutation_Operator()
6               Compute →  S(δ)
7               if F(X + δ) > 1 − F(X + δ) then
8                   Continue
9               else
10                  Output → δ
11              end if
12          end for
```

Listing 10: Pseudocodice per la generazione di un *Adversarial Sample*[5]

L'algoritmo per la creazione degli *Adversarial Sample* può essere riassunto nei seguenti punti[5]:

1. **Generazione casuale** della popolazione δ composta dalle varie *permission feature*.
2. **Individuazione** della *fitness function* che servirà per trovare una soluzione ottima per indurre il classificatore ad una predizione errata.
3. **Esecuzione** delle operazioni di *Mutazione*
4. **Generazione** di una **nuova popolazione** partendo dalle operazioni di mutazione del punto precedente e ritornare al punto 2. Se il numero prestabilito di iterazioni viene raggiunto, l'algoritmo termina.

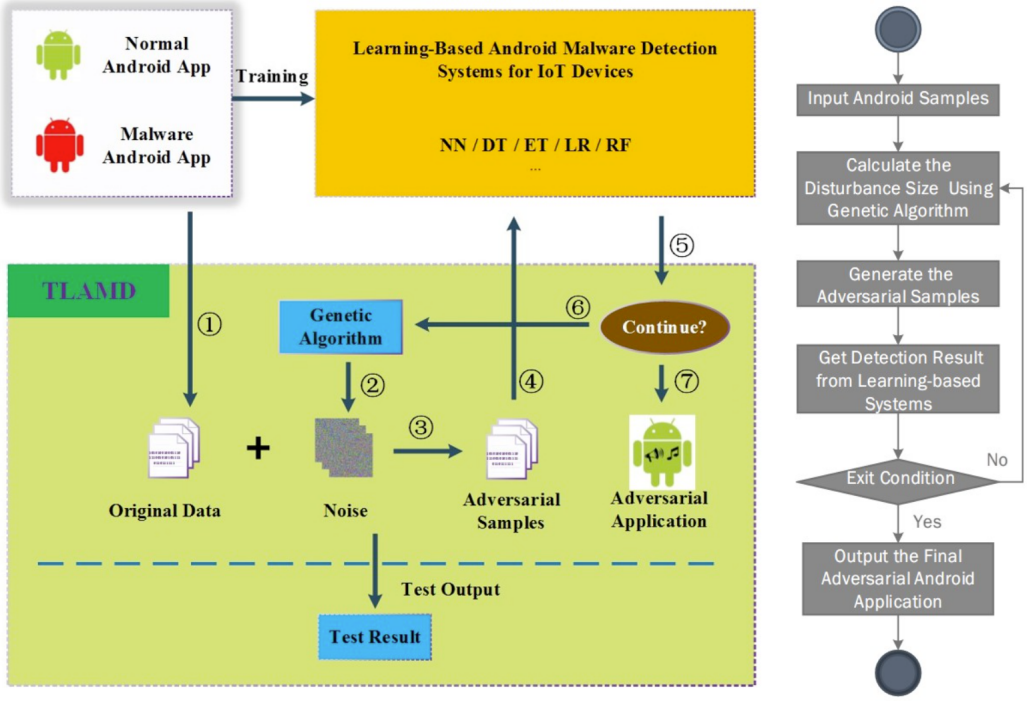


Figure 5: Funzionamento di TLAMD

Questo schema illustra il funzionamento dell'algoritmo per la generazione di Adversarial Sample[5]. ① Codice originale del Malware come input. ② Calcolo della perturbazione. ③ Generazione dell'*Adversarial Sample*. ④ Valore di ritorno del classificatore. ⑤ Controlla se la condizione di uscita è verificata. ⑥ Se non lo è, viene calcolata nuovamente la perturbazione con l'algoritmo genetico. ⑦ Se si, viene restituito in output il nuovo Malware.

Per verificare l'efficacia del loro attacco, hanno allenato 5 diversi modelli di classificatori tra cui *Logic Regression* (LR), *Decision Tree* (DT) e *Fully Connected Neural Networks* (NN), con lo stesso dataset utilizzato per *Drebin*. Solo quando questi raggiungeranno un determinato valore di *Accuracy* potranno essere attaccati.

Modello	**Accuracy**
NN (Neural Network)	99.83%
LR (Logic Regression)	99.45%
DT (Decision Tree)	99.86%
RF (Random Forest)	99.92%
ET (Extreme Tree)	99.96%

Table 3: La tabella mostra sulla sinistra i vari tipi di modelli utilizzati e sulla destra i valori la loro *Accuracy* [5]

Hanno generato Adversarial Sample per ogni tipologia di modello ed hanno verificato che il loro metodo risulta essere molto efficace: il *success rate* è sempre sopra l'80%[5] e molto spesso si avvicina al 100%[5]. Le *Neural Network* (NN) risultano estremamente vulnerabili (con un *success rate* del 100%, invece le *Extreme Tree* (ET) sono leggermente più resistenti delle altre (con un *success rate* di circa l'83%).

5 Conclusioni

In questo documento abbiamo introdotto il Sistema Operativo *Android* analizzandone l'architettura con una breve analisi dei vari livelli di cui è formata con un approfondimento sulle JVM utilizzate, la storia, le sue componenti principali, le applicazioni e la loro struttura. È stato introdotto il concetto di *Malware* con un elenco delle varie tipologie di software malevoli esistenti, alcuni accenni di storia ed alcune tecniche per generarli e per la loro individuazione e neutralizzazione. Abbiamo visto alcuni concetti base dell'*Adversarial Machine Learning* con i principali metodi di attacco a modelli di Classificazione per poi arrivare ad applicazioni pratiche contro *Malware Detection System* presenti in letteratura. Da ciò possiamo evincere che i sistemi di Malware Detection basati su intelligenza artificiali sono sì molto potenti e sicuramente più capaci dei tradizionali strumenti per la rilevazione di codici maligni, ma risultano anche vulnerabili ed estremamente insicuri contro *Adversarial Sample* studiati ed appositamente generati per ingannarli.

References

[1] Xiao Chen et al. *Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection*. 2018. arXiv: 1808.04218 [cs.CR].

[2] Przemyslaw Gilski and Jacek Stefanski. "Android os: a review". In: *Tem Journal* 4.1 (2015), p. 116.

[3] Google. *Project overview*. URL: https://developer.android.com/studio/projects.

[4] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. *Adversarial Machine Learning at Scale*. 2017. arXiv: 1611.01236 [cs.CV].

[5] Xiaolei Liu et al. "Adversarial Samples on Android Malware Detection Systems for IoT Systems". In: *Sensors* 19.4 (2019). ISSN: 1424-8220. DOI: 10.3390/s19040974. URL: https://www.mdpi.com/1424-8220/19/4/974.

[6] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub. "A survey on malware and malware detection systems". In: *International Journal of Computer Applications* 67.16 (2013).

[7] Alberto García Serrano. *Google se hace móvil. Android*. URL: https://media.cylex.mx/companies/1116/2352/uploadedfiles/11162352_634836870625603176_AndroidLM49_2012.pdf.

[8] Maryam Shahpasand et al. "Adversarial Attacks on Mobile Malware Detection". In: *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. 2019, pp. 17–20. DOI: 10.1109/AI4Mobile.2019.8672711.

[9] Wikipedia. *Android (operating system)*. URL: https://en.wikipedia.org/wiki/Android_(operating_system).