UNIVERSITÀ DEGLI STUDI DI GENOVA
Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

CAPDroid: Combining Analysis Processes for Android

Relatore:
Prof. Alessio Merlo
Correlatore:
Dott. Luca Verderame

Candidati:
Sara Martino
Luca Perazzo

Anno Accademico 2017 – 2018

Indice

1 Introduzione

Android è un sistema operativo per dispositivi mobili sviluppato da Google. Dalla sua introduzione sul mercato nel 2008, grazie anche al supporto di svariate aziende produttrici di smartphone come Sony, Samsung e HTC, che progettano telefoni specificatamente per sistemi Android, ha guadagnato una fetta sempre più considerevole di mercato, diventando il sistema operativo più diffuso per smartphone dal 2010 ad oggi. Nel primo trimestre del 2018 ha fatto registrare oltre l'85% delle vendite di smartphone, superando i 329 milioni di dispositivi venduti [1]. La popolarità crescente di questo sistema operativo ha avuto un impatto diretto sul Google Play Store, il quale è diventato il più grande store di applicazioni al mondo con più di 3 milioni di app disponibili per il download a Marzo 2018 [2]. Essendo le applicazioni un potenziale vettore di attacco ai danni degli utenti, risulta chiara la necessità di inserire dei controlli per assicurarne la sicurezza: oltre ad applicare diverse politiche di restrizione alle tipologie di app che possono essere caricate sullo store, l'azienda di Mountain View ha introdotto nel 2012 Google Bouncer, un sistema per scansionare automaticamente le applicazioni prima che vengano rese disponibili agli utenti; tale sistema, tra le altre attività, si serve di un'analisi statica alla ricerca di minacce note, esegue il software in un emulatore virtuale per identificarne il comportamento e analizza i dati che l'applicazione inserisce nel sistema alla ricerca di eventuali anomalie. Sebbene l'esatto funzionamento del Bouncer non sia stato reso noto per evitare di essere aggirato, poco dopo la sua introduzione, alcuni ricercatori sono riusciti a dissezionarlo completamente e nel giro di pochi mesi sono state ideate tecniche atte a bypassare tale meccanismo di sicurezza (maggiori informazioni a ri-

Worldwide Smartphone Sales to End Users by Operating System in 1Q18 (Thousands of Units)

Operating System	1Q18 Units	1Q18 Market Share (%)	1Q17 Units	1Q17 Market Share (%)
Android	329,313.9	85.9	325,900.9	86.1
iOS	54,058.9	14.1	51,992.5	13.7
Other OS	131.1	0.0	607.3	0.2
Total	**383,503.9**	**100.0**	**378,500.6**	**100.0**

Source: Gartner (May 2018)

Figura 1: Vendite globali di smartphone per sistema operativo (migliaia di unità)

guardo possono essere trovate in [3] e [4]); il fatto di poter eludere il Bouncer in maniera non eccessivamente complessa, a causa dell'impiego di controlli non adatti ad un'analisi approfondita ed esaustiva, ha permesso che numerosi malware continuassero ad essere resi disponibili al download (un esempio è il malware Judy che ha raggiunto tra i 4 e i 18 milioni di download prima di essere scoperto da CheckPoint e conseguentemente rimosso da Google[1]). A partire dal rilascio della versione 8 di Android nel 2017, è stato introdotto un meccanismo aggiuntivo real-time, Google Play Protect, che scansiona costantemente le app presenti sul dispositivo alla ricerca di comportamenti malevoli. Sebbene questo nuovo meccanismo abbia migliorato la situazione e sia in continua evoluzione, spesso tale sistema di sicurezza fallisce nel riconoscere i malware prima che essi infettino un grosso numero di device (come ad esempio nel caso di ExpensiveWall, un malware che ha infettato almeno 50 applicazioni dello store e che ha raggiunto un numero di download compreso tra 1 e 4 milioni prima di essere rimosso sotto indicazione di CheckPoint[2]) e a Luglio 2018 AV-TEST, lo ha classificato come il peggiore tra 21 prodotti di sicurezza Android analizzati[3], sulla base dei risultati ottenuti in seguito alla scansione di un elevato numero di malware[4].

Il generale interesse crescente nella sicurezza in ambito software, la maggiore diffusione di dispositivi mobili e relative applicazioni, il numero sempre crescente

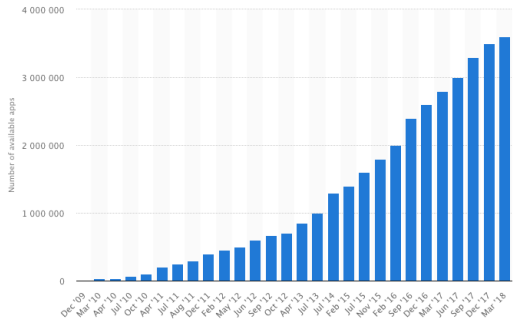

Figura 2: Numero di applicazioni disponibili sul Google Play Store

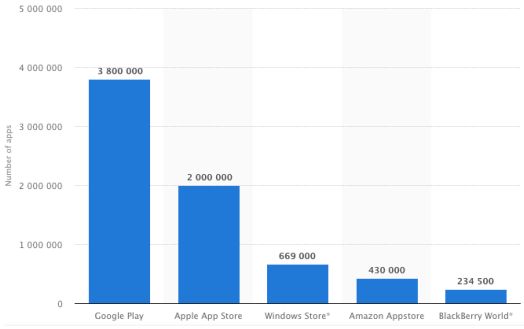

Figura 3: Numero di applicazioni presenti nei piú importanti Store

[1] https://blog.checkpoint.com/2017/05/25/judy-malware-possibly-largest-malware-campaign-found-google-play/

[2] https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/

[3] https://www.av-test.org/en/antivirus/mobile-devices/android/july-2018/

[4] https://www.av-test.org/en/antivirus/mobile-devices/android/july-2018/google-play-protect-10.6-182711/

di malware finalizzati a colpire tali piattaforme (come evidenziato nei report di sicurezza [5] e [6]) e la presenza di una percentuale significativa di questi persino negli store ufficiali [7], sono tutti fattori determinanti che hanno portato, negli ultimi anni, un numero sempre crescente di sviluppatori a dedicarsi alla creazione di strumenti che potessero condurre analisi automatiche migliori e più efficienti.

I tipi di analisi che possono essere eseguiti su un software sono essenzialmente l'analisi statica e quella dinamica: l'analisi statica consiste nell'analizzare il codice sorgente e/o i compilati al di fuori del contesto esecutivo cercando di individuare tutti i possibili comportamenti del programma alla ricerca di errori, *backdoor* e codice potenzialmente dannoso; l'analisi dinamica consiste al contrario nell'eseguire effettivamente il programma, solitamente in un ambiente controllato, per monitorarne il reale comportamento. Entrambi i metodi presentano caratteristiche positive e negative e sono solitamente utilizzati in modo indipendente l'uno dall'altro, anche a causa della loro natura opposta, ottenendo così risultati diversi e non necessariamente compatibili: è infatti facile incorrere nella presenza di falsi positivi in seguito ad un'analisi di tipo statico, in quanto questa non tiene conto della reale esecuzione del codice e quindi della presenza di eventuale *dead code*; d'altro canto l'utilizzo dell'analisi dinamica è limitato dall'impossibilità di coprire completamente il dominio di esecuzione e pertanto si possono ottenere numerosi falsi negativi. La scelta ideale è sicuramente quella di utilizzare entrambi i metodi: una possibilità è quella di studiare e confrontare i risultati delle due analisi con il fine di eliminare falsi positivi e negativi; ciò che però condurrebbe ad un miglior prodotto finale, sarebbe sicuramente riuscire a combinare i due metodi in modo da trovare un vero e proprio collegamento tra i due ed evitando di utilizzarli in modo indipendente.

L'obiettivo principale di questa tesi è stato quello di ideare un approccio innovativo che fosse in grado di combinare i vantaggi delle tecniche di analisi statica e dinamica.

Per raggiungere questo scopo, si è proceduto alla revisione sistematica dello stato dell'arte delle tecniche e degli strumenti di analisi di applicazioni; questa iniziale fase di ricerca è stata fondamentale per studiare le limitazioni e i possibili punti di miglioramento dei metodi di analisi attualmente esistenti.

Successivamente sono stati ideati e progettati una metodologia e un framework di riferimento per l'analisi automatizzata di software generici.

Infine, per mostrare la validità dell'approccio, è stato sviluppato un prototipo per il sistema operativo Android, testato inizialmente su alcune applicazioni costruite ad hoc in modo da verificarne il corretto funzionamento su casi d'uso noti; è stata inoltre compiuta una campagna di testing su 600 applicazioni ufficiali scaricate dal Google Play Store, in modo da ottenere dei risultati sperimentali.

1.1 Struttura della tesi

Questa tesi è strutturata come segue: nella parte introduttiva è stato introdotto il lavoro oggetto dell'elaborato, rendendo evidenti le motivazioni che hanno condotto a tale scelta.

Nella seconda sezione vengono presentati e descritti brevemente gli strumenti che sono stati utilizzati al fine di poter modellare nel modo migliore ogni fase del processo di analisi, illustrandone il funzionamento.

La terza sezione si concentra invece su Android, sistema operativo di riferimento scelto per lo sviluppo del prototipo, e in particolare sulla struttura interna delle applicazioni, in modo da poter dare un'idea più precisa del contesto e delle problematiche di maggior interesse per la questione trattata.

La quarta sezione si focalizza sulla parte di ricerca compiuta, elencando e descrivendo le varie tipologie e tecniche di analisi presenti allo stato dell'arte; in questa sezione sono spiegate quali siano le motivazioni che hanno condotto alla scelta di determinati tipi di approccio rispetto ad altri.

La quinta e la sesta sezione illustrano nel dettaglio la metodologia ideata, il relativo framework di riferimento e la possibile implementazione dello stesso, applicato al mondo delle applicazioni Android.

Nella settima sezione vengono mostrati i risultati sperimentali ottenuti e le performance del sistema in seguito alla fase di testing del prototipo su applicazioni prese dal Play Store.

Nell'ottava ed ultima sezione sono esposte delle considerazioni in merito al lavoro svolto, illustrando inoltre quali potrebbero essere dei miglioramenti da apportare in futuro sul sistema in modo tale da ampliare i casi di applicazione.

2 Strumenti Utilizzati

In questo capitolo vengono introdotti gli strumenti e le tecnologie necessarie allo studio e allo sviluppo prototipale. Poiché il processo di analisi ideato è il prodotto dell'unione di più fasi, il sistema finale risulta composto da più moduli, ognuno di essi trattato in maniera separata: per lo sviluppo di ogni modulo, sono stati scelti gli strumenti e i linguaggi ritenuti più opportuni e funzionali.

2.1 Android Software Development Kit (SDK)

Il Software Development Kit di Android è stato necessario per la costruzione di applicazioni di test (attraverso l'IDE Android Studio fornito da Android) e per l'utilizzo delle librerie di Android, degli emulatori forniti di default e dell'Android Debug Bridge (adb). Android consente la possibilità di scaricare l'IDE Android Studio direttamente dal sito ufficiale[5]; attraverso l'installazione di questo, verrà automaticamente scaricato anche l'Android SDK Manager, il quale permette di ottenere gli SDK desiderati.

2.1.1 Librerie

Le librerie di Android si possono dividere in due grandi categorie, le *Support Library* e le *Data Binding Library*: le prime permettono lo sviluppo di applicazioni che supportano una compatibilità anche con le versioni precedenti di API, mentre le seconde consentono di gestire il collegamento tra gli elementi di interfaccia e i dati presenti nell'app. Lo studio e l'utilizzo delle API presenti in queste librerie è stato fondamentale al fine dell'analisi di vulnerabilità.

2.1.2 Emulatore

L'SDK include un AVD Manager che permette la creazione e l'utilizzo di dispositivi Android virtuali sul proprio computer. Su tali device si possono installare applicazioni per poterne testare il funzionamento, interagendo con l'app stessa "toccando" lo schermo o utilizzando i tasti fisici di cui il dispositivo che si sta usando è fornito. Uno degli emulatori di default (Nexus 5, architettura x86, supporto fino alle API

[5] `http://developer.android.com/sdk/index.html`

21) viene utilizzato per emulare l'esecuzione delle applicazioni nella fase dinamica del sistema.

2.1.3 Android Debug Bridge (adb)

Android Debug Bridge è uno strumento da linea di comando incluso nel pacchetto *Platform-Tools* dell'SDK che permette di comunicare con un qualunque device che abbia come sistema operativo Android. È un programma client-server che include tre componenti:

- Un client che invia comandi; è eseguito sul proprio computer e può essere invocato da terminale attraverso un comando adb.
- Un demone (adbd) che fa eseguire i comandi sul device; è un processo in background sul dispositivo.
- Un server che gestisce la comunicazione tra client e demone; il server viene eseguito come processo in background sul proprio computer.

Quando il client viene lanciato, questo controlla se esiste già un server adb in esecuzione e, in caso negativo, avvia il processo del server. Quando il server si avvia apre la porta TCP locale 5037 e ascolta i comandi inviati dall'adb client (tutti i client adb usano la porta 5037 per comunicare con l'adb server). Il server a questo punto imposta le connessioni a tutti i dispositivi attivi e localizza gli emulatori scansionando le porte con numero dispari nel range 5555 - 5585, il range utilizzato dai primi 16 emulatori (ogni emulatore utilizza due porte sequenziali, una porta con numero pari per le connessioni alla console e una con numero dispari per la connessione con adb). Quando il server trova un adbd imposta una connessione a quella porta. Una volta che il server ha impostato le connessioni a tutti i dispositivi, possono essere usati i comandi adb per accedere a quei device.

2.2 ApkTool & Dare

ApkTool[6] e Dare[7] sono due tool che vengono utilizzati nella primissima fase di analisi. Il loro scopo è quello di disassemblare le applicazioni Android per mezzo

[6] `https://ibotpeaches.github.io/Apktool/`
[7] `http://siis.cse.psu.edu/dare/`

della *reverse engineering*. Data un'app in formato APK, riescono a decodificare le risorse fino ad ottenere quasi la loro forma originale, e sono in grado di ricostruirle applicando alcune modifiche in modo tale da rendere più agevole il processo di analisi del codice.

2.3 Wala

Le T.J. Watson Libraries for Analysis (WALA) sono delle librerie che forniscono strumenti per l'analisi statica del Java bytecode e linguaggi simili, e per Javascript. Inizialmente l'infrastruttura WALA era sviluppata indipendentemente come parte del progetto di ricerca DOMO all'IBM T.J. Watson Research Center. Nel 2006 IBM ha donato il software alla comunità, rendendolo a tutti gli effetti un progetto Open Source, consultabile e scaricabile attraverso una repository su GitHub[8]. Wala fa uso durante la propria analisi dell'*Intermediate Representation* (IR, struttura dati o codice, usata internamente da un compilatore o da una macchina virtuale per rappresentare il codice sorgente), la quale possiede una proprietà chiamata *Static Single Assignment (SSA)*, che richiede che ogni variabile sia assegnata esattamente una volta e che ogni variabile sia definita prima di essere utilizzata. Per una visione totale di tutte le caratteristiche che Wala può offrire, si rimanda al sito ufficiale[9].

2.4 Z3

Allo stato dell'arte Z3 è un *Satisfiability Modulo Theories (SMT)* solver, ovvero uno strumento utilizzato per la risoluzione di problemi decisionali che implicano formule logiche. Tale tool, proveniente dalla Microsoft Research, è spesso utilizzato nella *Constraint Programming*, poiché è anche in grado di risolvere CSPs (Constraint Satisfaction Problems) ovvero problemi matematici definiti come un insieme di oggetti il cui stato deve soddisfare un numero di vincoli o limitazioni. Tale strumento, ha performance molto alte ed è uno degli SMT solver più utilizzati reperibili sul web; nel 2015 ha vinto[10] l'ACM SIGPLAN Programming Languages Software Award[11], un prestigioso premio che riconosce l'impatto significativo e duraturo di un sistema software su ricerca e sviluppo nell'ambito dei linguaggi di programmazione. Z3

[8] `https://github.com/wala/WALA`
[9] `http://wala.sourceforge.net`
[10] `https://www.microsoft.com/en-us/research/blog/z3-wins-2015-acm-sigplan-award/`
[11] `http://www.sigplan.org/Awards/Software/`

può essere scaricato ed installato direttamente dalla sua repository di GitHub[12]; seguendo le guide si può eseguire il binding con molti linguaggi di programmazione; all'interno del progetto è stato utilizzato per risolvere vincoli attraverso uno script in Python.

2.5 AndroidViewClient

AndroidViewClient era stato originariamente concepito come un'estensione di *monkeyrunner*, ma successivamente si è evoluto come un puro Python tool che automatizza e semplifica la creazione di test script. È un test framework per applicazioni Android che:

- Automatizza e guida le app
- Genera script riutilizzabili
- Fornisce un'interazione UI basata sulle view, indipendente dal tipo di device
- Supporta l'utilizzo su dispositivi differenti
- Fornisce un semplice controllo per operazioni di alto livello
- Supporta tutte le API di Android

Per l'utilizzo di AndroidViewClient è necessario scaricare ed installare sul proprio computer il tool dalla repository di GitHub[13]. La fase di installazione comprende anche il download e installazione dei due componenti principali del tool, dump e culebra.

[12] `https://github.com/Z3Prover/z3`
[13] `https://github.com/dtmilano/AndroidViewClient`

3 Android

Le sezioni che seguono hanno lo scopo di descrivere l'architettura Android, la sua sicurezza, la struttura delle applicazioni, la comunicazione e il modo in cui il sistema operativo gestisce l'interazione utente, in modo da fornire una conoscenza più approfondita del contesto di lavoro.

3.1 Architettura della piattaforma

Android è un sistema operativo open-source basato su kernel Linux, creato per una vasta gamma di dispositivi. La piattaforma è strutturata secondo un'architettura a livelli come mostrato in Fig. 4.

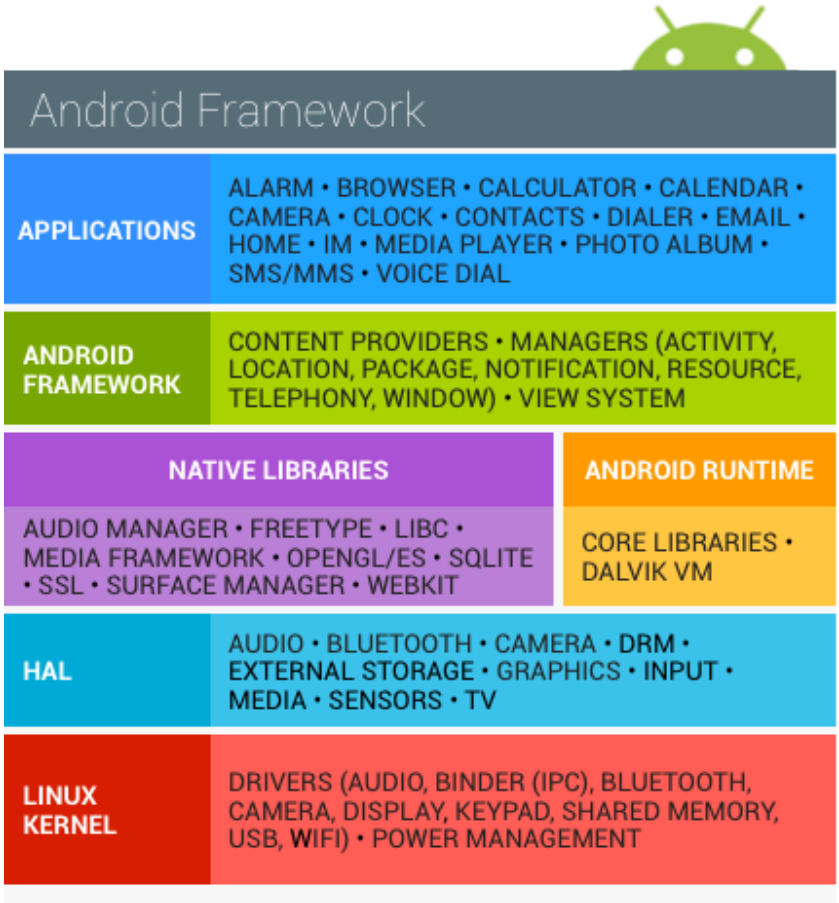

Figura 4: Stack di Android

3.1.1 Linux Kernel

Il livello più basso dell'architettura è il Linux Layer, che rende possibile l'astrazione dell'hardware e fornisce ai livelli superiori diversi driver e funzionalità elementari tra cui la gestione della memoria, dei consumi e dei processi. L'utilizzo di un kernel Linux offre non solo la possibilità di sviluppare driver hardware per un kernel conosciuto e testato, ma permette in generale di ereditarne le caratteristiche di affidabilità e sicurezza: la piattaforma Android sfrutta infatti la protezione Linux *user-based* assegnando ad ogni applicazione un identificativo utente univoco (UID), eseguendola in un processo separato e utilizzando gli UID per creare un'Application Sandbox a livello kernel; in questo modo le applicazioni possiedono un accesso limitato al sistema operativo e non possono interagire tra loro. Poiché l'Application Sandbox è inserito a livello kernel, il modello di sicurezza è esteso anche al codice nativo e alle applicazioni del sistema operativo stesso: in tal modo tutto il software sopra il kernel (come le librerie di sistema, l'Android framework, l'Application Runtime e tutte le altre applicazioni) viene eseguito all'interno della Sandbox. Il sistema così progettato fa in modo che non sia permesso ad un'applicazione pericolosa di poter danneggiare altre applicazioni, il sistema Android o il dispositivo stesso.

È da sottolineare la presenza di un driver dedicato alla gestione dell'Inter-Process Communication (IPC), fondamentale per permettere una comunicazione sicura e controllata tra componenti e applicazioni eseguiti su processi differenti. Una spiegazione dettagliata del meccanismo di IPC è rimandata alla Sezione 3.4.

3.1.2 Hardware Abstraction Layer (HAL)

L'Hardware Abstraction Layer fornisce un'interfaccia standard che mette a disposizione le funzionalità hardware del dispositivo al livello dell'Android Framework; consiste in diversi moduli di librerie, ognuno dei quali implementa un'interfaccia per uno specifico componente hardware, come ad esempio la fotocamera o il bluetooth. Quando una API di livello framework esegue una chiamata per accedere all'hardware del dispositivo, il sistema carica il modulo di librerie per quello specifico componente.

3.1.3 Android Runtime (ART)

ART è un ambiente di runtime che sostituisce Dalvik, la precedente *virtual machine* utilizzata da Android: per dispositivi con versione Android 5.0 o superiori, ogni

app viene eseguita all'interno del proprio processo con la propria istanza di Android Runtime. ART è ideato per supportare un forte parallelismo su device con poca memoria, eseguendo il codice in formato DEX (Dalvik Executable), uno speciale *bytecode* progettato specificatamente per Android, ottimizzato per un utilizzo minimo della memoria e generato a partire dai file Java che compongono l'applicazione.

3.1.4 Native Libraries

Questo layer contiene un insieme di librerie native scritte in linguaggio C e C++. Il sistema fornisce l'Android Framework per rendere disponibili le funzionalità di alcune di queste anche alle applicazioni: per esempio, si può accedere a OpenGL ES attraverso le Java OpenGL API, per aggiungere alla propria app un supporto alla manipolazione grafica in 2D e 3D.

3.1.5 Android Framework

L'Android Framework, o Java API Framework, mette a disposizione del livello applicativo l'insieme delle caratteristiche del sistema Android sotto forma di componenti e API in linguaggio Java, che formano i blocchi base per creare un'applicazione Android. Tra i vari blocchi vi sono:

- View System: è un insieme di View utilizzabili per creare l'interfaccia utente di un'app; include oggetti come bottoni, liste, caselle di testo e persino web browser incorporabili.
- Resource Manager: fornisce accesso alle risorse statiche come stringhe, immagini e file di layout.
- Notification Manager: fornisce a tutte le app la possibilità di mostrare avvisi personalizzati nella barra di stato.
- Activity Manager: gestisce il ciclo di vita delle applicazioni e mantiene lo stack di tutte le app aperte.
- Content Provider: permettono di accedere e/o condividere i dati tra le applicazioni, come ad esempio contatti o simili.

Gli sviluppatori hanno completo accesso al medesimo framework di API che viene utilizzato dal sistema Android stesso.

3.1.6 Applications Layer

Il livello applicativo è in cima allo stack architetturale. Android presenta un insieme di applicazioni base per email, messaggistica, calendario, browser internet, contatti e simili: queste applicazioni non possiedono uno stato speciale, pertanto l'utente può decidere di installare app di terze parti e impostarle come default (a parte alcune eccezioni, come ad esempio l'applicazione per gestire le impostazioni). Le app di sistema possono essere inoltre utilizzate per fornire funzionalità chiave che gli sviluppatori possono sfruttare attraverso la propria applicazione: ad esempio è possibile richiamare qualsiasi app di messaggistica già installata sul dispositivo per consegnare un messaggio, senza necessità di aggiungere questa funzionalità alla propria applicazione.

3.2 Android Security Mechanisms

Android prevede delle funzionalità di sicurezza integrate in modo da ridurre l'impatto delle problematiche legate a quest'ambito; alcune tra le più importanti caratteristiche sono elencate nel seguito:

- Android Application Sandbox: come descritto in Sezione 3.1.1, il meccanismo di Sandbox isola l'applicazione e i suoi dati e pone limitazioni riguardo l'insieme di risorse di sistema accessibili dall'app; tale accesso è gestito direttamente dal sistema. La comunicazione tra applicazioni è resa sicura grazie al meccanismo degli Intent che gestisce IPC e ICC (cfr. Sezione 3.4).

- Android Permission Model: l'utilizzo di API ritenute sensibili (ad esempio quelle riguardanti l'accesso alla rete, l'invio di SMS o l'utilizzo della camera, della localizzazione o del bluetooth) è regolato da un meccanismo basato sui permessi. Per fare uso delle API protette, un'applicazione deve necessariamente richiedere il permesso corrispondente al sistema: il metodo tradizionale prevede l'inserimento all'interno del Manifest di tutti i permessi necessari (cfr. Sezione 3.3.2); in fase di installazione dell'app, il sistema mostra all'utente i permessi richiesti, lasciando ad esso la decisione di concederli tutti o di rinunciare all'installazione. A partire dall'introduzione di Android 6.0 (API 23), è stata aggiunta la possibilità di richiedere permessi a runtime al momento dell'effettivo bisogno, mediante l'utilizzo della funzione requestPermissions();

in tal caso, il sistema mostra all'utente la richiesta solo al momento in cui essa viene effettivamente effettuata dall'app e non in fase di installazione. Il meccanismo dei permessi può inoltre essere utilizzato da un'applicazione per definire uno specifico permesso che un'altra app deve richiedere per utilizzarla.

- Lack of API: alcuni comportamenti di un'app sono vietati direttamente dalla mancanza delle API che permetterebbero interazioni non sicure con dati sensibili o con il sistema operativo. Ad esempio non sono previste API che permettano la manipolazione diretta della SIM card del device.

- Application Signing: ogni applicazione eseguita sulla piattaforma Android deve essere firmata dallo sviluppatore; il tentativo di installare app non firmate è rigettato dal Google Play Store o dal *package installer* del device stesso. Le applicazioni di sistema sono firmate mediante la chiave della piattaforma, mentre applicazioni di terze parti possono essere firmate direttamente dagli sviluppatori senza la necessità di una Central Authority.

- Encryption: il sistema Android supporta la cifratura dell'intero file system, che può essere attivata ad esempio per proteggere un device perso o rubato.

- Cryptography: Android fornisce un grande numero di algoritmi atti a proteggere i dati di un'applicazione mediante l'utilizzo della crittografia.

- Protection of Stored Data: Android permette tre modalità di salvataggio dei dati, ovvero tramite internal storage, external storage o content provider. Per quanto riguarda la memoria interna del dispositivo, il meccanismo di default di Android prevede che i dati siano accessibili alla sola app. In riferimento alla memoria esterna, questa è accessibile globalmente ed è pertanto sconsigliato il salvataggio di dati sensibili in essa. Infine, il meccanismo predisposto per la condivisione di dati tra applicazioni diverse è legato al'utilizzo dei content provider che saranno descritti in dettaglio in Sezione 3.3.1.

3.3 Struttura delle applicazioni

Le applicazioni Android possono essere scritte utilizzando i linguaggi Java, Kotlin e C++. Gli strumenti dell'Android SDK compilano il codice insieme ai dati e ai file di risorse formando un APK (Android Package), il quale è in sostanza un archivio

con estensione .apk; un APK include tutti i contenuti dell'applicazione, ed è il file utilizzato dai dispositivi Android per installare l'app.

3.3.1 App Components

I componenti sono gli elementi essenziali di ogni applicazione Android. Ciascun componente è un entry point attraverso il quale il sistema o un utente può effettuare accesso all'app. Esistono quattro tipi differenti di componente, ognuno dei quali con uno scopo diverso e un ciclo di vita proprio che ne definisce creazione, comportamento e distruzione.

- **Activity**: le activity sono i componenti essenziali delle applicazioni Android e rappresentano una singola schermata con interfaccia utente. Sebbene le activity appartenenti ad una stessa applicazione lavorino insieme per formare un'esperienza utente compatta, ognuna di esse è indipendente dalle altre. Il modo in cui esse sono gestite e collegate è uno dei concetti fondamentali caratterizzanti il modello di applicazione Android: a differenza degli altri linguaggi di programmazione, in cui le app sono eseguite a partire da un metodo *main()*, il sistema Android inizia l'esecuzione del codice in corrispondenza dell'istanza dell'activity definita come launcher (solitamente denominata *MainActivity*), invocando specifici metodi di callback in base al raggiungimento di specifici stati del suo ciclo di vita che variano durante il corso del suo utilizzo; queste callback (onCreate, onStart, onResume, onPause, onStop, onRestart, onDestroy), che prendono il nome di *lifecycle methods*, possono essere utilizzate da uno sviluppatore per gestire il comportamento dell'activity durante le transizioni da uno stato all'altro. Il lifecycle di un'activity è visibile in Fig. 5.

- **Service**: un servizio è un componente non provvisto di interfaccia utente che può svolgere operazioni in background come gestire transazioni sulla rete, riprodurre musica, svolgere I/O da file o interagire con un content provider. I servizi possono essere catalogati come *started* o *bound*, in base al modo in cui vengono avviati: uno *Started Service* si ottiene quando un altro componente chiama la funzione *startService()*; generalmente questo tipo di servizio non offre interazioni con il chiamante e continua ad essere eseguito in background anche se il componente che l'ha avviato viene terminato. Un *Bound Service* si ottiene invece quando uno o più componenti si legano ad esso chiamando la

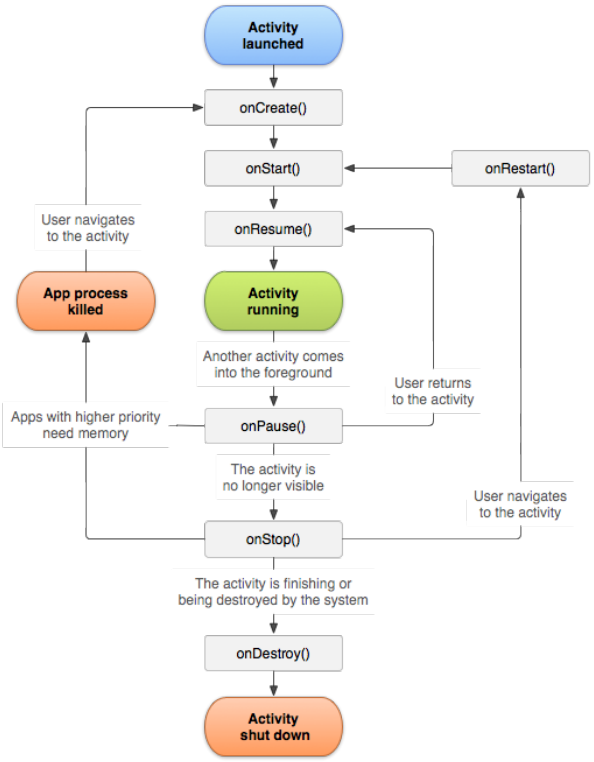

Figura 5: Activity Lifecycle

bindService() ed offre un'interfaccia client-server, che permette ai componenti di interagire con esso mandando richieste o ricevendo risultati sfruttando l'IPC; il servizio, in questo caso, rimane in esecuzione solamente finché almeno un componente rimane legato ad esso, altrimenti viene distrutto. Si può inoltre distinguere tra servizi in foreground e background: un *Foreground Service* svolge operazioni che sono visibili dall'utente (ad esempio riproduzione di musica) e deve necessariamente mostrare una notifica; un servizio di questo tipo continua la sua esecuzione anche se l'utente non interagisce con l'app. Un *Background Service*, svolge operazioni che non sono direttamente notate dall'utente (ad esempio compressione della memoria) e non necessita di una notifica; a partire dal livello API 26, il sistema impone restrizioni sull'esecuzione di questo tipo di servizi in background quando l'applicazione non sia in foreground.

- **Broadcast Receiver**: le applicazioni Android possono inviare o ricevere messaggi in broadcast dal sistema o da altre app, in maniera simile al design pattern del publish-subscribe. Questi messaggi sono inviati nel momento in cui

accade un evento di interesse: il sistema invia messaggi per eventi di sistema, ad esempio quando il dispositivo viene avviato o messo in carica; le app possono inviare messaggi personalizzati, ad esempio per notificare il completamento del download di nuovi dati. Le applicazioni possono registrarsi per ricevere determinati tipologie di messaggio: quando uno di questi è inviato, il sistema lo instrada automaticamente alle app che si sono sottoscritte per ricevere quel particolare tipo di messaggio. In generale, i messaggi di broadcast possono essere utilizzati come comunicazione di sistema tra le app, al di fuori del normale flusso utente.

- **Content Provider**: il content provider può essere utilizzato da un'applicazione per gestire l'accesso ai propri dati o a quelli di altre applicazioni, garantendo quindi la possibilità di condividere dati tra le app; essi incapsulano i dati fornendo un meccanismo per tenerli al sicuro e sono l'interfaccia standard che collega i dati di un processo al codice che viene eseguito all'interno di un altro. L'utilizzo di un content provider consente di configurare un controllo granulare sui permessi di accesso ai dati, permettendo di scegliere ad esempio di restringere l'accesso a un provider o configurare permessi distinti per la lettura e la scrittura del singolo dato. Un content provider permette l'accesso a svariate sorgenti di dati, sia record strutturati di database relazionali SQLite, sia dati non strutturati come immagini o file audio/video.

Un altro elemento fondamentale di Android, che non può essere considerato un componente a tutti gli effetti, in quanto non indipendente, è il **Fragment**. Questo può essere immaginato come una sezione modulare di un'activity ed è utilizzato per rappresentare una porzione specifica dell'interfaccia utente: possiede il proprio ciclo di vita, riceve i propri eventi di input e può essere aggiunto o rimosso mentre l'activity è in esecuzione; poiché ogni fragment deve sempre essere ospitato da un'activity, il suo ciclo di vita è direttamente legato a quello dell'activity di appartenenza. Fragment multipli possono essere combinati in una singola activity per costruire un'interfaccia su più riquadri, ed ognuno di essi può essere riutilizzato in activity differenti. Quando il fragment viene aggiunto come parte del layout dell'activity, vive in una *ViewGroup* interna alla gerarchia delle View dell'activity, mantenendo un proprio layout ben definito: può essere inserito come elemento al-

l'interno del layout file dell'activity, oppure programmaticamente aggiungendolo ad una ViewGroup esistente.

3.3.2 File Manifest

Ogni progetto di applicazione deve obbligatoriamente dichiarare un file *AndroidManifest.xml* con questo preciso nome, all'interno del pacchetto applicativo: questo file descrive le informazioni necessarie agli Android build tools, al sistema operativo e al Google Play Store. Nel manifest è necessario dichiarare i seguenti elementi:

- Package Name: solitamente coincide con il namespace dello sviluppatore; viene usato per determinare la posizione delle entità del codice durante il build del progetto. Quando l'app viene impacchettata, questo valore viene sostituito con l'ID dell'applicazione ottenuto dal file di Gradle e viene usato come identificativo univoco dell'app sul sistema e sul Google Play Store.

- App Components: includono activity, service, broadcast receiver e content provider. Ogni componente deve definire proprietà di base, come il nome della propria classe (Java o Kotlin). Possono essere anche dichiarati attributi opzionali, come il tag *exported* che determina se il componente può essere chiamato dall'esterno dell'applicazione o l'intent filter che descrive come il componente può essere attivato. Per quanto riguarda quest'ultimo, la sua dichiarazione è obbligatoria per l'activity launcher dell'applicazione, in modo che il sistema possa riconoscere il principale entry point dell'app. Activity, service e content provider che siano inclusi nel codice, ma non dichiarati nel manifest, non sono visibili al sistema, e dunque non potranno mai essere eseguiti. Il discorso è diverso per i broadcast receiver, i quali possono essere creati anche dinamicamente da codice.

- Permissions: necessari all'app per accedere a parti protette del sistema o di altre applicazioni; possono essere inoltre dichiarati permessi che altre app devono avere se vogliono accedere al contenuto di questa.

- HW/SW requirements: caratteristiche hardware e software necessarie all'app, che definiscono i dispositivi su cui l'applicazione può essere installata attraverso il Google Play Store.

Per una descrizione più completa di cosa possa essere inserito in un manifest, si rimanda alla documentazione ufficiale di Android[14].

3.3.3 App Resources

Un'applicazione Android è composta non solo da codice, ma anche da risorse separate dal codice sorgente come immagini, file audio e tutto ciò che riguarda la presentazione visiva dell'app: si possono ad esempio definire animazioni, menu, stili, colori attraverso file XML. Particolare importanza assumono i file XML di layout, utilizzati per definire la struttura di tutti i componenti dell'app che abbiano un'interfaccia utente. Gli elementi che compongono il layout sono costruiti usando una gerarchia di *View* e *ViewGroup*; una View mostra sul display qualcosa che l'utente può vedere e con il quale può interagire, mentre le ViewGroup sono dei contenitori che definiscono la struttura del layout per le altre View, come mostrato in Fig. 6. Queste ultime vengono utilizzate per creare componenti UI, come Button o TextView. Usando queste risorse è più semplice personalizzare le varie caratteristiche dell'applicazione senza modificarne il codice.

Per ogni risorsa che viene inclusa nel progetto Android, l'SDK build tools definisce un identificativo univoco intero, attraverso il quale si ha un riferimento alla risorsa dal codice dell'applicazione. Quando le applicazioni vengono compilate, ogni file di layout è compilato come risorsa di tipo View e sarà quindi necessario caricare il layout che si intende mostrare inserendo il metodo appropriato nel codice del componente e specificando l'id corretto. Nel caso delle activity ad esempio, il metodo *setContentView()* si occupa di caricarne il layout e deve essere chiamato all'interno del metodo *onCreate()* (primo metodo del lifecycle che viene eseguito quando un'activity viene avviata).

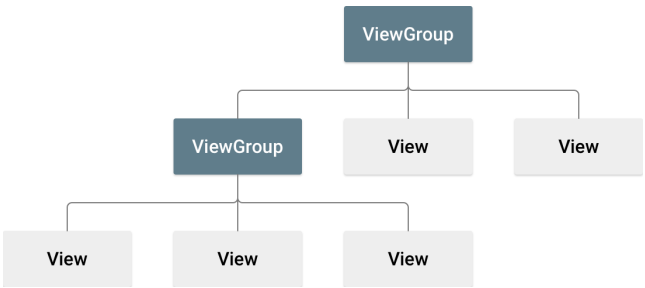

Figura 6: Esempio di gerarchia di ViewGroup e View

[14] `https://developer.android.com/guide/topics/manifest/manifest-intro`

3.4 Intra/Inter-App Communication

Come evidenziato in Sezione 3.3.1, un'applicazione è composta da una serie di componenti che possono essere di tipologie differenti. Per permettere la comunicazione interna all'app, il sistema consente di sfruttare un meccanismo atto a gestire la comunicazione tra componenti, detta Inter-Component Communication (ICC). Tale procedimento prevede di inviare al sistema un messaggio asincrono che specifichi un *intento* di avviare un determinato altro componente; il sistema si occupa di prendere in carico tali messaggi, detti **Intent**, e di eseguire la richiesta. Tre dei quattro componenti (activity, service e broadcast receiver) sono attivabili attraverso tali messaggi, creando dei collegamenti a runtime che permettono la comunicazione tra essi. I content provider al contrario, non sono attivati tramite intent, bensì azionati quando arriva una richiesta dal *ContentResolver*: esso gestisce tutte le transazioni dirette con il content provider, permettendo al componente che sta compiendo la transazione di utilizzare i metodi messi a disposizione dal ContentResolver stesso, garantendo un livello di astrazione tra il content provider e il componente.

Quando l'utente richiede l'avvio di una applicazione, il sistema operativo istanzia un processo per tale app (a meno che questa non sia già in esecuzione) e avvia il componente principale definito nell'Android Manifest (cfr. Sezione 3.3.2). Poiché, come descritto in Sezione 3.1.1, il sistema riserva ad ogni applicazione un processo separato con permessi che restringono l'accesso alle altre app, un'applicazione non può direttamente interagire con un'altra o attivare un componente non proprio atto a svolgere una particolare funzione: il meccanismo degli Intent può essere nuovamente sfruttato per la gestione dell'Inter-Process Communication (IPC). Anche in questo caso il sistema esegue la richiesta definita dall'Intent, controllando che essa non violi particolari permessi e creando un canale di comunicazione sicuro tra i moduli altrimenti isolati. Un intent può quindi essere definito come un oggetto che specifica un messaggio per attivare un determinato componente (sia esso o meno appartenente all'app) o uno specifico tipo di componente (in base alle funzioni svolte), permettendo la contestuale trasmissione di dati.

3.5 Eventi utente

In questa sezione si vuole tracciare una panoramica della gestione degli input utente da parte del sistema Android durante l'utilizzo di un'applicazione.

Esistono molti modi per intercettare gli eventi utente: focalizzandosi su ciò che riguarda l'interfaccia, l'approccio adottato è quello di catturare gli eventi attraverso l'ausilio di specifiche View con le quali l'utente interagisce. All'interno delle varie classi di View che vengono usate per comporre il layout, esistono molteplici metodi di callback pubblici, utilizzabili per la gestione degli eventi UI. Questi metodi sono chiamati dall'Android Framework quando si verifica una determina azione che coinvolge l'oggetto in questione: ad esempio, quando un bottone viene premuto, il sistema esegue l'*onTouchEvent()* definita per quell'oggetto. Per intercettare gli eventi di interesse e specificare il comportamento che l'applicazione deve tenere, una possibilità è quella di estendere la classe in questione e sovrascrivere il metodo di callback appropriato; in alternativa, poiché estendere ogni oggetto per gestire ogni evento può risultare poco pratico, le varie classi di View contengono una collezione di interfacce chiamate *event listener* che permettono una più semplice definizione dei metodi di callback: ogni event listener contiene un singolo metodo di callback che è chiamato dall'Android Framework quando la View sulla quale il listener è stato registrato viene stimolata da un utente. All'interno delle interfacce di event listener si possono trovare metodi di callback come: onClick(), onLongClick(), onFocusChange(), onKey(), onTouch(), onCreateContextMenu().

4 Stato dell'Arte

In questa sezione è descritto il risultato del processo di ricerca, studio ed esame dello stato dell'arte di tecniche e strumenti di analisi di applicazioni; tale fase è stata fondamentale al fine di comprendere quali fossero le metodologie più applicate, i casi d'uso, gli strumenti esterni utilizzati e i risultati ottenuti, e soprattutto allo scopo di studiare le limitazioni e i possibili miglioramenti da apportare. L'obiettivo primario, è stato capire quale fosse il metodo migliore per analizzare un'applicazione Android in modo da poter trovare il maggior numero possibile di vulnerabilità o comportamenti potenzialmente dannosi per l'utente.

4.1 Analisi di Applicazioni

Nel primo trimestre del 2018, su un totale di oltre 383 milioni di dispositivi venduti nel mondo, Android è stato il sistema operativo che ha guadagnato la fetta di mercato più ampia, raggiungendo l'85.9% di vendite contro il solo 14.1% di iOS [1]. A Marzo 2018, il numero di applicazioni disponibili al download superava i 3.6 milioni sul solo Google Play Store (100 mila in più rispetto al Dicembre dello stesso anno) [8], senza considerare tutte le app provenienti da store non ufficiali. Avendo contezza di questi dati, non stupisce il fatto che il problema della sicurezza Android sia molto attuale: trovare una metodologia che possa efficientemente stabilire quali rischi comporti l'utilizzo di una determinata applicazione Android è una questione di primaria importanza nell'ambito della ricerca.

Analizzando la letteratura scientifica presente sull'argomento, si possono delineare due tipi di approccio al problema: analisi statica e analisi dinamica dell'applicazione. La descrizione dettagliata delle caratteristiche di questi due tipi di analisi si trova nelle sezioni che seguono (Sezioni 4.2 e 4.3), ma ciò che è importante sottolineare è che entrambe le tipologie ricavano dei risultati a sé, ovvero non esiste necessariamente una reale compatibilità tra i dati ricavati dall'una o dall'altra analisi. Preso coscienza di ciò, la ricerca è stata incentrata su quali potessero essere i modi per legare i risultati delle due analisi. Allo stato dell'arte, tale approccio è definito *ibrido* ed è in fase di studio: alcuni ricercatori hanno provato a ideare soluzioni simili (cfr. Sezione 4.4), ma il problema è ostico e non sembra essere stato completamente risolto, a causa di varie motivazioni che saranno discusse in seguito. Alla luce di ciò, l'obiettivo è diventato quello di individuare i punti deboli e le limi-

tazioni degli strumenti di analisi di questo tipo per concepire un metodo ibrido più solido e innovativo.

4.1.1 Scopi delle Analisi

Dalla revisione dello stato dell'arte è emerso che l'analisi di applicazioni Android può essere impiegata allo scopo di raggiungere svariati risultati. È possibile comunque identificare alcuni obiettivi ricorrenti:

- Ricerca di vulnerabilità all'interno dell'applicazione, che potrebbero essere sfruttate da un attaccante per effettuare azioni malevole come *hijacking* o *injection*.
- Controllo dei permessi, per capire quali siano dichiarati, ma non realmente utilizzati o viceversa.
- Monitoraggio del consumo di energia, non solo per migliorare l'efficienza di un'applicazione, ma anche per rilevare app malevole che, secondo alcune ricerche (ad esempio [9]), possono essere identificate in base all'impronta o firma energetica.
- Ricerca di *private data-leaks*, ovvero rilevare se dati sensibili possano essere trasmessi al di fuori dell'applicazione senza che l'utente ne sia a conoscenza.
- Individuazione di cloni, per capire se un'applicazione sia stata clonata (ad esempio [10]) o per rilevare malware appartenenti a famiglie note (ad esempio [11]).
- Generazione di test case, sperimentabili in fase dinamica, per assicurare l'esecuzione di determinati percorsi.
- Verifica del codice per garantire la correttezza di una data applicazione (ad esempio controllare che i requisiti di sicurezza richiesti siano rispettati dall'app in esame).
- Ricerca di errori di memoria, per esempio utilizzo scorretto di allocazione e deallocazione o potenziale perdita di dati.
- Rilevamento di errori di concorrenza, ovvero verificare ad esempio che non siano presenti deadlock.

4.2 Analisi Statica

L'analisi statica di un software coinvolge generalmente degli strumenti che prendono in input il codice sorgente del programma, lo esaminano senza eseguirlo, e producono dei risultati attraverso il controllo della struttura, della sequenza di istruzioni e di come le variabili sono processate all'interno dei diversi metodi chiamati. Tipicamente un processo di analisi statica comincia con la rappresentazione del codice analizzato sotto forma di modello (ad esempio un call graph o un diagramma UML): tali modelli astratti sono scelti in base al risultato che si vuole ottenere in quanto ognuno di essi fornisce la base per l'utilizzo di tecniche di analisi di tipo diverso. La scelta di utilizzare un call graph (grafo che rappresenta le relazioni tra i metodi chiamati all'interno di un programma) nel contesto dell'analisi statica è decisamente la più comune: una possibile spiegazione per questa tendenza diffusa si può ricercare nel fatto che la maggior parte degli approcci statici per l'analisi di app sono implementati a partire da framework come Soot [12] o librerie come Wala [13] che forniscono strumenti in grado di facilitare la costruzione di call graph [14]. Per questo motivo, questo elaborato si concentra sull'utilizzo di call graph come base per l'analisi statica.

Il vantaggio più grande di questo tipo di analisi consiste nel fatto che, attraverso essa, si ha la certezza di analizzare l'intero codice del programma, diversamente da quanto accade utilizzando l'analisi dinamica. D'altro canto però, in caso di presenza di librerie di terze parti molto grosse o anche semplicemente nel caso di applicazioni molto pesanti, esiste la possibilità che l'analisi diventi gravosa da un punto di vista computazionale e quindi poco efficiente. Inoltre, un altro aspetto da tenere presente, è il fatto che i risultati dell'analisi potrebbero comprendere *falsi positivi*, ovvero porzioni di codice che vengono segnalate come sospette, ma in realtà non potranno mai essere eseguite in alcun modo (*dead code*).

4.2.1 Tecniche di Analisi

Le tecniche di analisi statica esistenti sono numerose e hanno scopi e ambiti di applicazioni diversi; inoltre sono spesso utilizzate in combinazione per ottenere risultati più complessi ed esaustivi. In questa sezione, sono elencate alcune di quelle che sono state prese in considerazione durante la fase di ricerca.

- **Control-flow analysis**: è una tecnica utilizzata per determinare il flusso di esecuzione di un'applicazione e di tutti i possibili percorsi che possono essere

effettuati, tenendo in considerazione i salti condizionati, ovvero punti del codice che alterano il sequenziale flusso delle istruzioni in base al fatto che una condizione logica sia verificata o meno. Il *control flow* viene espresso mediante un *control-flow graph* (CFG), un grafo diretto in cui ogni nodo rappresenta un blocco base di codice (una dichiarazione o un'istruzione) e ogni ramo indica un possibile flusso di controllo tra due nodi.

- **Data-flow analysis**: è una tecnica utilizzata per raccogliere informazioni riguardo i possibili insiemi di valori che le variabili di un programma possono assumere o, in altre parole, per comprendere come le variabili siano utilizzate in ogni punto del programma (ad esempio quali variabili abbiano valore costante o quali siano utilizzate prima di essere ridefinite). Questo tipo di tecnica è spesso utilizzato dai compilatori per l'ottimizzazione del codice.

- **Points-to analysis**: questa tecnica è un tipo particolare di *pointer analysis* che consiste nel calcolare un'astrazione statica di tutti i dati a cui un puntatore (o una variabile) può puntare durante l'esecuzione del software; un'altra tipologia di pointer analysis è l'*alias analysis*, che determina se sia possibile avere accesso ad una locazione in memoria in più di un modo.

- **Symbolic Execution**: è una tecnica che permette di determinare quali input portino all'esecuzione di un determinato punto del programma. Si procede assegnando valori simbolici alle variabili che vengono propagati durante la normale esecuzione del programma: questi valori sono utilizzati per generare espressioni e vincoli riguardanti i salti condizionali presenti nel codice, i quali dovranno essere successivamente risolti (ad esempio da un SMT solver) per produrre possibili valori di input che soddisfino i branch che si incontrano all'interno di un determinato percorso. Tali input sono poi utilizzati per generare un insieme di test case che verranno adoperati per esplorare realmente il percorso attraverso un'analisi dinamica. L'obiettivo di questa tecnica è solitamente quello di creare test case che permettano di garantire la più alta copertura di codice possibile in fase dinamica (*code coverage*).

- **Concolic Execution**: detta anche *concolic testing*, è una tecnica ibrida, che sfrutta sia la symbolic execution, sia la *concrete execution* (tecnica di testing che fa uso di input concreti, assegnando ad ogni variabile un valore iniziale

random), partendo dall'idea di guidare l'esecuzione simbolica mediante valori di input concreti. Tale procedura è di tipo ricorsivo e prevede i seguenti passi: esecuzione di un'analisi concreta del programma utilizzando valori reali casuali; esecuzione di una symbolic execution per costruire una rappresentazione simbolica del percorso risultante dagli input scelti; negazione dell'ultimo vincolo non ancora negato, ottenuto tramite l'esecuzione simbolica del path (se non vi è nessun vincolo non precedentemente negato, l'algoritmo termina); impiego di un SMT solver per trovare un insieme di valori che soddisfino la nuova espressione (se il vincolo non può essere soddisfatto, si ripete il passo precedente); reiterazione della procedura con i nuovi valori concreti, in modo da visitare un nuovo percorso di esecuzione. Le motivazioni principali che hanno portato alla nascita e all'utilizzo di questa tecnica sono probabilmente legate al tentativo di semplificare l'implementazione della symbolic execution che, senza il supporto della normale esecuzione del programma attraverso l'instrumentazione del codice, necessiterebbe dello sviluppo di un interprete simbolico per il linguaggio di programmazione considerato. Un altro vantaggio importante dato dall'utilizzo della concolic execution è senza dubbio il fatto che i valori concreti possono essere utilizzati per semplificare i vincoli trovati nel caso in cui essi siano molto complessi o addirittura non siano risolvibili da un solver (come ad esempio vincoli non lineari molto articolati).

- **Taint Analysis**: questa tecnica di analisi statica, si concentra sul flusso delle informazioni e dei dati all'interno di un programma. Può essere utilizzata per individuare fughe di dati (*data-leakages*) o per trovare vulnerabilità dell'applicazione legate alla presenza di input non correttamente validati. Nel primo caso, vengono *segnati* e quindi tracciati i dati considerati sensibili e tutti i dati dipendenti da essi: se uno di essi raggiunge un punto del codice sospetto (ad esempio un metodo che viola politiche di sicurezza o che invia dati all'esterno dell'applicazione), viene segnalata una possibile vulnerabilità. Nel secondo caso invece si *marcano* quei dati che provengono da fonti non verificate (o comunque modificabili dall'esterno) e tutti quelli dipendenti da essi: se uno di questi raggiunge un punto di codice pericoloso (ad esempio viene passato in ingresso ad una query SQL), viene segnalato l'utilizzo di un dato potenzialmente pericoloso.

- **Program Slicing**: è una tecnica utilizzata durante l'analisi per ridurre l'insieme potenziale di comportamenti da monitorare, al fine di snellire l'analisi mantenendo soltanto quelli più interessanti. Più formalmente, data una variabile di interesse 'v' del programma 'p', tale tecnica consiste nel considerare le sole istruzioni in 'p' che potrebbero influenzare il valore di 'v'.

4.2.2 Android Static Analysis Challenges

Questa sezione definisce le principali problematiche che possono affliggere un'analisi statica condotta su un'applicazione Android.

- Dalvik Executable: le applicazioni Android sono prevalentemente scritte in Java, ma sono compilate in DEX (Dalvik Executable) che, come descritto in Sezione 3.1.3, è un particolare tipo di bytecode sviluppato appositamente per questo sistema operativo. Gran parte degli analizzatori tradizionali per codice o bytecode Java non sono in grado di eseguire l'analisi ed è dunque necessario utilizzare un analizzatore statico progettato per Android che sia in grado di analizzare direttamente questo tipo di bytecode o, quantomeno, di tradurlo in un formato supportato.

- Entry Point: a differenza della maggior parte dei software, le applicazioni Android non presentano un singolo metodo *main*, bensì contengono svariati *entry point* che vengono chiamati dal framework di sistema a runtime (i.e. i lifecycle methods e le callback); per queste motivazioni costruire un call graph globale dell'app risulta decisamente più complicato.

- Lifecycle Methods: in Android, rispetto a Java o al C, i vari componenti di un'applicazione possiedono il loro personale ciclo di vita; ognuno di questi implementa i propri metodi di *lifecycle*, che vengono chiamati dal sistema per iniziare/fermare/riprendere l'esecuzione del componente, in base alle necessità del sistema stesso: ad esempio un'app in background può essere prima stoppata (ad esempio quando il sistema non ha a disposizione sufficienti risorse), e successivamente rimessa in esecuzione (*restarted*) quando l'utente esegue un'azione che ne comporta il ritorno in foreground. La difficoltà derivante da questa caratteristica consiste nel fatto che questi metodi non sono direttamen-

te collegati al flusso di esecuzione, e ciò ostacola la solidità di alcuni scenari di analisi, rendendo inoltre più difficile la costruzione di un call graph completo.

- Callback Events: oltre ai metodi di lifecycle, anche i metodi per gestire eventi utente (interazioni UI) ed eventi di sistema (e.g. bassa disponibilità di memoria, cambiamento della posizione del GPS) costituiscono delle problematiche per le analisi; infatti, siccome gli eventi possono accadere in ogni istante, gli analizzatori statici non sono in grado di costruire e mantenere un modello affidabile per gli eventi e diventa dunque complicato cercare di considerare tutte le possibili varianti di esecuzione.

- ICC: come già discusso nel capitolo precedente, la comunicazione all'interno di un'app Android avviene attraverso l'uso di *intent*; ci si riferisce solitamente a questo tipo di comunicazione come *Inter-Component Communication* (ICC). I metodi per attivare l'ICC utilizzano parametri speciali, contenenti tutte le informazioni necessarie a specificare il componente obiettivo, l'azione richiesta e i dati da trasmettere; tali metodi vengono processati dal sistema, il quale si prende carico di risolverli ed attivare il componente corretto a runtime. Di conseguenza, gli analizzatori statici trovano particolarmente complicato ipotizzare quale componente sia collegato ad un altro, a meno che non si faccia uso di euristica avanzata.

- Dynamic code loading: un possibile metodo per aggirare un tool di analisi statica è quello di caricare del codice esterno malevolo in fase di esecuzione. Android non applica controlli di sicurezza appropriati sul codice esterno e gli sviluppatori spesso non sono consapevoli dei rischi o non riescono a implementare meccanismi di protezione corretti e robusti. Poiché le tecniche per caricare il codice runtime vengono spesso utilizzate anche da applicazioni benigne, i sistemi di analisi non possono utilizzare la semplice presenza di tale funzionalità come prova certa della presenza di un malware [15].

- Java-Inherited: a tutte queste problematiche si aggiungono quelle derivanti da Java stesso, come la gestione della riflessione, del codice nativo, del multithreading e del polimorfismo.

4.3 Analisi Dinamica

L'analisi dinamica consiste nell'eseguire un'applicazione su un dispositivo virtuale (di solito un emulatore) con il fine di monitorarne il comportamento. Affinché questo tipo di analisi possa essere considerata efficace, il software in esame deve essere eseguito avendo a disposizione un numero sufficiente di *test input* in modo da riuscire a stimolare l'applicazione: riproducendo infatti il maggior numero possibile di eventi e catene di eventi, si ha una conseguente più alta probabilità di rilevare comportamenti sospetti al fine dell'analisi.

La stimolazione di un software può essere condotta manualmente o automaticamente. Nel caso di stimolazione automatica, prendendo in considerazione software ad eventi e/o che prevedano interazione con l'utente, l'impiego di input generati in modo casuale o semi-casuale, spesso non permette di stimolare possibili comportamenti malevoli, che potrebbero infatti presentarsi soltanto in seguito ad un determinato tipo di interazioni eseguite da un utente, o come risultato di determinate condizioni del dispositivo. In generale, rispetto all'analisi statica, non si riesce ad arrivare all'analisi completa di tutto il codice di cui il software è composto, in quanto la natura ad eventi e l'imprevedibilità dell'interazione utente rendono il dominio di esecuzione infinito; d'altro canto però, la stimolazione di un percorso considerato sospetto, permette di stabilire se una determinata azione possa essere effettivamente compiuta o meno. L'analisi dinamica infatti, non soffre di alcune delle problematiche legate all'analisi statica, in quanto non influenzata dalla natura del codice eseguito, sia esso offuscato, protetto da crittografia o da qualsiasi altro mezzo contro l'analisi statica.

4.3.1 Android Dynamic Analysis Challenges

Questa sezione descrive le principali problematiche legate ad un'analisi dinamica condotta su un'applicazione Android.

- Dominio di esecuzione: come già anticipato in Sezione 4.3, un grande problema che affligge l'analisi dinamica è sicuramente quello legato al numero di eventi e combinazioni di essi che possono verificarsi durante una normale esecuzione di un software. In particolare, un'applicazione Android prevede una grande interazione con l'utente ed è fortemente condizionata dalla condizione del dispositivo su cui è installata: il framework di sistema gestisce gli eventi di

sistema (legati ad esempio alla batteria o alla memoria del dispositivo) e quelli esterni (ad esempio user input, SMS o chiamate) invocando gli opportuni lifecycle methods dei componenti dell'app, modificando il flusso di esecuzione (cfr. Sezioni 3.5 e 4.2.2) che dipende chiaramente dall'ordine in cui gli eventi si verificano; ciò rende impossibile per un tool di analisi dinamica prevedere quali possano essere tutti i possibili percorsi da testare.

- Delayed execution: un metodo diffuso per aggirare un'analisi di tipo dinamico è quello di programmare un'esecuzione posticipata del codice malevolo. In questo modo, l'applicazione attende il verificarsi di un evento particolare (ad esempio lo scadere di un timer, la ricezione di un comando inviato da un server remoto o l'arrivo di un SMS da un particolare mittente) prima di eseguire le istruzioni pericolose e ciò rende molto più complesso replicare tale comportamento in fase di analisi dinamica; chiaramente più il codice malevolo è difficile da raggiungere, più è facile che l'analisi dinamica non riesca a rilevarlo.

- Emulatore: il fatto di utilizzare un emulatore per condurre l'analisi dinamica, concede agli autori di malware la possibilità di adottare contromisure per rilevare l'esecuzione in ambiente protetto; in questo modo un'applicazione malevola può assumere comportamenti benigni per sfuggire all'analisi [16].

4.4 Tool Esistenti

Questa sezione si prefigge l'obiettivo di descrivere alcuni degli strumenti di analisi di applicazioni che sono stati analizzati e studiati durante il processo di revisione dello stato dell'arte.

Come già anticipato in Sezione 4.1, il problema di effettuare analisi su applicazioni Android è di primaria importanza per la comunità, e ciò è dimostrato anche dal numero elevatissimo di tool di questo tipo, ognuno dei quali si concentra su aspetti e scopi specifici. Il processo di ricerca ha evidenziato la tendenza a prediligere un approccio statico rispetto a quello dinamico.

Tra i più interessanti tool statici esistenti, ci sono ad esempio Approver [17], un tool professionale per l'analisi completa delle applicazioni mobili sviluppato da Talos S.r.l.s in collaborazione con il Computer Security Lab del dipartimento DIBRIS del-

l'Università di Genova, e Amandroid [18]; effettuando però la sola analisi statica del codice, questi tool non sono in grado di garantire con assoluta certezza che tutto ciò che sia individuato come potenzialmente pericoloso, possa poi presentare realmente una minaccia durante un normale utilizzo dell'applicazione.

Tra i tool dinamici più notevoli vi sono invece DroidBot [19], AppsPlayground [20] e TaintDroid [21]. Nei primi due casi, si è notato un approccio simile, basato sulla stimolazione dell'applicazione in maniera semi-casuale: vengono innanzitutto effettuate delle assunzioni in base agli elementi di interfaccia posseduti dal componente in foreground, per poi cercare di stimolare l'esecuzione di quanti più comportamenti possibili sfruttando bottoni, editText, checkbox e altri elementi ricavati dall'interfaccia; la casualità è data dal fatto che gli input inseriti e l'ordine di esecuzione degli eventi sono randomici. Nel caso di DroidBot esiste inoltre la possibilità di creare degli script che eseguano delle azioni prestabilite, ma questa caratteristica aggiuntiva è scarsamente supportata e gli script configurabili permettono un numero di azioni davvero limitate. Un tipo di analisi del genere non è dunque in grado di arrivare alla stimolazione di vulnerabilità che dipendono dall'esecuzione di azioni ed eventi ben precisi. TaintDroid è invece un tool per analisi dinamica diverso rispetto ai due precedenti, poiché si occupa esclusivamente di monitorare ciò che accade all'interno di un dispositivo Android alla ricerca di fughe di dati attraverso il tracciamento delle sorgenti di dati sensibili; questo tool necessita di essere inserito all'interno di un'immagine di Android scaricata da AOSP (*Android Open Source Project*[15]) e perché possa produrre risultati, ha bisogno che un analista stimoli manualmente il sistema all'interno del quale viene inserito.

Come spiegato in Sezione 4.1, l'ultima fase del processo di ricerca è stata quella di analisi oculata delle pubblicazioni riguardanti metodi di analisi ibrida, che coinvolgessero sia l'analisi statica sia quella dinamica. Ad eccezione di un'unica pubblicazione tra quelle trovate, che approccia questa metodologia creando un modello in fase statica, per poi raffinarlo in un secondo momento attraverso i dati raccolti dall'esecuzione vera e propria dell'applicazione [22], il resto dei lavori segue un metodo piuttosto ricorrente, che consiste nell'utilizzo dell'analisi statica per ottenere dei percorsi di esecuzione con relativi vincoli, da utilizzare successivamente per ef-

[15] Per informazioni a riguardo, si rimanda al sito ufficiale `https://source.android.com/`

fettuare test dinamici. Di seguito sono elencati una serie di pubblicazioni analizzate che sfruttano questa metodologia di analisi in maniera interessante.

AppIntent [23] è un articolo del 2013 che propone un tool automatico (non disponibile online) che, data una trasmissione di dati sensibili, fornisce la sequenza di manipolazioni GUI che corrisponde alla sequenza di eventi che hanno portato alla trasmissione stessa; lo scopo del framework è quello di permettere ad un analista umano di discriminare in maniera più semplice tra *user-intended data transmission* e *unintended data transmission*. AppIntent fa uso di una nuova tecnica sviluppata dagli autori, chiamata *event-space constraint guided symbolic execution*, ovvero una versione modificata della symbolic execution che mira a limitare il problema del path explosion.

SIG-Droid [24] è un framework utilizzato per il testing di applicazioni Android; sfrutta un programma di analisi per estrarre il modello dell'applicazione e la symbolic execution guidata da tale modello per ottenere degli input test che assicurino la copertura di ogni branch raggiungibile nell'applicazione. Si basa su due tipi di modello: l'*Interface Model*, che si occupa di trovare i valori che un'app può ricevere dall'interfaccia, che vengono poi trasformati in valori simbolici utilizzabili per la costruzione di vincoli; il *Behavior Model*, usato per guidare la symbolic execution e generare sequenze di eventi. Anche per questo lavoro, non è stata trovata alcuna traccia di un prototipo disponibile sul web.

IntelliDroid [25] è un prototipo di un tool che fa uso di una tecnica di analisi definita dagli autori *targeted analysis*; secondo quanto evinto dal paper, tale tecnica genera un piccolo insieme di test case (in fase statica), che dovrebbero essere in grado di poter innescare i comportamenti malevoli in modo che questi possano essere individuati durante un'analisi dinamica. Grazie alla disponibilità del prototipo su GitHub [26], il tool è stato scaricato ed analizzato. La parte statica è risultata interessante, poiché la cosiddetta targeted analysis si è rivelata essere sostanzialmente una symbolic execution che si concentra sul raggiungimento di parti di codice ben precise (e configurabili). Per comprendere quali siano i punti del codice che potrebbero portare all'esecuzione di comportamenti malevoli se raggiunti durante l'esecuzione, gli sviluppatori hanno condotto una ricerca secondo la quale il

rilevamento di un malware può avvenire staticamente attraverso l'analisi delle API utilizzate, delle system call o degli effetti collaterali di basso livello dell'applicazione (come l'utilizzo della batteria); gli autori hanno deciso di sfruttare una lista di *targeted Android APIs* per costruire una sovra-approssimazione dei comportamenti pericolosi di un'app. Viene lasciata all'utente la possibilità di specificare target di qualsiasi tipo, per poter personalizzare l'analisi.

I creatori di IntelliDroid sostengono di lavorare in un contesto di control e data flow, in modo da individuare non solo il singolo metodo che contiene l'API, ma l'intera serie di eventi (event-chain), il loro ordine e gli specifici input che portano alla chiamata dell'API a partire da un entry point dell'applicazione. Dai risultati dei test condotti sul prototipo, e da una accurata analisi del codice sorgente, è emerso che il metodo di costruzione di questa catena non porta sempre a risultati corretti e non è generalizzabile a tutti i possibili tipi di evento verificabili: la metodologia sfrutta come modello di analisi un call graph in cui ogni nodo rappresenta un metodo dell'applicazione; ogni metodo di callback (sia quelli di sistema, come i metodi di lifecycle, sia quelli dipendenti da eventi utente, come le onClick()) sono considerati entry point. Gli archi tra i nodi rappresentano possibili flussi di esecuzione tra i metodi dell'applicazione: a parte non prevedere il collegamento tra i nodi di lifecycle, come per la maggior parte dei call graph di applicazioni Android in quanto ciò renderebbe la costruzione del modello molto complessa, gli autori non prendono in considerazione tutti quei collegamenti originati dall'ICC e dall'interazione dell'utente con l'applicazione, che sono fondamentali per condurre un'analisi utile e completa. Quasi tutti i percorsi trovati iniziano direttamente con il nodo che contiene l'API, senza aver traccia di quale sia il reale percorso che può portare al raggiungimento di questo metodo a partire dal launcher effettivo dell'applicazione (solitamente la MainActivity). Per gli autori questo comportamento è accettabile in quanto, in fase di stimolazione, si concentrano su eventi di sistema (e.g. ricezione di sms, cambio coordinate GPS) ed utilizzano la shell di adb per raggiungere direttamente ogni componente dell'applicazione. Questo modo di procedere però causa la perdita di riferimento a molti dati che potrebbero essere necessari alla corretta esecuzione del path (come l'utilizzo di informazioni inviate da altri componenti tramite ICC), e in ogni caso fallisce se applicato alla stimolazione di un normale flusso di esecuzione guidato da input utente, che non è infatti previsto da IntelliDroid.

Per quanto riguarda la parte di analisi dinamica, IntelliDroid sfrutta l'utilizzo a

runtime dell'SMT solver Z3[16] per risolvere i vincoli trovati, e TaintDroid per monitorare le informazioni del device. Una caratteristica interessante riguarda il fatto che gli autori propongono l'injection di eventi a livello Device-Framework anziché a livello di Application, che assicura una consistenza maggiore nel caso di simulazione di eventi dipendenti dal sistema; per fare ciò è però indispensabile instrumentare l'ambiente dinamico: è necessario scaricare un'immagine di Android da AOSP e customizzarla attraverso una patch di TaintDroid e una di IntelliDroid, procedura abbastanza lunga e complessa. A parte questa notevole caratteristica, la mancanza di informazioni complete estratte dall'analisi statica, l'incapacità di stimolare eventi di interfaccia e l'impossibilità di ottenere un percorso di esecuzione completo a partire dal launcher dell'applicazione, rendono questo tool non adatto a raggiungere l'obiettivo dell'elaborato.

Gli strumenti e i lavori analizzati falliscono nel creare una completa sinergia tra le tecniche di analisi statica e dinamica; per questo motivo la tesi di ricerca è volta a proporre un nuovo approccio di analisi, in grado di apportare miglioramenti rispetto alle varie metodologie descritte all'interno del capitolo appena concluso.

[16] `https://github.com/Z3Prover/z3`

5 Approccio CAP

Il processo di ricerca è stato eseguito con il fine di ottenere una visione completa dello stato dell'arte per acquisire non solo informazioni riguardo alle metodologie di analisi esistenti, ma anche conoscenza delle limitazioni dei vari approcci e spunti di miglioramento, in modo da arrivare ad ideare una metodologia di analisi robusta e funzionale. L'obiettivo di questa tesi è quello di definire un nuovo approccio che sia in grado di coniugare le tecniche di analisi statica con quelle di analisi dinamica. Per questo motivo sono stati sviluppati inizialmente una metodologia di analisi e un framework di riferimento chiamato **CAP** (**Combining Analysis Processes**), e in un secondo momento la relativa implementazione per il sistema operativo Android, chiamato **CAPDroid**, che ha l'obiettivo di mostrare la fattibilità e l'efficacia dell'approccio proposto. Nel prosieguo di questo capitolo saranno descritti la metodologia e il framework CAP. Per l'implementazione e i risultati sperimentali si rimanda ai capitoli successivi.

5.1 Metodologia

Come già discusso in precedenza (Sezione 4), una delle problematiche riscontrate nel contesto dell'analisi di software è quella relativa alla scarsa compatibilità dei risultati di analisi statica e dinamica, che, a causa delle limitazioni dei due tipi di analisi e della loro indipendenza, devono essere in qualche modo combinati con il fine di avere un risultato globale che mitighi gli aspetti negativi delle due.

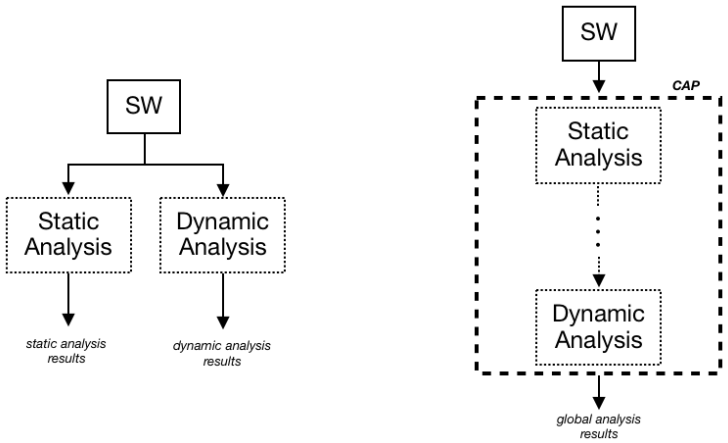

Figura 7: Combining Analysis Processes

Come mostrato in Fig. 7, a differenza di quanto accade utilizzando l'analisi statica e quella dinamica in maniera separata, comportamento tipico degli approcci tradizionali, la metodologia CAP prevede l'esecuzione di queste tecniche in maniera sequenziale, facendo in modo che i risultati tra le diverse fasi previste siano interconnessi e propedeutici all'attuazione delle fasi successive; questo consente di ottenere un'esecuzione mirata del software utilizzando dunque una tecnica ibrida, al fine di sfruttare le caratteristiche positive di entrambe le tecniche. Il metodo CAP si serve dei risultati ottenuti da una prima fase di analisi statica, opportunamente filtrati e normalizzati, per creare un insieme di input atti all'esecuzione mirata e automatizzata del software in un ambiente dinamico, verificando quindi se i comportamenti potenzialmente pericolosi individuati durante la fase statica, possano essere effettivamente eseguiti.

5.1.1 Stimolazione dell'Interfaccia Utente

Questa sezione è volta a porre particolare attenzione su una importante caratteristica del metodo CAP, ovvero il grande interesse prestato alla gestione degli eventi UI; questa peculiarità è da considerarsi un punto di forza della metodologia, in quanto la stimolazione accurata dell'interfaccia utente presenta per natura diverse limitazioni che la rendono particolarmente ostica, come descritto ad esempio in [27], [28] e [29]. In caso di presenza di GUI, per automatizzare l'emulazione del comportamento che potrebbe avere un utente reale durante l'utilizzo di un software, è necessario che la metodologia preveda non solo il riconoscimento di tutte le schermate che il programma può mostrare durante la sua esecuzione e della sequenza in cui queste possono essere visualizzate, ma anche il rilevamento di tutti gli elementi che compongono le schermate stesse: deve dunque essere possibile distinguere gli elementi di layout, la loro posizione e il loro scopo in modo da poter simulare l'interazione utente in fase dinamica. A tal fine, la fase di analisi statica è fondamentale e deve essere effettuata in modo da fornire, oltre ai percorsi verso i target con relativi vincoli, un modello che comprenda tutti gli elementi dell'interfaccia utente con relativi dettagli riguardanti il tipo di elemento, la sua collocazione e altre informazioni necessarie ad identificarlo univocamente e a comprendere come esso possa essere stimolato. È dunque possibile generare dei test case che replichino il comportamento che un utente avrebbe se dovesse utilizzare l'applicazione per raggiungere i target considerati.

5.2 Il Framework CAP

La metodologia CAP può essere attuata attraverso un framework suddiviso in diverse fasi, ognuna delle quali risulta propedeutica per quella successiva. Le sezioni che seguono descrivono nel dettaglio il framework, la cui architettura è visibile in Fig. 8.

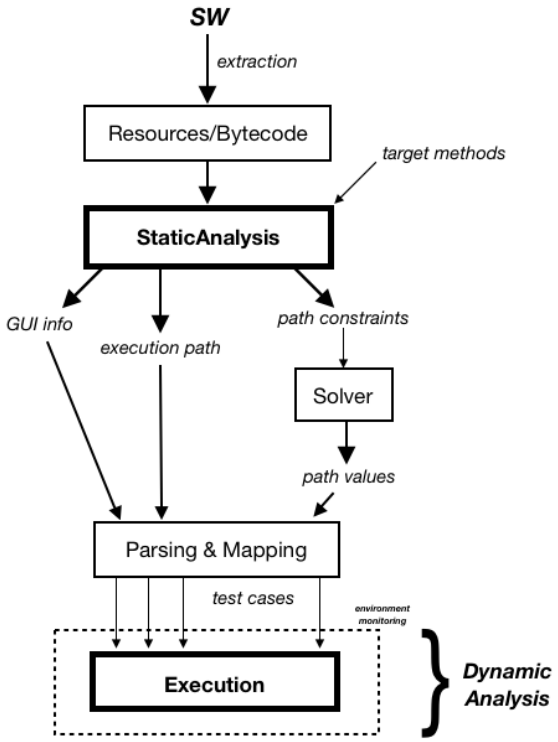

Figura 8: Architettura generica Framework CAP

L'input del sistema nella sua interezza è composto dal software da analizzare e da un file di testo contenente i metodi che devono essere considerati i target dell'analisi. È importante sottolineare come l'approccio sia stato progettato in modo da essere altamente configurabile: il metodo CAP è in grado di risalire ad ogni chiamata di una qualsiasi funzione che venga indicata all'inizio dell'analisi come metodo *target*, e di ricostruire la catena di eventi che devono essere eseguiti per raggiungere tale metodo con annessi vincoli sugli input utente.

5.2.1 Extraction

La prima fase prevista dal metodo CAP ha il compito di manipolare il software in ingresso in modo da estrarne il bytecode e tutte le risorse di cui si serve; queste

operazioni devono essere effettuate mediante un tool che deve essere scelto in base al tipo di programma sotto esame.

5.2.2 Static Analysis

La seconda fase è quella di analisi statica: questo stadio è fondamentale in quanto si occupa di estrarre tutte le informazioni riguardanti il software, necessarie al corretto completamento delle successive analisi. Facendo riferimento a quanto descritto in Sezioni 4.2 e 4.4, la principale tecnica di analisi scelta per questa fase è una versione guidata e mirata dell'esecuzione simbolica, che punta all'individuazione di target specifici (e configurabili) all'interno del codice, prendendo spunto da quanto fatto in [25]; in questo modo si è in grado di limitare la ricerca di percorsi possibili ai soli che corrispondano a determinati comportamenti. Il modello pensato per condurre l'analisi è un control flow graph (cfr. Sezione 4.2) generabile a partire dall'analisi del bytecode del software. Nello specifico, questo blocco del framework CAP prevede l'esecuzione di diverse fasi indispensabili per l'ottenimento dei risultati desiderati:

1. Analisi delle risorse: è necessario condurre un'analisi delle risorse estratte durante il preprocessing del programma. Questo procedimento è fortemente dipendente dal tipo di software in esame, e necessita di uno studio preliminare che evidenzi quali siano le risorse e le informazioni da prendere in considerazione.

2. Ricostruzione della *class hierarchy*: deve essere ricostruita la class hierarchy, ovvero la gerarchia delle classi del programma che definisce le dipendenze tra esse; tale procedura deve essere eseguita a partire dal bytecode del programma stesso.

3. Individuazione degli entry point: sulla base delle informazioni estratte dalle risorse, è possibile individuare gli entry point del software. Per quanto riguarda l'analisi di software basati sulla programmazione ad eventi[17], gli entry point possono essere numerosi e non sempre riconoscibili a partire dalle sole risorse; pertanto questo insieme deve essere raffinato nella fase successiva.

[17] L'event-driven programming è un paradigma di programmazione in cui il flusso di esecuzione è determinato dal verificarsi di eventi. Per ulteriori informazioni a riguardo, si rimanda al sito `http://www.technologyuk.net/software-development/designing-software/event-driven-programming.shtml`

4. Costruzione incrementale del modello: la metodologia CAP prevede la costruzione di un primo call graph a partire dalla conoscenza degli entry point, della gerarchia delle classi del programma, del bytecode e delle informazioni estratte in seguito all'analisi delle risorse. Tale grafo deve poi essere analizzato alla ricerca di nodi che possano essere considerati ulteriori punti di accesso al programma, con il fine di completare l'insieme precedentemente individuato: il modello può essere dunque raffinato mediante l'iterazione del processo finché non risulti essere accurato. Anche in questo caso, la possibilità di analizzare programmi event-driven rende il processo più complicato in quanto i call graph tradizionali non considerano le connessioni tra nodi scaturite dal verificarsi di eventi: ciò rende necessaria l'analisi approfondita di ogni nodo del grafo al fine di tenere traccia di quelli che contengono istruzioni legate ad eventi che permettano una connessione implicita con un altro nodo (un esempio di nodo da tracciare è un nodo che contenga la registrazione di una callback su un oggetto della GUI o l'istruzione che richiede al sistema la visualizzazione di una particolare finestra). Per evitare di ripetere l'analisi del codice di ogni nodo, che viene già effettuata durante la generazione del modello e che risulta dispendiosa in termini di tempo, il procedimento appena descritto può essere eseguito contemporaneamente alla costruzione del grafo. Nello stesso modo è inoltre possibile tenere traccia dei nodi che contengono una o più istruzioni target, informazione fondamentale per l'analisi successiva.

5. Targeted symbolic execution: il grafo generato può essere utilizzato per effettuare un'esecuzione simbolica mirata, ovvero una versione modificata della symbolic execution che punti ad ottenere tutti i percorsi che portino alla chiamata dei metodi target a partire dagli entry point, con annessi vincoli (cfr. Sezione 4.4).

6. Backwards path dependencies: le informazioni aggiuntive ottenute al punto 4, riguardanti i collegamenti tra i nodi non presenti nel grafo, possono essere utilizzate per migliorare i percorsi trovati attraverso un processo iterativo. Nel caso in cui il primo nodo di un percorso non possa essere raggiunto da nessun altro nodo, tale percorso può essere considerato completo; al contrario, se il primo nodo del path dipende da almeno uno degli altri nodi del grafo (per esempio nel caso in cui il nodo sia una callback), lo si può inserire in

testa al percorso, ottenendo un potenziale nuovo path. Ogniqualvolta venga aggiunto un nodo, questo deve essere analizzato in modo da tener traccia di come arrivare alla chiamata dell'istruzione che permette effettivamente il collegamento implicito (ad esempio tale nodo potrebbe contenere l'istruzione di registrazione della callback dell'esempio precedente). Questo processo deve essere ripetuto fino a trovare un nodo indipendente ed è necessario per ottenere percorsi completi nel caso di analisi di programmi ad eventi. Ogni nuovo path trovato è da considerarsi *feasible* solo se il primo nodo è un entry point del programma.

7. Guided & targeted symbolic execution: l'ultimo stadio dell'analisi statica prevede la generazione dei constraint per i nuovi path. È possibile ottenere questo risultato eseguendo una versione modificata della precedente esecuzione simbolica mirata, in modo che non solo punti a raggiungere i target finali, ma sia inoltre guidata dai nuovi percorsi scoperti.

Il risultato dell'analisi statica è particolarmente utile in fase dinamica, in quanto permette di ottenere solamente test case rilevanti, consentendo di eseguire solo una piccola percentuale di codice rispetto al totale e diminuendo quindi notevolmente il tempo necessario all'analisi. L'output di questa fase comprende i path che portano all'esecuzione delle porzioni di codice sospette, i vincoli che devono essere soddisfatti affinché tali percorsi siano eseguiti e tutte quelle informazioni riguardanti il software (e.g. elementi dell'interfaccia utente) indispensabili per le fasi seguenti.

5.2.3 Solver

Questa fase risulta necessaria a causa dell'utilizzo della symbolic execution: è fondamentale infatti l'impiego di un SMT solver al fine di risolvere i vincoli generati nello stadio precedente, in modo da assegnare alle variabili simboliche dei valori reali che soddisfino i constraint. La scelta dell'SMT solver condiziona il resto dell'analisi in quanto, nel caso in cui esso non riesca a portare a termine la risoluzione di uno dei vincoli, il path corrispondente non potrà essere stimolato in fase dinamica.

5.2.4 Parsing & Mapping

Lo stadio che segue è quello di manipolazione dei risultati e generazione dei test case per la stimolazione dinamica. L'obiettivo di questa fase è quello di costruire

degli script, uno per ogni percorso segnalato dalle fasi precedenti, che permettano di stimolare la rispettiva catena di eventi in fase dinamica e che contengano quindi tutte le informazioni necessarie affinché la stimolazione possa aver luogo: in altre parole gli script devono essere scritti in un formato compatibile con l'ambiente dinamico scelto per la successiva fase e devono contenere tutti i dati raccolti dagli stadi precedenti formattati in modo opportuno. L'algoritmo di creazione ideato consiste, nello specifico, nell'assegnare ad ogni nodo del path le istruzioni opportune che devono essere eseguite dall'emulatore affinché il metodo che rappresenta sia eseguito correttamente (i.e. con i giusti input e al momento giusto). Per poter permettere la stimolazione di programmi event-driven, è fondamentale che tali istruzioni siano scelte in base alla tipologia di evento che deve verificarsi e in accordo con un *mapping* che deve essere precedentemente effettuato: è necessario infatti condurre uno studio preliminare sull'insieme di tutti gli eventi che possono verificarsi ed effettuare una mappatura che colleghi ogni evento alle istruzioni da eseguire per stimolarlo.

5.2.5 Dynamic Analysis

L'analisi dinamica è la parte conclusiva dell'intero processo ed è stata ideata su ispirazione degli script configurabili di DroidBot [19]: mediante l'utilizzo di ambienti con sandbox o emulatori, si eseguono i test case generati precedentemente in modo da stimolare i comportamenti sospetti in maniera automatica, verificando se le parti di codice potenzialmente pericolose possano realmente essere eseguite.

A questo punto, il sistema può essere supportato mediante il monitoraggio dell'ambiente di esecuzione del software: sfruttando una tecnica di *monitoring* del dispositivo durante la fase di stimolazione dinamica, come ad esempio un proxy per il traffico di rete o un monitor di chiamate API, si possono avere informazioni certe su cosa accada realmente sul device; ad esempio si potrebbe verificare l'effettivo invio verso l'esterno di dati sensibili o il tentativo di collegarsi alla rete mediante protocolli di comunicazione non sicuri. Quest'ultimo modulo è chiaramente dipendente non solo dal tipo di contesto in cui si sta operando, ma anche da quale vulnerabilità si stia cercando e da quali siano i sistemi esistenti disponibili in commercio.

6 CAPDroid: un'Implementazione

In questa sezione dell'elaborato è presentato **CAPDroid** (**Combining Analysis Processes for Android**), il prototipo di un tool progettato per il sistema operativo Android, in grado di rilevare percorsi che portino a potenziali vulnerabilità all'interno di un'applicazione, verificandone al contempo l'effettiva possibilità di esecuzione durante il normale utilizzo utente dell'app. Al fine di descrivere al meglio l'implementazione e per motivare le varie scelte effettuate, sono presenti riferimenti ai tool presentati in Sezione 2, al contesto di lavoro (i.e. il sistema Android e le sue applicazioni, argomenti trattati in Sezione 3), ai tipi di analisi e le relative tecniche presentate in Sezione 4, e alla metodologia CAP descritta nel capitolo precedente.

6.1 Architettura

Il sistema progettato, sulla base di quanto descritto in Sezione 5.2, si divide in vari moduli: come si nota dalla Fig. 9, si possono individuare sostanzialmente cinque diverse fasi, eseguite in sequenza, che a partire da un'applicazione in formato APK e da un file di testo che definisce i metodi target, portano all'injection di test case all'interno di un emulatore Android. Per motivazioni chiarite in Sezione 6.1.2, si è deciso di fornire in input al sistema anche il file .jar contente una delle versioni delle librerie di base di Android; per mantenere lo strumento configurabile, è possibile sostituire la libreria fornita di default. Forniti i target, le librerie e l'applicazione, l'output di ogni modulo rappresenta l'input per il modulo successivo, in modo tale che il sistema possa funzionare in completa autonomia.

Le sezioni che seguono descrivono le fasi di analisi in maniera dettagliata.

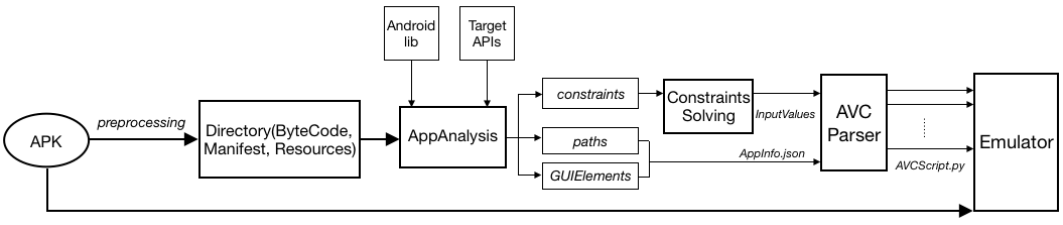

Figura 9: Architettura di CAPDroid

6.1.1 Preprocessing

La prima fase si occupa di pre-processare l'applicazione fornita in ingresso: all'inizio del processo l'applicazione è infatti in formato APK, ed è quindi impossibile applicare tecniche di analisi poiché il codice sorgente e le relative risorse non sono disponibili, essendo compresse all'interno dell'APK stesso. Per poter estrarre da questo tipo di archivio il bytecode e i vari file XML fondamentali per l'analisi, si fa uso di due strumenti open source reperibili online, quali ApkTool e Dare (presentati in Sezione 2.2): dall'utilizzo dei due, si ottiene in output una cartella contenente tutte le risorse XML dell'app raccolte in una directory chiamata res, l'Android Manifest e i file classes.dex e classes.jar, ovvero tutti gli elementi necessari per condurre l'analisi statica.

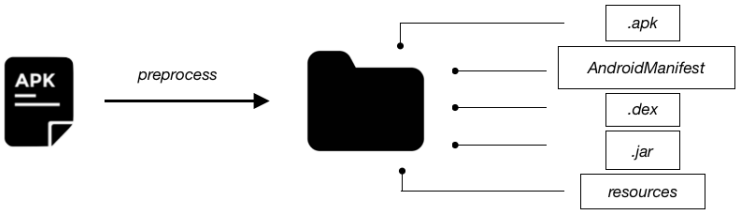

Figura 10: Modulo di Preprocessing

6.1.2 Static Analysis

Il modulo successivo dell'architettura è atto all'esecuzione dell'analisi statica: come descritto in Sezione 5.2.2, tale stadio è di fondamentale importanza in quanto ha il compito di estrarre le informazioni necessarie al corretto proseguimento del processo di analisi. Il linguaggio utilizzato per implementare questo blocco è Java; viene inoltre sfruttata la libreria di analisi Wala (cfr. Sezione 2.3).

L'input di questo blocco consiste nel risultato della fase di preprocessing dell'APK, nelle librerie di Android e nel file contentente la lista di target method da prendere in considerazione per l'analisi, identificati mediante il modo in cui sono rappresentati nel bytecode (ad esempio android.telephony.TelephonyManager.getDeviceId() Ljava/lang/String;). Conformemente a quanto descritto in Sezione 5.2.2, si possono individuare diverse fasi necessarie a concludere l'analisi:

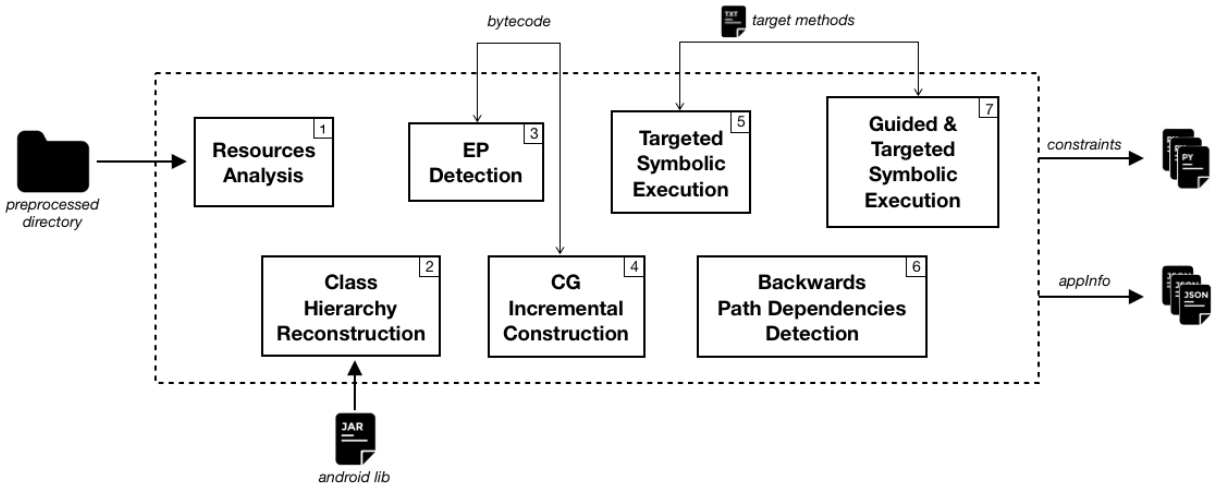

Figura 11: Modulo di analisi statica

1. Analisi delle risorse: il modulo prevede innanzitutto l'analisi dell'Android Manifest in modo da ottenere le informazioni relative all'applicazione e ai i componenti in esso dichiarati; in particolare sono estratti il package, il nome dell'app e per ogni componente il nome, il valore del tag *exported* e le informazioni sull'*intent filter* (cfr. Sezione 3.3.2). Sono poi esaminati tutti i rimanenti file XML di risorse riguardanti la GUI dell'applicazione: per ogni elemento di layout trovato si tiene traccia dell'id corrispondente, del tipo (e.g. Button, TextView, EditText) e, nel caso sia definita una callback, delle informazioni riguardanti l'onClick (cfr. Sezione 3.5). Inoltre viene individuata l'activity launcher dell'app in base all'intent filter.

2. Ricostruzione della class hierarchy: a partire dal bytecode dell'applicazione e sfruttando i metodi messi a disposizione da Wala, si ricostruisce la gerarchia delle classi dell'applicazione. Poiché il file APK non contiene tutte le classi base delle librerie Android, bensì solo quelle di supporto utilizzate realmente dall'app, per completare la costruzione della gerarchia delle classi, è stato necessario inserire una versione del file .jar di librerie di base Android (che può essere sostituita da una qualunque versione a scelta dell'utilizzatore). Eventuali problemi di compatibilità di CAPDroid con app che sfruttino librerie più recenti sono da ricondurre alla scelta iniziale effettuata dall'utilizzatore.

3. Individuazione degli entry point: a partire dalla lista dei componenti estratta al punto 1, si individuano i punti di accesso dell'app che consistono nei metodi di lifecycle dei componenti e dell'applicazione stessa e dalle callback precedentemente individuate.

4. Costruzione incrementale del modello: in accordo con quanto descritto in Sezione 5.2.2, il grafo viene costruito in modo incrementale. Questa scelta è risultata particolarmente utile per il caso di applicazione considerato, in quanto non tutti gli entry point di un'app Android possono essere individuati a partire dalle informazioni estratte dalle risorse: ad esempio i BroadcastReceiver possono non essere dichiarati nell'Android Manifest (cfr. Sezione 3.3.2), così come non è obbligatorio indicare esplicitamente la callback di un elemento di layout contestualmente alla definizione di esso nell'XML corrispondente (cfr. Sezione 3.5). Nello specifico, il modello è generato mediante l'utilizzo di un metodo di Wala che permette di costruire un call graph affiancando questa operazione alla realizzazione di una pointer analysis (cfr. Sezione 4.2.1) seguendo una politica di tipo ZeroOneCFA[18], in modo da ottenere anche informazioni sull'heap del programma[19]. Ogni nodo del grafo rappresenta un metodo dell'applicazione; per ogni nodo viene inoltre costruito un control flow graph che modelli il flusso di istruzioni interno al metodo corrispondente. Un'altra particolarità prevista dal framework CAP che è risultata utile in questa fase, è la decisione di affiancare alla costruzione del call graph un'analisi approfondita di ogni nodo per identificare quei collegamenti impliciti tra nodi che in Android sono particolarmente numerosi a causa della presenza della GUI e dei meccanismi di IPC e ICC. Questo procedimento è realizzato mediante l'implementazione di un listener che viene chiamato ogni qual volta la costruzione del grafo preveda l'analisi di un nuovo nodo e che ha il compito di tener traccia dei nodi contenenti istruzioni target, registrazioni programmatiche di callback, registrazioni di Broadcast Receiver ed istruzioni legate a meccanismi di ICC e all'utilizzo di Fragment.

[18] `http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis`

[19] Con il termine heap si denota la parte della memoria principale allocata dinamicamente durante l'esecuzione del programma. Per maggiori informazioni a riguardo si rimanda alla spiegazione approfondita fornita da Oracle sul sito `https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html`

5. Targeted symbolic execution: il grafo ottenuto viene utilizzato per ricercare i percorsi possibili dagli entry point ai nodi che contengono metodi target attraverso una ricerca di tipo Depth-First Search (DFS)[20]. I nodi dei percorsi trovati possono essere analizzati grazie ai cfg costruiti precedentemente al fine di ricavare i vincoli del path. Questa fase si occupa anche di recuperare dati aggiuntivi riguardanti gli elementi di layout, tenendo traccia, per ogni percorso, delle informazioni sugli elementi ad esso legati, ovvero id, tipo di elemento, componente di appartenenza, tipo di dato estratto (ad esempio boolean nel caso venga utilizzato lo stato di una CheckBox mediante la funzione isChecked()).

6. Backwards path dependencies: in maniera conforme a quanto descritto in Sezione 5.2.2, grazie alle informazioni aggiuntive estratte al punto 4, è effettuata una ulteriore analisi iterativa atta a scoprire se esistano dipendenze non esplicite e quindi non riscontrabili mediante il solo utilizzo del grafo. Ogni nuovo nodo aggiunto in testa al path viene analizzato grazie al relativo control flow graph fornito da Wala in modo da comprendere quali istruzioni interne al nodo debbano essere eseguite al fine di raggiungere quella che permette il collegamento implicito. I nuovi percorsi trovati sono considerati *infeasible* se il primo nodo della catena non è un entry point. Questa fase dell'analisi è particolarmente complessa in quanto non è sufficiente considerare la sola istruzione in questione. Per chiarire questo punto, si può fare l'esempio del lancio di un'activity a partire da un'altra activity: si supponga di aver trovato un path che abbia come primo nodo uno dei metodo di lifecycle di un'activity chiamata MyActivity e di aver aggiunto in testa al percorso un nodo che rappresenta il metodo di un'altra activity, in quanto il punto 4 ha riscontrato la presenza di un collegamento implicito tra le due. Si supponga inoltre che sia stata utilizzata la funzione startActivity(Intent intent) per lanciare MyActivity a partire dal nuovo nodo; tale metodo lancia una nuova activity sulla base delle informazioni contenute nell'Intent, che definiscono non solo il componente da lanciare, ma anche i dati da trasmettere. È chiaro che il raggiungimento di questa istruzione non comporta necessariamente che il nuovo componente lanciato sia MyActivity, in quanto il nuovo nodo potrebbe lanciare componenti

[20] `https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/`

diversi a seconda di determinate condizioni (e.g. il valore di un EditText, la scelta di un RadioButton) che devono quindi essere prese in considerazione al fine di ottenere in fase dinamica una stimolazione corretta del percorso.

7. Guided & targeted symbolic execution: come ultimo passo, ogni nodo dei nuovi path è analizzato nel dettaglio al fine di ottenere i constraint corrispondenti, tenendo in considerazione non solo le istruzioni target, ma anche quanto estratto al punto 6. Anche in questo caso sono collezionate le informazioni relative agli elementi di layout come accade nella fase di targeted symbolic execution descritta al punto 5.

L'output di questa fase è composto dalle catene di eventi che portano al raggiungimento delle istruzioni target, dai constraint che devono essere soddisfatti al fine della corretta esecuzione dei path e dalle informazioni legate all'interfaccia utente indispensabili per il proseguimento dell'analisi.

6.1.3 Constraints Solving

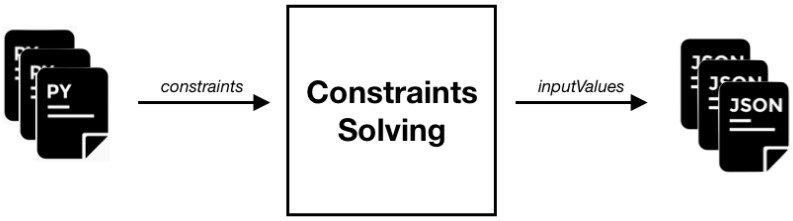

Figura 12: Modulo di risoluzione dei vincoli

In uscita dal modulo di analisi statica si hanno a disposizione le informazioni essenziali ai fini della prosecuzione del flusso di analisi; il modulo descritto in questa sezione si occupa della risoluzione dei vincoli derivanti dall'esecuzione simbolica mediante l'utilizzo dell'SMT solver Z3 (presentato in Sezione 2.4). Gli elementi della GUI e i vari percorsi trovati verranno processati nella successiva fase di parsing, che verrà dicussa in Sezione 6.1.4.

In ingresso al blocco considerato sono presenti una serie di file che descrivono i vincoli, ognuno dei quali si riferisce ad un determinato percorso di stimolazione.

Nello specifico, ogni file contiene la dichiarazione di tutte le variabili di un determinato percorso sotto forma di *simboli*, a cui sarà necessario attribuire un valore, e i constraint rappresentati come un'unica espressione logica (ogni vincolo presente nel path viene legato agli altri in AND o OR) da passare come parametro di entrata ai metodi di Z3. Il blocco sostanzialmente esegue uno script Python che sfrutta i pacchetti di Z3 per risolvere tutti i constraint che gli vengono passati in input. Ogni file in ingresso viene analizzato in maniera separata: se i vincoli sono risolvibili, il solver genera un insieme di valori da attribuire ai simboli affinché tali constraint siano soddisfatti e il blocco in analisi crea un nuovo file in cui ogni simbolo è mappato insieme al relativo valore. Nel caso in cui il solver non sia in grado di trovare valori che possano soddisfare i vincoli, il file di output che verrà generato segnalerà l'impossibilità di trovare un risultato; in questo caso il percorso corrispondente a quel file sarà ritenuto *infeasible* ancora prima di essere eseguito a runtime.

6.1.4 AVCParser

La fase di parsing e generazione di test case è la penultima del processo. Il suo compito è quello di generare una serie di script Python, che siano in grado di stimolare in maniera completamente automatizzata ogni percorso individuato durante la fase statica.

I dati disponibili a questo stadio del processo consistono nelle informazioni relative ai percorsi da intraprendere per arrivare al raggiungimento dei target, ai vari elementi della GUI legati al percorso (questi due tipi di informazioni si presentano in maniera aggregata sotto forma di tanti file JSON quanti sono i percorsi trovati, come descritto in Sezione 6.1.2), e ai valori reali soddisfacenti i vincoli necessari all'esecuzione dei path (calcolati precedentemente come esposto in Sezione 6.1.3).

Il componente implementato per condurre tale fase è chiamato **AVCParser**; esso si serve del tool Python AndroidViewClient (introdotto in Sezione 2.5), e prevede fondamentalmente l'esecuzione di due passi, come si nota anche in Fig. 13.

Il primo stadio del processo, consiste nel parsing dei file JSON in ingresso e l'inserimento del contenuto in una struttura dati organizzata e compatta; tale struttura è progettata come segue: contiene un riferimento al numero del percorso da stimolare e, per ogni percorso, i nodi (che rappresentano i metodi dell'applicazione) da eseguire identificati univocamente e l'ordine di esecuzione; per ogni nodo si ha

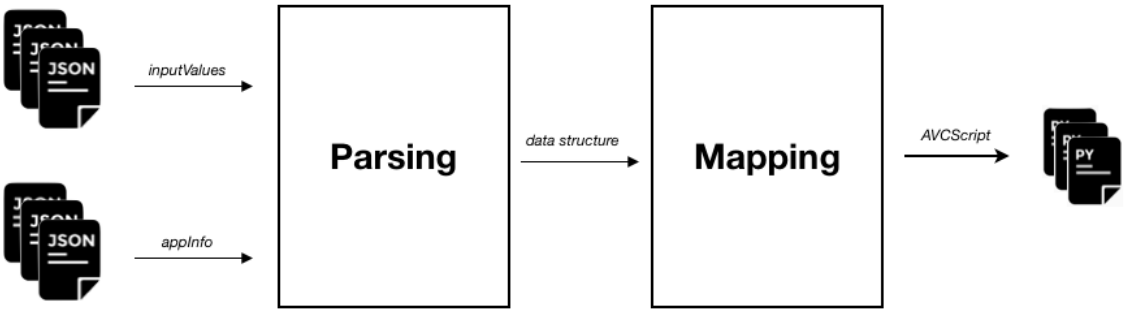

Figura 13: AVCParser

traccia del nome del metodo, della classe in cui è contenuto, del tipo di evento che deve verificarsi in modo che esso sia eseguito (al fine di comprendere in fase dinamica quale tipo di azione compiere effettivamente sull'applicazione) e dell'id dell'elemento di layout che deve essere stimolato affinché tale evento si verifichi (come ad esempio un bottone per le onClick). La struttura dati possiede inoltre, per ogni percorso, una lista di tutti gli elementi di layout che vengono utilizzati all'interno dei nodi della catena: per ogni elemento è presente il relativo id, il nome, la tipologia di view di appartenenza e, nel caso in cui tale elemento influisca su uno dei vincoli del percorso (ad esempio nel caso in cui la selezione di una checkbox sia condizione necessaria per il raggiungimento del target), il nome della variabile che gli era stata assegnata in fase di utilizzo di Z3 e il valore che gli deve essere attribuito in fase di esecuzione. Il secondo stadio del processo è il *core* effettivo del componente: si occupa di analizzare in sequenza la struttura dati creata, utilizzando le informazioni in essa contenute al fine di produrre in output uno script per ogni percorso da intraprendere; tali script rappresentano i test case da passare in ingresso alla fase dinamica e contengono istruzioni che sfruttano AndroidViewClient per collegarsi all'emulatore attraverso l'uso di adb (cfr. Sezione 2.1.3) ed eseguire le stimolazioni opportune. Come già evidenziato in Sezione 5.2.4, per poter automatizzare l'intera creazione degli script, è stato necessario studiare l'insieme dei possibili eventi d'interazione utente ed effettuare una mappatura che collegasse ogni evento alle istruzioni da eseguire per stimolarlo: questo processo si è basato su alcuni assunti che si sono resi necessari a causa della complessità del problema e che verranno discussi in Sezione 6.2. L'algoritmo di creazione ideato consiste, nello specifico, nell'assegnare ad ogni nodo della catena le istruzioni opportune che devono essere eseguite affinché

tale evento sia stimolato correttamente (con i giusti input e al momento giusto): tali istruzioni devono essere scelte in base alla tipologia di evento che deve verificarsi (informazione contenuta nella struttura dati) in accordo con il *mapping* degli eventi precedentemente effettuato. Le sezioni iniziali e finali di ogni script generato in uscita dall'AVCParser risultano fisse, ovvero si occupano di connettersi al dispositivo, installare l'applicazione in esame su di esso, lanciare l'activity principale dell'app e disinstallare l'applicazione dal dispositivo una volta terminata la stimolazione. La parte centrale varia invece a seconda dell'APK analizzato e comprende la chiamata vera e propria dei metodi di AndroidViewClient.

Quest'ultima parte della sezione è volta a spiegare nel dettaglio quali scelte siano state prese in riferimento all'ordine di stimolazione degli eventi.

I vincoli estratti durante la symbolic execution hanno come variabili i dati ottenibili in seguito alla stimolazione degli elementi di layout dell'interfaccia. Tenendo presente il meccanismo di ICC (cfr. Sezione 3), ognuno di questi dati può essere utilizzato all'interno del componente a cui appartiene, oppure all'interno di altri, qualora questi dati vengano trasferiti attraverso gli Intent. In fase statica si prendono in considerazione tutti i possibili utilizzi del dato, a prescindere da cosa l'abbia generato, in modo da costruire vincoli completi essenziali alla determinazione di un valore corretto da inserire a runtime; in fase dinamica però, è necessario che sia rimasta traccia del componente in cui questo dato è stato effettivamente originato, in quanto il valore dovrà essere impostato non solo sull'elemento corretto, ma anche al momento opportuno, ovvero quando il componente di appartenenza è in primo piano. Come conseguenza di questa considerazione, si è ritenuto opportuno differenziare tra stimolazioni atte ad impostare correttamente gli elementi di layout da cui dipendano dei vincoli, e quelle necessarie per *triggerare* le callback. Siccome la chiamata di una callback comporta l'esecuzione di un metodo che potrebbe potenzialmente servirsi dei dati derivanti dall'utilizzo dell'interfaccia utente o lanciare un nuovo componente, l'ordine di esecuzione degli stimoli per ogni componente deve prevedere necessariamente che sia prima effettuata l'interazione con tutti gli elementi di layout, e successivamente quella che attiva l'eventuale callback.

In secondo luogo, è stato necessario tenere presenti anche tutti quegli elementi di layout incontrati durante il percorso, il cui valore non influisce direttamente sull'esecuzione di esso, poiché non rilevanti al fine del soddisfacimento dei constraint. Il fatto che tali elementi siano trascurabili per l'analisi non esclude però che possano

essere utilizzati dall'applicazione, pertanto non eseguire alcuna interazione nei confronti di questi, potrebbe in certi casi portare alla non corretta esecuzione dell'app o addirittura al crash (ad esempio se il valore di un EditText rimane a null potrebbe essere generata un'eccezione se lo sviluppatore non ha effettuato una corretta validazione dell'input): di conseguenza è stato deciso di assegnare a tutti questi elementi un valore di default e di stimolarli come fatto per gli elementi dei vincoli.

6.1.5 Test Cases Injection

La quinta ed ultima fase del sistema ha lo scopo di compiere l'analisi dinamica dell'applicazione. Nello specifico, questo blocco effettua l'*injection* in ambiente emulato dei test case originati dalla precedente fase al fine di eseguire i percorsi che essi definiscono, arrivando quindi a stimolare la specifica istruzione target.

L'input di questo modulo consiste in una serie di test case sotto forma di script, in grado di esplorare particolari percorsi di esecuzione per il raggiungimento delle chiamate API all'interno dell'applicazione, che potrebbero costituire una minaccia per l'utente. Questi file sono contenuti in una directory all'interno della cartella preprocessata dell'APK in esame, con il nome *AVCScript_pathX.X.py*, dove le *X* indicano l'identificativo del path che si sta stimolando.

Il modulo implementato si serve di un emulatore: come già anticipato in Sezione 2.1.2 il prototipo sfrutta il Nexus 5 messo a disposizione dall'SDK di Android. Per avviare l'emulatore è necessario eseguire la seguente linea di comando da terminale:

```
emulator -avd Nexus_5_API_21_x86
```

Inoltre, prima di avviare l'analisi dinamica, è necessario che adb sia in funzione per creare un collegamento al device. Sempre da terminale, il comando da eseguire sarà:

```
adb shell
```

Per eseguire uno degli script in ingresso, ad esempio *AVCScript_path0.0.py*, sarà necessario spostarsi nella directory contenente lo script e lanciare da terminale il seguente comando:

```
/usr/bin/pythonw AVCScript_path0.0.py
```

Si potrà quindi osservare l'installazione dell'applicazione all'interno dell'emulatore, l'avvio di essa e le interazioni eseguite mediante l'uso di AndroidViewClient in maniera completamente automatica. Una volta terminata la stimolazione, l'app sarà disinstallata.

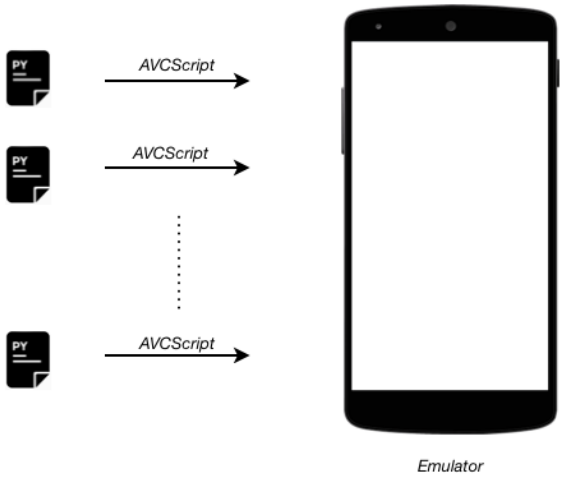

Figura 14: Modulo di analisi dinamica

Come descritto in Sezione 5.2.5, il risultato di questa fase può essere sfruttato da un sistema di *monitoring* aggiuntivo per verificare che l'utilizzo dell'API non costituisca una reale minaccia, scelto in base al tipo di chiamate API scelte come target (tale strumento non è previsto in fase progettuale, in quanto non costituisce una parte integrante dell'architettura, bensì un supporto ad essa): ad esempio nel caso in cui l'API target sia il metodo openConnection(), si può verificare che l'applicazione si stia collegando ad Internet mediante l'utilizzo di protocolli di connessione sicura attraverso strumenti che monitorano il traffico di rete (e.g. CharlesProxy[21], Wireshark[22]).

6.2 Limitazioni

Questa sezione ha lo scopo di effettuare delle riflessioni che rendano evidenti le limitazioni di CAPDroid, dovute alle scelte prese nel corso dell'implementazione del prototipo.

[21] `https://www.charlesproxy.com`
[22] `https://www.wireshark.org`

Static Analysis

Per quanto riguarda il modulo di analisi statica, una prima osservazione è legata ai componenti ed elementi di layout presi in considerazione. Per garantire un'analisi il più completa possibile ed applicabile a casi reali, il modulo gestisce tutti i componenti di Android (Activity, Service, Broadcast Receiver, Content Provider) e i Fragment; per quanto riguarda gli elementi di layout invece, a causa del numero molto elevato di tipologie e varianti e dello studio approfondito necessario per comprendere come trattare ognuno di essi, si è deciso di considerarne un sotto insieme di base per la versione attuale del tool. Questa scelta influenza chiaramente la ricerca di path e la conseguente generazione di vincoli: nello specifico, nel caso in cui l'applicazione in esame faccia uso di elementi di layout non trattati dal prototipo, non è assicurato che l'insieme dei path generati e degli annessi vincoli sia completo.

Un'altra considerazione necessaria riguarda una limitazione a cui è in generale soggetta la symbolic execution, ovvero la problematica della *path explosion*: il numero di path feasible di un programma cresce esponenzialmente in maniera proporzionale alla grandezza del programma (o meglio al numero di nodi del grafo, degli archi e dei branch presenti in ogni nodo) e può diventare infinito in caso di presenza di *unbounded loop* (come descritto in [30] e [31]). Sulla base delle informazioni raccolte nella fase di ricerca, l'implementazione del modulo statico mette in atto alcune procedure che mitigano in parte questo problema. Nello specifico, in fase di targeted symbolic execution, il fatto di limitare la ricerca dei percorsi ai soli che portino ad istruzioni target, diminuisce lo spazio dei path: in presenza di un branch il percorso si biforca solo se entrambi i rami possono effettivamente portare all'esecuzione del target; in caso contrario ne viene considerato solo uno e il percorso resta singolo. In fase di guided & targeted symbolic execution, il miglioramento è ancora maggiore in quanto non solo l'analisi è mirata a raggiungere determinati obiettivi, ma è guidata dalle informazioni estratte nella fase di riconoscimento di *backwards dependencies* descritta precedentemente, diminuendo ulteriormente il numero di path possibili. Infine in entrambe le situazioni sono presi i seguenti accorgimenti: in caso di presenza di loop le iterazioni sono limitate ad un massimo di 5; sono valutate le condizioni di branch costanti durante l'analisi, in modo da eliminare a priori path *infeasible* a causa di condizioni mai verificate. Non sono invece attualmente applicate metodologie atte a mitigare le problematiche derivanti dalla presenza di funzioni ricorsive

all'interno dell'applicazione analizzata.

Constraints Solving

La risoluzione dei vincoli è una fase essenziale per l'utilizzo di tecniche di analisi come l'esecuzione simbolica (cfr. Sezioni 4.2.1 e 5.2.3). Per l'implementazione di questo prototipo, come già detto in Sezione 6.1.3, la scelta è stata quella di servirsi delle librerie fornite da Z3 [32].

Come per tutti gli altri moduli del prototipo, è stato necessario decidere in fase di progettazione del tool quali fossero gli aspetti principali da tenere in considerazione per lo sviluppo di questa prima versione. Al momento, il modulo di Constraints Solving si occupa di gestire e risolvere una parte di tipologie di vincoli, ovvero quelle riguardanti i tipi primitivi di dato. Tale limitazione è stata considerata una semplificazione accettabile al fine di raggiungere l'intento della tesi, che ha come obiettivo di base quello di proporre una nuova metodologia e mostrare la validità dell'approccio formulato durante l'elaborato mediante l'implementazione del prototipo.

AVCParser

In riferimento all'AVCParser, una prima importante considerazione riguarda l'insieme di eventi mappati in questa fase. Il percorso che si ha a disposizione in seguito all'analisi statica è una sorta di flusso completo dei metodi esatti che vengono invocati all'interno dell'applicazione per arrivare al raggiungimento della chiamata API cercata; tra questi metodi, quelli che necessitano del verificarsi di un evento per essere eseguiti sono i seguenti: i metodi di lifecycle dei componenti, che possono essere chiamati sia in seguito ad eventi di sistema, sia a causa di determinate sequenze di azioni effettuabili da un utente; le callback di eventi di sistema, ad esempio cambi di stato di memoria o batteria; le callback *triggerate* da eventi esterni, ad esempio ricezione di SMS; i metodi degli event listener registrati sulle view del layout (cfr Sezione 3.5). Il notevole numero di eventi e combinazioni di essi che possono scaturire a partire dalle azioni di un utente, da fattori esterni o dalle condizioni del sistema, rappresenta il problema principale riscontrato in fase di *mapping* degli eventi. Inoltre per ognuno degli eventi possibili presi in considerazione, è necessario condurre uno studio del comportamento di Android per comprendere quali siano le esatte azioni che vengono eseguite in seguito al verificarsi di tale evento, processo

non banale in quanto la conseguenza di un evento può dipendere anche dalla specifica applicazione o dallo stato della stessa. Per queste motivazioni, per la prima versione del prototipo, è stato deciso di considerare un sottoinsieme dei possibili eventi, che potrà in futuro essere ampliato: nello specifico vengono creati script per quei percorsi che coinvolgano eventi strettamente legati all'interazione dell'utente con l'app; di questo numeroso insieme di eventi sono stati poi effettivamente presi in considerazione quelli generati a partire dalle interazioni basilari che possono essere effettuate con un sottoinsieme di base di elementi di layout (e.g. button, checkbox, editext), che sono gli stessi presi in esame in fase statica.

Una seconda osservazione è legata al punto di inizio di ogni stimolazione: ogni script inizia l'esecuzione dell'app a partire dall'activity launcher per rendere la stimolazione quanto più simile alla realtà. In generale infatti, a meno che un'applicazione non sia in background poiché già stata utilizzata dall'utente, al momento di avvio la prima schermata ad essere caricata dal sistema è l'activity principale. Fanno eccezione i componenti dichiarati esportabili nell'Android Manifest: un'applicazione esterna potrebbe lanciare uno di questi e in tal caso il sistema caricherebbe direttamente il componente in questione. Questa semplificazione, sebbene possa essere abbandonata in sviluppi futuri, non può essere considerata una vera e propria limitazione in quanto deriva dal fatto di considerare i soli eventi generabili a partire dall'interazione con l'utente.

Test Cases Injection

Le limitazioni in questa fase del processo sono legate al tipo di emulatore che viene utilizzato; nello specifico, il Nexus 5 fornito dall'SDK di Android garantisce il supporto fino alle API 21. Applicazioni che sfruttino una versione più recente di API andrebbero testate per verificarne i risultati.

6.3 Use Case

Per testare il prototipo appena descritto, esplicitarne le effettive capacità e mostrare la validità dell'approccio proposto, è stata sviluppata appositamente un'applicazione di test. Lo scopo di questo esperimento è mostrare come l'analisi effettuata da CAPDroid sia in grado di rilevare una serie di comportamenti sospetti all'interno dell'applicazione e di stimolarla in maniera automatica in modo da eseguire ognuno

di questi, permettendo così di valutarne la pericolosità. Questa sezione presenta il particolare caso d'uso scelto, basato su un'applicazione costruita ad hoc in modo tale che fosse possibile mostrare le principali feature del tool all'interno di uno scenario verosimile.

Tale applicazione è concettualmente simile a quella che potrebbe essere l'applicazione di una banca: l'utente che interagisce con l'app è un membro della banca che sfrutta spesso questa applicazione per accedere al proprio conto e usufruire dei servizi messi a disposizione. Chiaramente, tale app presenta alcune vulnerabilità di cui sono ignari sia la banca, sia gli utenti. La struttura generale è la seguente: avviando l'app, l'utente accede alla propria area personale; in questa schermata sono presenti il logo della banca e una serie di azioni che possono essere eseguite dall'utente per accedere alle altre schermate dell'applicazione. Dentro il codice sono state inserite alcune API Android classificabili come sospette: in particolare API riguardanti la connessione al web e i servizi di localizzazione. Nel seguito sono elencati i percorsi sospetti presenti all'interno dell'applicazione con annessi risultati evidenziati dall'analisi condotta sulla stessa:

- Attivazione di un servizio in background da parte dell'applicazione che monitora di continuo la posizione dell'utente. La fase statica è stata in grado di individuare tutte le stimolazioni da eseguire per arrivare all'attivazione di tale servizio; una volta eseguita la fase dinamica, si è notato che effettivamente tale servizio viene attivato e l'applicazione si serve del GPS. Questo comportamento è stato ritenuto anomalo in quanto l'applicazione in possesso dei permessi per i servizi di localizzazione, li sfrutta per mantenere costantemente sotto

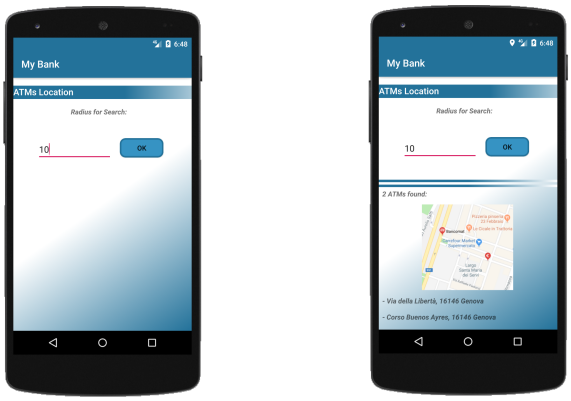

Figura 15: Schermate ricerca e visualizzazione ATM

controllo la posizione dell'utente. Il servizio viene attivato in seguito all'utilizzo della ricerca degli ATM presenti in zona, come mostrato in Fig. 15, ma anziché sfruttare la posizione solamente per questa operazione, l'app traccia il dispositivo.

- Visualizzazione delle transazioni effettuate. In fase statica vengono individuati più percorsi da intraprendere per visualizzare lo storico dei propri movimenti bancari; il motivo di questo comportamento risiede nel fatto che viene concessa all'utente la visualizzazione dello storico sia per le transazioni in entrata, sia per quelle in uscita, e viene inoltre permesso di inserire il numero di mesi precedenti per il quale si intende visionare i movimenti. Una volta effettuate queste scelte, verrà mostrata la schermata con le relative transazioni desiderate, recuperate dal sito della banca attraverso le richieste opportune (sfruttando quindi una connessione alla rete). Le API di connessione per l'esecuzione di tali azioni sono i target ricercati, e per ognuna di queste chiamate (raggiungibili in relazione alle scelte dell'utente), viene individuato un diverso percorso con relativi vincoli sugli input. In questo particolare caso sono stati individuati tre percorsi diversi: due di questi sfruttano una connessione di tipo HTTPS, mentre il terzo utilizza HTTP. Tutti e tre i percorsi sono stati stimolati in fase dinamica in modo da poter stabilire con certezza la pericolosità della chiamata. Al termine della stimolazione di questi percorsi, i due che utilizzavano una connessione sicura non sono stati considerati critici, mentre quello che utilizzava HTTP è stato considerato come effettivamente pericoloso. In questo caso CAPDroid viene supportato da un sistema in grado di monitorare il traffico di rete.

Questo semplice test mostra le caratteristiche principali del tool: la capacità di analizzare l'intero codice tramite analisi statica, l'estrazione delle sole informazioni utili, il calcolo dei valori di input corretti per una determinata esecuzione, la traduzione delle informazioni in istruzioni in grado di stimolare l'applicazione e infine l'esecuzione mirata di un percorso in un ambiente dinamico, attraverso il quale è possibile monitorare il reale comportamento dell'app. Lo scenario riguardante le transazioni è stato volutamente costruito in maniera da risultare piuttosto complesso, in modo da poter mostrare le capacità principali di CAPDroid. Nello specifico, il percorso che dalla *MainActivity* permette di raggiungere l'API che si connette

alla rete tramite protocollo non sicuro, prevede una cospicua serie di interazioni con l'interfaccia; in ordine (come mostrato dagli screenshot forniti in Fig. 16), le azioni automaticamente eseguite dallo script sono state le seguenti:

- Installazione dell'applicazione sul device e lancio della MainActivity.
- Selezione del bottone corretto per essere indirizzati nella schermata che permette di decidere quali movimenti visualizzare.
- Una volta selezionato il bottone, un *AlertDialog* compare richiedendo all'utente di confermare di essere l'effettivo proprietario dell'account corrente; per poter proseguire nel percorso è stato necessario premere il pulsante corretto di tale Dialog.
- Una volta indirizzati nella schermata delle transazioni, per arrivare alla chiamata HTTP è stato necessario inserire i corretti valori riguardanti i vincoli del percorso: nello specifico tale chiamata necessitava del corretto *radioButton* selezionato e dell'inserimento di un valore superiore a tre all'interno dell'*EditText* presente nella schermata; tali valori sono stati calcolati dalla fase di *constraints solving*.
- Infine, dopo aver impostato correttamente tutti gli elementi di layout presenti, è stato premuto il bottone per far partire la richiesta HTTP, e sono state dunque visualizzate le transazioni richieste.
- Tale stimolazione ha permesso attraverso l'uso di WireShark di monitorare che la richiesta effettuata dall'applicazione fosse effettivamente di tipo non sicuro.

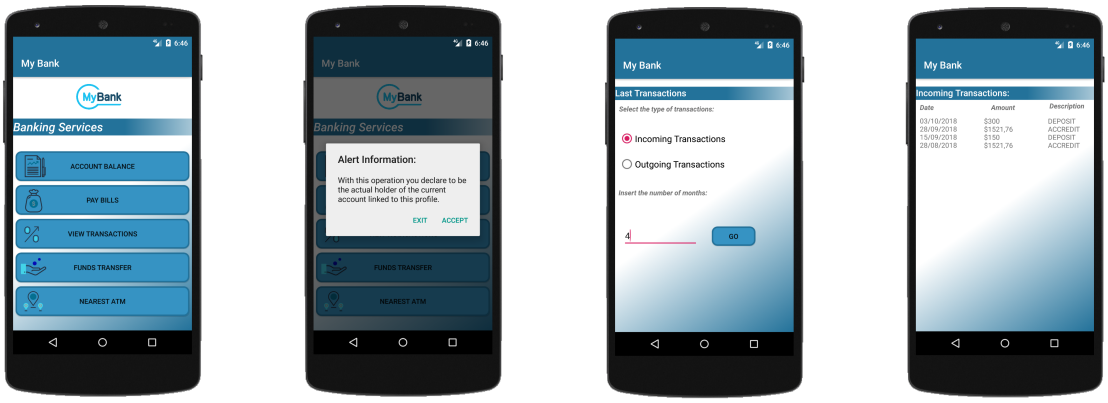

Figura 16: Sequenza stimolazioni percorso HTTP

7 Risultati Sperimentali

In questa sezione verrano presentati i risultati sperimentali ottenuti a seguito di una fase di testing eseguita su un dataset di applicazioni ufficiali provenienti dal Google Play Store, scelte a partire da quelle elencate nella classifica di Androidrank[23] tra Giugno e Agosto 2018.

Lo scopo di questa fase è quello di valutare l'efficacia del prototipo per Android presentato in Sezione 6 direttamente su applicazioni che vengono utilizzate ogni giorno dagli utenti Android.

La struttura modulare del tool permette innanzitutto di poter raccogliere dati per ogni fase distinta del processo di analisi; in secondo luogo, attraverso questo testing, è possibile effettuare delle considerazioni sui diversi risultati ottenuti, in modo da capire quali siano i blocchi meno robusti dello strumento, per quali motivazioni e in che modo si possano migliorare. Tali esperimenti hanno quindi lo scopo non solo di mostrare la validità del prototipo implementato, ma risultano anche essenziali a decidere quali migliorie si potranno apportare in futuro a CAPDroid; eventuali modifiche a singoli blocchi potranno essere facilmente inserite nel contesto dello strumento, in quanto la divisione in moduli del sistema permette la possibilità di sostituire le singole parti di esso facendo solamente attenzione agli input e agli output della determinata fase.

7.1 Contesto di Analisi

In questa sezione sono riportate informazioni necessarie a comprendere il contesto in cui è stata effettuato il testing del prototipo, ovvero i moduli analizzati, l'ambiente scelto, il dataset utilizzato e le API fornite come target.

Poiché il prototipo, durante la fase implementativa, è stato testato solamente su applicazioni costruite ad hoc, prima di eseguire il testing vero e proprio è stata condotta una fase di *debugging*, descritta in Sezione 7.2, in modo tale da poter eseguire la campagna di analisi avendo risolto il maggior numero possibile di problematiche derivanti dall'analisi di applicazioni reali, che possono avere un grado di complessità differente e non noto a priori.

[23] `https://www.androidrank.org/listcategory`

Come risultato secondario di tale fase è stato anche possibile prendere decisioni in merito allo svolgimento del successivo testing.

Come sarà spiegato in maniera più esaustiva nel seguito del capitolo, il *debugging* ha anche evidenziato come il modulo di *constraints solving* non sia ancora in grado, allo stadio attuale, di generare tutti quei valori essenziali per il successivo blocco. Questa mancanza di informazioni penalizza le fasi successive che, in assenza degli input corretti per la generazione dei test case che descrivono i percorsi, non possono a loro volta proseguire il processo di analisi.

Sulla base di queste osservazioni, i dati sperimentali mostrati in questo elaborato comprendono i soli output delle prime due fasi dell'architettura di CAPDroid, per poter effettuare considerazioni più rilevanti in merito alle analisi condotte.

Per quanto riguarda il modulo di analisi statica, tale blocco è stato implementato prestando attenzione all'*exception handling* per cercare di ridurre al minimo la possibilità di terminazione non corretta della sua esecuzione a causa di errori di implementazione; è necessario però sottolineare che la presenza di problematiche non ancora scoperte o irrisolte, potrebbe comunque causare una terminazione non regolare del programma, che può essere rilevata grazie al meccanismo di log inserito.

È fondamentale sottolineare che il testing effettuato, in termini di dati ottenuti, evidenzia un importante risultato: nonostante la fase di risoluzione di constraint limiti il funzionamento del tool nel suo insieme, il modulo di analisi statica di CAPDroid è in grado di ottenere, anche nel caso di applicazioni reali, le informazioni necessarie per quanto riguarda percorsi e vincoli di ogni genere (compresi quelli relativi a stringhe ed oggetti più complessi).

7.1.1 Ambiente

La fase di testing è stata eseguita su un portatile MacBook Pro con sistema operativo MacOS Sierra, CPU Intel Core i5 2,4GHz e 4 GB di memoria RAM.

È stato deciso di utilizzare questo ambiente per condurre i test in quanto Dare, uno dei due tool di cui fa uso la fase di preprocessing presentato in Sezione 2, può essere adoperato solo su sitemi Unix-like. La parte di analisi statica può essere eseguita anche su Windows, ma per mantenere uniformità riguardante le performance di analisi, è stata condotta anch'essa su MacOS.

È stato possibile ottenere una misura della differenza in termini di tempistiche per quanto riguarda l'analisi statica attraverso un portatile Asus con sistema operativo Windows 10, CPU Intel Core i7 3,1 GHz e 8 GB di memoria RAM; il portatile con hardware più potente impiega all'incirca l'80% di tempo in meno nell'eseguire quest'analisi. I dati forniti nelle sezioni seguenti saranno comunque riferiti all'ambiente che ospita MacOS.

7.1.2 Dataset

Come già anticipato all'inizio della Sezione 7, per produrre risultati significativi riguardanti la validità del metodo di analisi, è stato utilizzato un dataset di applicazioni scaricate in modo causale sulla base delle app presenti nella classifica proposta da Androidrank; tale insieme è formato nello specifico da 600 applicazioni, ognuna delle quali viene identificata univocamente attraverso il proprio hash MD5.

La motivazione della scelta di testare il prototipo su un insieme di applicazioni scaricate direttamente dallo store ufficiale di Android, risiede nel fatto di voler mostrare la reale utilità del tool: CAPDroid permette infatti di verificare la presenza di vulnerabilità o possibili comportamenti malevoli non solo se applicato a casi d'uso costruiti ad hoc, ma anche in seguito all'analisi di app reali di comune utilizzo, di cui si servono giornalmente milioni di utenti. È quindi importante sottolineare che, per quanto riguarda questa fase di testing, le applicazioni in esame non sono necessariamente malware, ma al contrario dovrebbero essere in linea teorica applicazioni benigne, in quanto scaricate dallo store ufficiale.

7.1.3 API

Una fase essenziale per quanto riguarda l'esecuzione dell'analisi è senza dubbio la scelta delle API che si intendono monitorare. Come già spiegato in Sezione 6.1.2, i risultati ottenibili dalla fase di analisi statica sono strettamente collegati all'insieme di target forniti in input: avendo presente i passi condotti durante questo stadio, è facile concludere che le performance di questo blocco, in termini di risultati, sono influenzate dal tipo e dal numero di API che si stanno ricercando in quanto certe API permettono di tracciare un numero molto più elevato di percorsi rispetto ad altre. Per assicurare una valutazione coerente dei risultati, è stato necessario scegliere, a monte della fase di testing, un'insieme di esse per utilizzarle

come input dell'analisi di tutte le 600 applicazioni in esame. Non avendo a priori una conoscenza delle applicazioni esaminate e della loro tipologia (in quanto, come già detto, selezionate in modo casuale), è stato deciso di utilizzare un insieme di API che comprendesse metodi che fanno riferimento a contesti differenti; nello specifico sono state prese in considerazione alcune tra quelle riguardanti *Location*, *BroadcastReceiver*, *ContentResolver*, *TelephonyManager*, *SMS*, *File I/O*, *Network* e *Reflection*.

7.2 Debugging

In primo luogo è stata prevista una fase di *bug-fixing*, in modo da individuare eventuali errori implementativi sfuggiti durante lo sviluppo, sulla base di 100 app selezionate in modo casuale dal dataset. Questa prima analisi è stata condotta manualmente in modo da verificare in dettaglio il comportamento corretto dello strumento e permettere la correzione del numero più alto possibile di errori. Il medesimo test è stato poi ricondotto sullo stesso campione in seguito alle modifiche effettuate, per poter valutare il miglioramento apportato. L'analisi ha condotto ai risultati esposti nel seguito.

La fase di *preprocessing* è stata eseguita senza alcun errore, e il tempo impiegato per estrarre tutte le risorse necessarie per effettuare l'analisi statica di queste 100 app è durata 150 minuti e 59 secondi, per una media di circa 90 secondi ad applicazione. Tale modulo non è quindi stato modificato, in quanto non sono stati riscontrati problemi di alcun tipo.

Per quanto riguarda il modulo statico, l'analisi condotta ha invece permesso di risolvere problematiche non emerse in fase implementativa e di precedente debugging su applicazioni costruite ad hoc. Il risultato dell'analisi di tale modulo, prima e dopo aver effettuato le correzioni, è visibile in Fig. 17. Come si nota, l'esame dei dati ottenuti ha permesso di suddividere le applicazioni in 4 categorie:

- Analisi Completa: in questo insieme vengono inserite tutte le applicazioni che terminano l'analisi in modo corretto e senza alcun tipo di errore.

- Analisi con Eccezioni: fanno parte di questo set tutte le app che terminano l'analisi in modo corretto, ma generano delle eccezioni che vengono catturate e gestite.
- Failure: indica l'insieme di analisi che terminano in modo non corretto (crash).
- Analisi non Terminata: contiene le analisi di applicazioni che non terminano entro un tempo ragionevole (in questa fase di debugging è stato considerato accettabile un tempo inferiore a 30 minuti. Ulteriori considerazioni a riguardo sono discusse in Sezione 7.3).

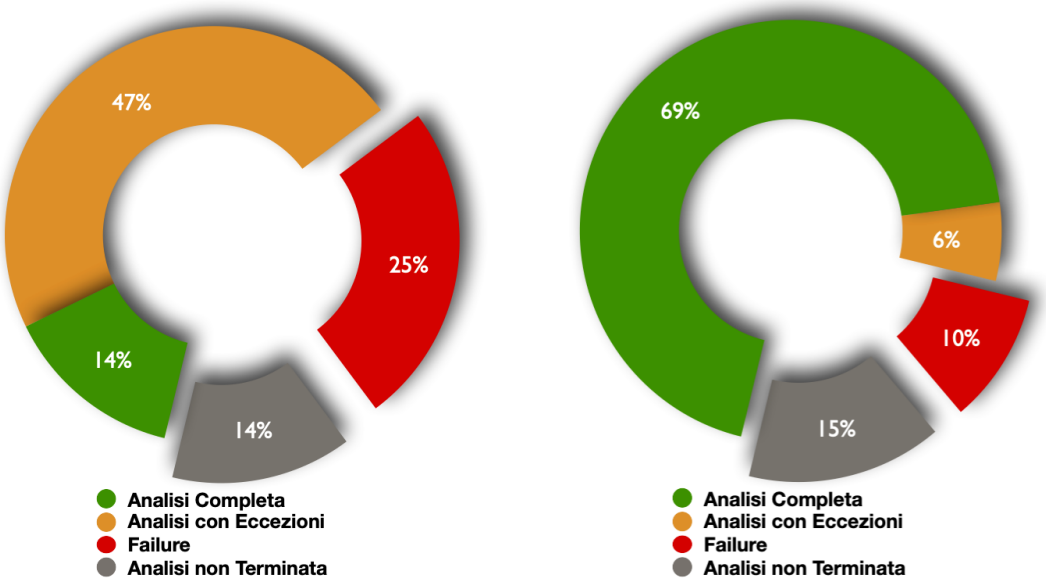

Figura 17: Confronto tra i risultati ottenuti su un campione di 100 app prima e dopo aver effettuato il bug-fixing

Come evidenziano i grafici in Fig. 17, i risultati ottenuti in seguito al bug-fixing sono nettamente migliori rispetto ai precedenti. La correzione degli errori ha consentito di ridurre i casi di failure del 60%: una sola delle applicazioni precedentemente contenute in questo insieme è risultata un nuovo caso di non terminazione dell'analisi, mentre le restanti app terminano in tempi inferiori ai 30 minuti. Per quanto riguarda le applicazioni che terminano l'analisi, quelle che generano eccezioni catturate e gestite sono decisamente diminuite, passando dal 77% al solo 8% del totale.

Dal punto di vista delle tempistiche, in riferimento alle sole analisi che terminano entro 30 minuti, la durata media risulta essere di 98 secondi. Questo valore è stato calcolato sulla base dei dati ottenuti dalle analisi effettuate in seguito alla correzione degli errori.

Come anticipato in Sezione 7.1, il *debugging* è servito anche a valutare la validità del modulo di risoluzione dei vincoli. L'esecuzione dell'analisi statica sulla base di questo campione di 100 app ha permesso di individuare una serie di percorsi alle API target (137), per cui sono stati generati i relativi constraint al fine della corretta esecuzione del path. L'insieme di tutti questi vincoli è stato passato in input al modulo di risoluzione, per poter valutare le percentuali di successo e errori ottenibili: il blocco non è stato in grado di fornire un output completo di valori corretti in nessuno dei casi analizzati, in quanto tutti i vincoli relativi ai percorsi contengono al loro interno almeno uno o più elementi che li rendono non risolvibili. Effettuando un'analisi più approfondita, è emerso che le casistiche in cui il modulo fallisce nell'attribuzione dei valori, combaciano sempre con i vincoli che si riferiscono ad oggetti, i quali al momento non sono supportati da tale blocco.

7.3 Timeout

Per poter effettuare il testing sulle restanti 500 app del dataset in modo automatizzato è stato necessario inserire un timeout al fine di non incorrere nella eventualità di ottenere un test di durata eccessiva (e potenzialmente infinita), data da una possibile non terminazione di una delle analisi. Questa esigenza è risultata evidente grazie ai risultati ottenuti in fase di debugging di CAPDroid, da cui emerge che il 15% del campione scelto non può essere analizzato in un tempo inferiore a 30 minuti: i dati ricavati sono stati fondamentali nella scelta di un valore di timeout ragionevole entro cui terminare forzatamente l'analisi di un'app per proseguire con la successiva. Tale valore è stato fissato a 10 minuti in quanto i valori di durata massima riscontrabili per le analisi delle 75 app che terminano, si aggirano tra i 400 e i 500 secondi.

Per verificare la bontà di questa scelta, si è deciso di condurre innanzitutto un ulteriore test sulle sole 15 applicazioni di cui non è stata terminata l'analisi, analizzandole nuovamente per un tempo non superiore a 60 minuti: anche in questo caso, l'analisi non è terminata entro il limite imposto. Con l'aggiunta di questo risulta-

to è stato possibile effettuare una stima del tempo totale necessario per effettuare un'analisi complessiva delle 90 applicazioni che non causano crash di CAPDroid, con timeout di 10, 30 e 60 minuti:

- Timeout di 10 minuti: considerando i 98 secondi di media per le 75 analisi che terminano e i 600 secondi che trascorrono per l'analisi di ognuna delle 15 app prima che questa sia terminata forzatamente, si arriva ad un totale di circa 4 ore e 30 minuti di analisi.
- Timeout di 30 minuti: in questo caso, per quanto riguarda le analisi che non terminano, ognuna di esse viene effettuata per 1800 secondi prima di essere interrotta a causa dello scadere del timeout. La stima del tempo totale necessario all'analisi complessiva è in questo caso di 9 ore e 30 minuti.
- Timeout di 60 minuti: per l'ultimo timeout scelto, la stima della durata totale dell'analisi risulta essere di circa 17 ore.

Queste valutazioni confermano la validità della decisione, in quanto l'aumento del valore di timeout peggiora notevolmente il tempo complessivo necessario all'analisi, senza apportare alcun miglioramento al numero di app terminate entro tale limite.

7.4 Testing

In questa seconda fase sono state analizzate le restanti 500 applicazioni del dataset in modo tale da ottenere risultati statistici riguardanti l'analisi di applicazioni reali.

Per quanto riguarda la fase di *preprocessing*, il campione è stato analizzato in 902 minuti e 17 secondi, con una media di circa 108 secondi per ciascuna applicazione; come già aveva evidenziato la fase di debugging, il modulo non ha mai fallito, ottenendo una percentuale di successo del 100%.

In riferimento al modulo di analisi statica, i risultati ottenuti sono mostrati in Fig. 18: su 500 applicazioni analizzate, 380 terminano l'analisi, 72 sono terminate in modo forzato per lo scadere del timeout e 48 causano un failure del sistema. Come si nota, il grafico è praticamente identico a quello generato a partire dal campione di 100 applicazioni post bug-fixing: l'unica differenza è data dal fatto che il numero di

app terminate aumenta dell'1%. L'alta percentuale di analisi terminate, anche nel caso di un campione più grande, mostra la fondamentale importanza della fase di debugging, che ha permesso di risolvere un numero molto alto dei bug del modulo; inoltre, il fatto che tale grafico sia così simile a quello precedente, porta a supporre che la maggior parte di eccezioni e crash potenzialmente verificabili in successive analisi, possa ricadere nelle casistiche già incontrate.

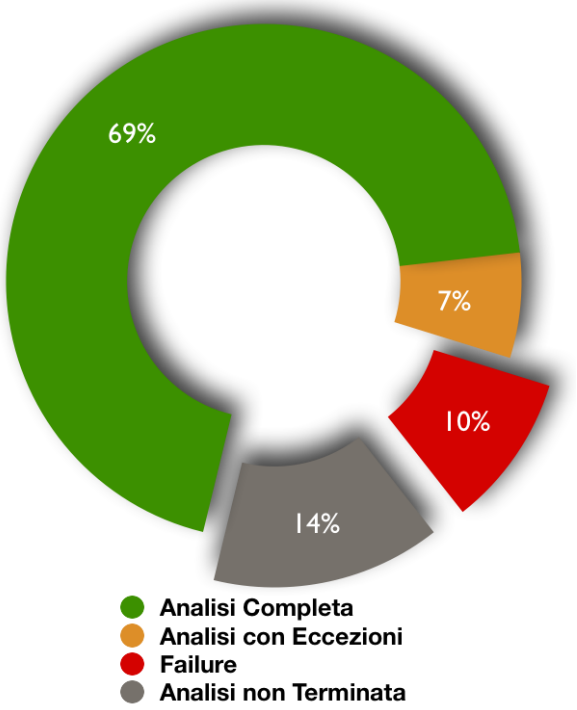

Figura 18: Risultati del testing condotto su un campione di 500 applicazioni

Per quanto riguarda le performance del modulo in termini di tempo, per le sole applicazioni che concludono l'analisi entro il timeout fissato, la durata complessiva dell'analisi è di circa 580 minuti, per una media di circa 1 minuto e 30 secondi per ogni applicazione.

Considerando nuovamente le 380 applicazioni per cui l'analisi termina, il modulo statico di CAPDroid è stato in grado di individuare almeno un percorso per raggiungere almeno uno dei target nel 93.9% dei casi (357 applicazioni su 380). Nello specifico, nel corso di questa campagna di testing sono stati estratti in media circa 90 diversi path per ognuna di queste.

Come ultimo passo, per le 380 applicazioni che terminano, si è deciso di utilizzare i risultati ottenuti in questa fase in modo tale da ottenere informazioni aggiuntive che hanno permesso di estrarre la frequenza di utilizzo di una API, sulla base del numero di app che contengono almeno una chiamata ad essa. Questa operazione è stata condotta per poter avere modo di considerare tali risultati da un'altro punto di vista. Data la natura dell'insieme di target scelto, si è deciso di raggruppare le API in base alle tipologie indicate in Sezione 7.1.3. Il risultato ottenuto è mostrato in Fig. 19.

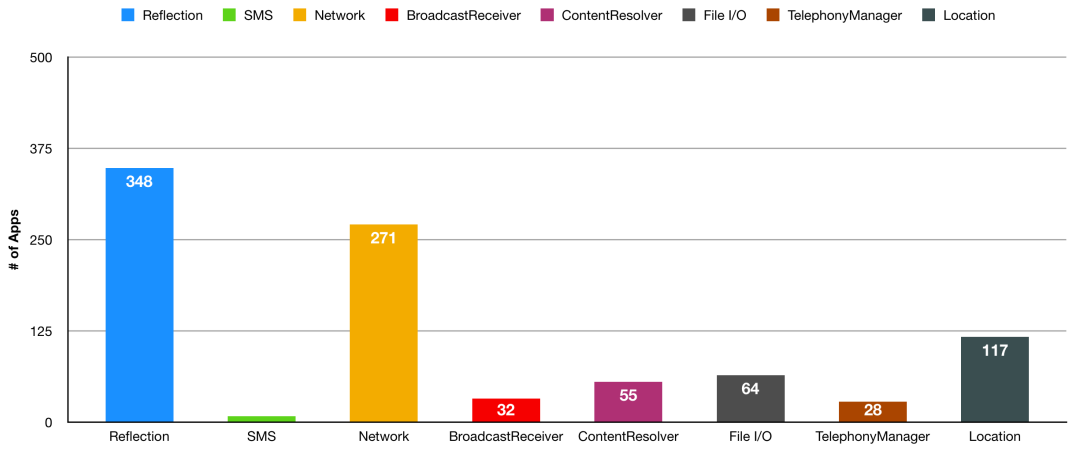

Figura 19: Numero di app contententi API raggruppate per tipologia

Come si può notare vi è una grande differenza in termini di utilizzo: le API di reflection e network sono state rilevate rispettivamente nel 91% e 71% dei casi, mentre quelle relative agli SMS solamente in 8 delle 380 terminate.

7.5 Confronto con Intellidroid

Per concludere questa fase di testing, è stato deciso di effettuare un confronto in termini di risultati ottenibili con IntelliDroid, il tool che più si avvicina alla concezione di CAPDroid.

Inizialmente è stato analizzato lo use case di Sezione 6.3 per capire le differenze tra i due tool in termini di qualità di analisi. Su tale applicazione infatti è stato possibile valutare la correttezza degli output forniti, in quanto si aveva accesso all'intero codice e alla struttura dell'app. Durante questo test, IntelliDroid è stato in grado di

segnalare tutti i percorsi per individuare le chiamate API presenti all'interno dell'applicazione, ma ognuno di questi consiste nel solo nodo contenente il metodo target, ad eccezione del caso di tracking della posizione mediante l'attivazione di un servizio in background, dove il percorso comincia dall'activity che lancia il servizio (che in ogni caso non è il launcher dell'app). Tutti i percorsi trovati hanno generato vincoli, ma alcuni di questi presentano delle incongruenze; mancano inoltre informazioni essenziali riguardanti gli elementi di layout: l'identificativo, la tipologia di elemento, la sua collocazione all'interno del percorso da stimolare, sono tutti dati necessari se si intende fare l'injection di percorsi di questo tipo. IntelliDroid classifica questi path come *UI_interaction* e non è dunque in grado di riprodurre l'esecuzione di tali percorsi a livello dinamico, siccome l'interazione con l'interfaccia grafica non è una delle feature supportate da questo tool (cfr. Sezione 4.4).

Successivamente è stato scelto di analizzare le 100 app della fase di Sezione 7.2, per fare una comparazione in termini quantitativi. Innanzitutto si vuole sottolineare come Intellidroid sia un tool già testato su un insieme di applicazioni reali, ma lo sviluppo di questo prototipo risale al 2016 mentre l'API level delle app analizzate è di due anni più recente. Il fatto che il tool non sia stato più manutenuto dalla sua realizzazione, può aver causato problemi di incompatibilità che possono spiegare il basso numero di analisi portate a termine dal modulo di analisi statica.

Non essendo a conoscenza dell'effettivo contenuto delle applicazioni in esame, l'unico modo per comparare i tool è stato quello di valutare il numero di applicazioni analizzate sulla base delle categorie formalizzate durante il *debugging*; le app sono state analizzate in sequenza, impostando lo stesso timeout di 30 minuti utilizzato per il bug-fixing di CAPDroid. Intellidroid non presenta un meccanismo di log simile a quello implementato per CAPDroid (creato appositamente per la fase di testing per monitorare tutti gli eventuali comportamenti del tool), perciò non è stato possibile analizzare i risultati del processo con la stessa precisione. Da quello che si è potuto verificare, l'analisi di Intellidroid termina entro il timeout in 44 delle 100 applicazioni, mentre la precentuale di completamento di CAPDroid è mostrata in Fig. 17. Sulla presenza di eventuali failure e la possibilità di discriminazione tra analisi concluse con o senza eccezioni non è stato possibile esprimersi per l'assenza dei log.

In conclusione, questa sezione finale del capitolo di risultati sperimentali, è volta

a mostrare il tentativo di testare nella maniera più trasversale possibile il prototipo sviluppato sulla base della metodologia di approccio formalizzata. I risultati ottenuti nell'intera campagna di testing hanno individuato i punti di forza e quelli migliorabili dell'intero sistema e saranno presi in considerazione nel capitolo conclusivo come base per gli eventuali sviluppi futuri di CAPDroid.

8 Conclusioni e Sviluppi Futuri

L'approfondito studio della letteratura scientifica riguardante l'ambito dell'analisi di software ha permesso di concludere che gli approcci classici applicabili al fine di verificare, in modo automatizzato, la presenza di vulnerabilità all'interno di un programma, spesso non garantiscono l'ottenimento di risultati completi e totalmente corretti. In questo elaborato di laurea sono stati quindi studiati e proposti una metodologia di analisi innovativa, detta CAP, e un relativo framework di riferimento, descritti in dettaglio in Sezione 5.

Per valutare l'effettiva efficacia dell'approccio ideato, si è deciso di applicare tale metodo all'ecosistema Android attraverso l'implementazione di un prototipo, CAPDroid (Sezione 6), in grado di analizzare una serie di applicazioni secondo la logica proposta.

Inizialmente, per verificare il reale funzionamento dello strumento, è stata sviluppata un'applicazione (presentata in Sezione 6.3) che simulasse uno scenario tipico di utilizzo di un'app Android; tale applicazione è stata analizzata da CAPDroid ottenendo risultati che si sono dimostrati particolarmente interessanti da un punto di vista qualitativo.

In un secondo momento è stata condotta una fase di debugging su 100 applicazioni reali scaricate dal Google Play Store e successivamente una fase di testing su un campione di 500 app, in modo da poter ottenere un primo insieme di risultati sperimentali. Nonostante il progetto di tesi non avesse come obiettivo primario quello di ottenere un prototipo in grado di analizzare in maniera esaustiva applicazioni di questo tipo, i risultati ottenuti si sono rivelati interessanti: per la maggior parte dell'insieme di app testate, CAPDroid è riuscito a individuare potenziali rischi in fase di analisi statica, come evidenziato in Sezione 7.

L'obiettivo di proporre un nuovo approccio di analisi ibrida è stato quindi raggiunto: i test condotti sul prototipo implementato sulla base della metodologia CAP, nonostante esso sia una prima versione migliorabile in futuro, hanno evidenziato che tale metodo è in grado di arricchire i risultati ottenibili attraverso strumenti della medesima tipologia presenti allo stato dell'arte: un esempio di ciò può essere trovato in Sezione 7.5, dove vengono confrontati i risultati ottenuti da CAPDroid con quelli ricavabili per mezzo di IntelliDroid [25].

Come considerazione finale in merito a questo elaborato, si intende specificare

nuovamente che l'intento di ricerca non è stato quello di stabilire se un determinato software sia classificabile come malware, quanto la possibilità di ricercare aspetti o comportamenti particolari all'interno di un programma, generare dei test case atti a riprodurli ed effettuare un'analisi dinamica sulla base di queste informazioni, per valutare se questi possano realmente essere considerati pericolosi in fase di esecuzione. Considerando il fatto che la caratteristica di configurabilità del framework CAP permette di ricercare l'esatto percorso interno al software per arrivare ad una qualsiasi chiamata presente nel codice che sia specificata in input al sistema, un tool che si basi sulla metodologia proposta può non solo ricercare vulnerabilità, ma anche essere sfruttato per trovare e stimolare qualunque porzione di codice possa essere considerata di interesse da parte dell'utilizzatore.

In conclusione, la metodologia e il framework CAP permettono un'analisi automatizzata di tipo ibrido di software generici. Il tool CAPDroid, sviluppato specificatamente per Android, è ancora in fase prototipale e alcuni moduli della sua architettura presentano limiti dovuti a scelte effettuate in fase di progettazione e implementazione dello stesso, rese necessarie dall'esigenza di ottenere una prima versione funzionante e operativa entro la conclusione di questa tesi. Viste le grandi potenzialità in termini di miglioramento che l'architettura così concepita possiede, la sezione che segue ha lo scopo di elencare i possibili sviluppi futuri che garantirebbero un'evoluzione notevole del tool proposto.

Sviluppi Futuri

In Sezione 6.2 sono state sottolineate diverse limitazioni per quanto riguarda l'implementazione del tool CAPDroid; l'approccio di analisi proposto si è dimostrato valido in linea teorica ed è risultato applicabile anche nella pratica, ma alcune fasi del prototipo possono essere singolarmente migliorate per portare benefici all'intero processo di analisi.

Per quanto riguarda il modulo di static analysis, componente fondamentale del sistema, possibili miglioramenti riguardano da una parte l'ampliamento dell'insieme di elementi di layout presi in considerazione per l'analisi, aggiungendo ad esempio elementi più complessi come picker, listview o spinner, dall'altra l'aggiunta di ulteriori tecniche di mitigazione della path explosion, ad esempio la gestione delle problematiche legate alla presenza di funzioni ricorsive nell'app. Inoltre una possi-

bile evoluzione di tale blocco, consiste nell'utilizzo aggiuntivo di tecniche di Taint Analysis (Sezione 4.2.1) per rendere ancora più solida e precisa l'individuazione di potenziali vulnerabilità: oltre all'individuazione dei data leakage, possibile mediante l'inserimento di questa tipologia di analisi, si potrebbe arrivare non solo a raggiungere e stimolare la chiamata di un qualunque metodo, ma anche a poter indicare i parametri specifici che deve avere il target cercato.

Il blocco di risoluzione dei vincoli rappresenta il modulo maggiormente migliorabile del prototipo, in base anche a quanto evidenziato in Sezione 7.2. Un possibile raffinamento per questa fase del sistema consisterebbe nello studio atto ad ampliare le casistiche di vincoli risolvibili, che al momento rappresentano un insieme di base.

Per quanto concerne l'AVCParser, possibili migliorie riguardano l'ampliamento della mappatura degli eventi con l'aggiunta di nuovi elementi di layout (così come suggerito per l'analisi statica), eventi di sistema (ad esempio il cambio di stato della batteria) ed eventi esterni (ad esempio l'arrivo di una chiamata). Inoltre potrebbe essere utile uno studio approfondito del lifecycle dei componenti Android per poter comprendere i possibili cambi di stato di essi in base al verificarsi dei vari eventi (esempi di eventi che causano un cambio di stato nel caso di un activity sono evidenziati in Fig. 5) ed ottenere quindi informazioni più precise riguardo alle istruzioni da associare ad ogni evento mappato.

Infine, in riferimento al blocco di dynamic analysis, la scelta di un emulatore diverso permetterebbe di ridurre al minimo potenziali errori legati all'incompatibilità delle versioni di API.

Riferimenti bibliografici

[1] Gartner. Gartner says worldwide sales of smartphones returned to growth in first quarter of 2018. [Online]. Available: https://www.gartner.com/newsroom/id/3876865

[2] Statista. Number of available applications in the google play store from december 2009 to june 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[3] J. Oberheide and C. Miller, "Dissecting the Android Bouncer," in *Summercon 2012*, 2012.

[4] N. J. Percoco and S. Schulte, "Adventures in BouncerLand," in *Proc. of Black Hat USA*, 2012.

[5] CheetahMobile. (2017) Mobile security report for the first half of 2017. [Online]. Available: http://www.cmcm.com/blog/en/security/2017-08-09/1090.html

[6] McAfee. (2018) Mcafee mobile threat report q1, 2018. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf

[7] SophosLabs, "SophosLabs 2018 Malware Forecast," Tech. Rep., 2018.

[8] Statista. Number of available applications in the google play store from december 2009 to june 2018. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[9] H. Yang and R. Tang, "Power Consumption Based Android Malware Detection," *Journal of Electrical and Computer Engineering*, pp. 1–7, 2016.

[10] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 163–173.

[11] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting Android Malware Using Clone Detection," *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 942–956, 2015.

[12] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, *The Soot framework for Java program analysis: a retrospective*, Cetus '11, 2011.

[13] J. Dolby and M. Sridharan, *Static and Dynamic Program Analysis Using WALA*, IBM T.J. Watson Research Center PLDI 2010 Tutorial, 2010.

[14] L. Li, T. F. Bissyand, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67 – 95, 2017.

[15] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proceedings 2014 Network and Distributed System Security Symposium*, 2014, pp. 23–26.

[16] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14*, 2014, pp. 447–458.

[17] Talos S.r.l.s. (2016) Approver. [Online]. Available: https://www.talos-sec.com/

[18] F. Wei, S. Roy, and X. Ou, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," *Ccs*, pp. 1329–1341, 2014.

[19] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight UI-guided test input generator for android," in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, 2017, pp. 23–26.

[20] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground : Automatic Security Analysis of Smartphone Applications," in *CODASPY '13 (3rd ACM conference on Data and Application Security and Privac)*, 2013, pp. 209–220.

[21] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[22] H. Chen, H. F. Leung, B. Han, and J. Su, "Automatic privacy leakage detection for massive android apps via a novel hybrid approach," in *IEEE International Conference on Communications*, 2017.

[23] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, 2013, pp. 1043–1054.

[24] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for Android applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015*, 2016, pp. 461–471.

[25] M. Y. Wong and D. Lie, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware," in *Proceedings 2016 Network and Distributed System Security Symposium*, no. February, 2016, pp. 21–24.

[26] miwong. A targeted input generator for android that improves the effectiveness of dynamic malware analysis. [Online]. Available: https://github.com/miwong/IntelliDroid

[27] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing gui applications," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5885 LNCS, 2009, pp. 69–87.

[28] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for dynamic analysis of iOS applications," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7039 LNCS, 2012, pp. 65–77.

[29] V. L. L. Dantas, "Testing and maintenance of graphical user interfaces," Ph.D. dissertation, INSA Rennes, 2015.

[30] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[31] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, pp. 367–381.

[32] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.