ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea in
Ingegneria e Scienze Informatiche

Sicurezza delle applicazioni Android: definizione di una metodologia per l'individuazione di vulnerabilità nelle librerie

Relatore:
Gabriele D'Angelo

Presentata da:
Denise Nanni

I Sessione Luglio
Anno Accademico 2023/2024

Indice

1 Introduzione

Più della metà della popolazione globale possiede uno *smartphone* [1], un dispositivo diventato indispensabile nella vita di tutti i giorni che permette di svolgere operazioni che spaziano in diversi ambiti e contiene una grande quantità di informazioni personali, come foto, messaggi, dati bancari e medici, password. Grazie al costo vantaggioso e alla sua adozione da parte di molti produttori di smartphone, il sistema operativo *Android* è diventato il più diffuso al mondo, con una quota di mercato che supera il 70% [2]. A differenza del suo maggiore concorrente, *iOS*, Android è *open-source* ed è un sistema molto più aperto, che lascia un buon grado di libertà agli utenti e agli sviluppatori, i quali possono creare applicazioni e distribuirle in maniera semplice tramite gli *store* ufficiali o non ufficiali.

La semplicità con cui è possibile sviluppare e distribuire applicazioni ha portato a un'enorme crescita del mercato, che conta milioni di applicazioni disponibili solo sullo store ufficiale di *Google*, *Google Play Store*, il cui catalogo è in continua espansione [3]. Alla base di ciascuna di esse, vi sono delle funzionalità comuni, come la comunicazione con server remoti, la gestione di dati e la visualizzazione di informazioni, per questo motivo si è resa necessaria la creazione di pacchetti *software*, conosciuti come *librerie*, che implementino tali funzionalità e che possano essere utilizzati in modo semplice e standardizzato. Una libreria è un insieme di funzioni e strutture dati specializzati nella gestione di un determinato aspetto di un'applicazione che può essere facilmente inclusa in un progetto senza conoscerne i dettagli implementativi, permettendo agli sviluppatori di usufruire di codice già ampiamente testato e funzionante, risparmiando tempo e risorse. Quando si include una libreria creata da terzi, si parla di libreria *di terze parti*. Le librerie di terze parti sono utilizzate in quasi la totalità delle applicazioni, rappresentando in media più del 60% del codice di un'applicazione [4], e ne esiste una gran varietà per molti casi d'uso.

La possibilità di includere pacchetti esterni ad un'applicazione da un lato permette il riutilizzo del codice, dall'altro introduce un rischio, perché non si conosce il codice sottostante e non si ha una gestione diretta dello stesso. Gli smartphone, per le loro caratteristiche, sono dispositivi intrinsecamente vulnerabili in quanto il loro obiettivo primario è il risparmio di risorse ed energia, inoltre sono strumenti che vengono utilizzati quotidianamente da un'enorme fetta di popolazione a svariati scopi, rendendoli appetibili per attacchi informatici di vario genere. Quando si include codice esterno, lo si fa con

l'assunzione che esso sia sicuro, ma ciò non è sempre vero. Le librerie di terze parti comportano spesso un rischio aggiuntivo dato che possono contenere falle di sicurezza sfruttabili per compromettere un'applicazione, spesso al fine di rubare dati sensibili; molte di esse implementano insufficienti meccanismi di sicurezza e possono essere una vera minaccia per la privacy degli utenti, specialmente in un mondo in cui il valore dei dati è sempre più alto [5]. Anche quelle distribuite da aziende rinomate non sono esenti da rischi, come successo in passato a *Facebook* e *Dropbox*, le cui librerie contenevano falle che permettevano di rubare dati sensibili, introdurre codice malevolo o impossessarsi dell'account [6].

Il processo di risanamento di una vulnerabilità una volta scoperta e resa nota si compone di tre passaggi che riguardano diversi attori: la prima azione spetta agli sviluppatori responsabili della libreria, che devono celermente correggere il problema e pubblicare una nuova versione; il secondo passo consiste nel migrare alla nuova versione in tutte le applicazioni che ne fanno uso, ciò è compito degli sviluppatori dell'applicazione stessa, che devono poi rilasciare un aggiornamento. Infine, sono gli utenti che rendono effettiva la modifica installando l'aggiornamento proposto. Tale processo può richiedere diverso tempo, a seconda della gravità della vulnerabilità, la presenza di dipendenze nelle applicazioni che complicano la migrazione e l'attenzione degli sviluppatori e degli utenti. Il tempo che intercorre tra la disponibilità di una nuova versione e l'utilizzo nell'applicazione è detto *lag tecnico* ed è stato osservato essere spesso molto lungo, causando un periodo di esposizione al rischio per gli utenti [7]. Non tutti i rilasci di nuove versioni coincidono con la risoluzione di una vulnerabilità, gli sviluppatori di applicazioni tendono a trascurare le librerie, intervenendo solo quando strettamente necessario per includere nuove funzionalità o risolvere problemi, tralasciando invece la sicurezza: nella maggioranza dei casi non si agisce per rimanere al passo con i rilasci finché non si verificano problemi concreti, nonostante molte volte la migrazione richieda sforzi minimi, per noncuranza, timore di incompatibilità o mancata conoscenza di nuove versioni disponibili [8].

L'utilizzo di librerie con vulnerabilità note è un problema poiché espone gli utenti, solitamente inconsapevoli, a violazioni della privacy e compromissione dei propri dati personali. Quanto più un'applicazione rimane affetta da una vulnerabilità più o meno grave, tanto più è possibile che essa venga sfruttata, perciò l'identificazione delle librerie affette e la loro sostituzione con versioni sicure è un'operazione basilare per garantire la sicurezza.

L'obiettivo dello studio condotto è delineare una metodologia con cui ottenere applicazioni da analizzare per ottenere un quadro completo delle librerie utilizzate, quindi valutarne la sicurezza. Nei capitoli seguenti si introdurranno le applicazioni, come sono strutturate e l'ambiente in cui operano, il ruolo delle librerie di terze parti e la valutazione dei rischi, per poi presentare la metodologia proposta e i risultati ottenuti.

2 Applicazioni e sicurezza

Un'applicazione, anche detta "app", è un software in esecuzione su un dispositivo mobile che ha a disposizione un suo spazio di memorizzazione dedicato e con accesso permesso solo ai suoi componenti. Quando si parla di sicurezza delle applicazioni mobile si intende l'insieme di procedure e tecniche utilizzate per la prevenzione di attacchi informatici che, sfruttando le debolezze strutturali di tale applicazione, si infiltrano nel sistema accedendo a informazioni riservate.

Di seguito si fornirà una panoramica di come le applicazioni sono strutturate, sia nella fase di sviluppo che di distribuzione, dell'ambiente in cui vengono eseguite e si tratterà più approfonditamente del concetto di sicurezza.

2.1 Struttura di un'applicazione Android

Un'applicazione è costruita su dei componenti base che costituiscono i punti di accesso per il sistema o l'utente, ciascuno con un proprio ciclo di vita e uno scopo preciso [9]:

- le *activity* sono le schermate di interazione con l'utente, ciò che è visibile e con cui è possibile interagire;
- i servizi sono processi dell'app eseguiti in background per effettuare computazioni più o meno lunghe;
- i *broadcast receiver* sono gestori di eventi ricevuti dal sistema operativo o da altre applicazioni;
- i *content provider* sono gestori di insiemi di dati resi disponibili anche ad altre applicazioni.

Per quanto riguarda lo sviluppo, il progetto di un'applicazione Android è strutturato in modo da separare i vari elementi che la definiscono, principalmente quattro:

- i sorgenti, il codice effettivo;
- le risorse, gli elementi necessari all'applicazione non relativi al codice come immagini o font;
- il *manifest*, un file XML di configurazione che contiene le informazioni essenziali dell'app;

- i file di *build*, dove vengono configurate le dipendenze e la compilazione.

Il manifest contiene i metadati che descrivono approfonditamente i componenti, i permessi utente richiesti, l'hardware e il software utilizzato. È il file più importante, il sistema operativo legge il suo contenuto per costruire l'applicazione, ciò che non è definito al suo interno non esiste e non può essere utilizzato.

I file più importanti ai fini dell'analisi, però, sono quelli di build. Tramite essi si indica come effettuare la compilazione del progetto, modificando le opzioni del compilatore, specificando il percorso del codice sorgente e le dipendenze utilizzate, tra cui le librerie di terze parti. L'esito della compilazione è un file in formato **APK**, un archivio compresso contenente gli elementi necessari per l'esecuzione dell'app, trattato successivamente.

2.1.1 Librerie di terze parti

Una libreria di terze parti (*Third-Party Library*, **TPL**) è un pacchetto software sviluppato e mantenuto da soggetti esterni al gruppo o all'organizzazione che lavora su un progetto. Sono largamente utilizzate per integrare nelle applicazioni delle funzionalità di utilizzo comune di cui, perciò, esistono già implementazioni consolidate e testate, reperibili da *repository* pubblici, come *Maven Central*, oppure gestiti dalla comunità, ad esempio *GitHub*.

Nei repository solitamente è presente uno storico delle versioni delle librerie pubblicate, infatti, oltre a conoscere il nome della libreria da usare, è importante sapere quale versione si vuole includere nel progetto: queste informazioni sono contenute nei file di build. Ci sono vari schemi di numerazione delle versioni, quello più popolare e maggiormente adottato è il *versioning semantico*, in cui il valore assegnato alla versione ha un significato preciso relativo al tipo di modifiche apportate nel rilascio. Il valore è in formato puntato composto da tre campi di cui il primo indica una versione *major*, con modifiche che non mantengono la retrocompatibilità con le altre versioni, mentre il secondo e il terzo si riferiscono principalmente a modifiche compatibili, uno per funzionalità aggiuntive (*minor*), l'altro per correzione di errori (*patch*) [10].

L'importanza della scelta della versione riguarda sia le funzionalità che sono state periodicamente introdotte, sia la stabilità. Le buone pratiche riguardo l'adozione di librerie esterne consigliano di mantenersi quanto più possibile al passo con gli ultimi rilasci per essere certi di avere le funzionalità più aggiornate e le correzioni di bug più recenti oltre ai progressivi *fix* di sicurezza.

Le librerie Android più diffuse possono essere distinte tra quelle relative al *frontend*, che si occupano di animazioni, elementi di interfaccia, caricamento ottimizzato di immagini, e *backend*, ad esempio supporto alla *dependency injection*, crittografia, controllo della memoria, gestione di dati.

Uno studio del 2019 ha analizzato le abitudini degli sviluppatori per quanto concerne l'adozione delle **TPL**, incluse nell'82% delle applicazioni, e ha evidenziato la tendenza diffusa

a non aggiornare le librerie in maniera costante o, addirittura, a non aggiornarle affatto [7]. In particolare, le librerie backend, non impattanti sull'interfaccia, sono maggiormente trascurate nonostante il loro ruolo fondamentale, poiché non direttamente percepibili dall'esperienza utente. Considerata la loro criticità, sarebbe invece appropriato mantenerle al passo con le ultime versioni dato che da loro dipende una buona parte della gestione dell'app e, di conseguenza, della sua sicurezza.

2.1.2 Android Package

Per la distribuzione e l'installazione di un'app Android si utilizzano gli *Android Package*, file in formato APK che contengono tutto ciò che serve all'applicazione per essere eseguita. Il formato APK non è altro che un archivio compresso assimilabile ad uno ZIP quindi, estraendolo, si può esaminare il suo contenuto che segue una struttura ben definita [11]:

- `AndroidManifest.xml`, il file obbligatorio che descrive l'applicazione;
- `META-INF`, una cartella contenente i file di firma digitale;
- `MANIFEST.MF` contiene l'hash di tutti i file dell'archivio ed è utilizzato per verificare l'integrità dell'APK;
- `res` e `assets`, cartelle contenenti risorse e file di asset;
- `lib`, una cartella contenente le librerie native;
- `kotlin`, la cartella specifica per i progetti Kotlin;
- `classes.dex`, contenente i file dell'applicazione compilati in DEX bytecode (*Dalvik Executable bytecode*).
- `resources.arsc`, un file relativo ai collegamenti tra il codice e le risorse;

I file DEX sono particolarmente importanti perché contengono l'intero codice sorgente dell'applicazione, incluso quello delle librerie di terze parti. Specialmente per applicazioni corpose, spesso si trovano più file DEX denominati `classes` seguiti da un numero progessivo; questa suddivisione è necessaria a causa del limite di 64000 metodi per file, superato il quale si deve ricorrere a delle tecniche di ottimizzazione, che siano esse atte a ridurre il numero di metodi o a dividere il codice in più file [12]. Questo limite non è imposto dal sistema operativo ma dai *runtime* Dalvik e ART, che sono i motori di esecuzione delle applicazioni Android.

Un altro formato di distribuzione utilizzato da Google è l'*App Bundle*, un pacchetto che contiene tutto il codice e le risorse dell'app e rimanda la creazione e la firma dell'APK al momento del download. L'idea è di costruire un pacchetto più piccolo contenente quelle parti di codice e risorse che sono comuni a tutti i dispositivi affiancato da altri APK, detti *split*, che contengono le parti specifiche per ciascun dispositivo; sarà lo store di Google

al momento del download a generare tutti gli archivi specifici per il dispositivo in uso, riducendo così lo spazio occupato e il tempo di download [13].

2.2 L'ambiente Android

Android è un sistema operativo *open-source* sviluppato da Google, progettato per dispositivi mobili come smartphone e tablet, diventato popolare grazie alla sua versatilità, personalizzazione e alla vasta gamma di applicazioni disponibili. L'architettura del sistema Android, mostrata nella figura 2.1, si compone di cinque livelli principali [14]:

- *kernel Linux*: è il cuore del sistema operativo, responsabile della gestione delle risorse hardware, della sicurezza e della comunicazione tra le applicazioni e l'hardware. Il kernel Linux è stato modificato per adattarsi alle esigenze di Android, aggiungendo nuovi *driver* e funzionalità per supportare i dispositivi mobili, ma fornisce tutte le tecniche di sicurezza tipiche di Unix;
- *Hardware Abstraction Layer* (HAL): è un'interfaccia tra il kernel Linux e il *framework* Android, che permette di utilizzare l'hardware del dispositivo senza dover conoscere i dettagli di implementazione;
- *runtime Android* (ART): è il motore che esegue i file DEX dell'applicazione, ne viene creato un'instanza per ciascuna e utilizza metodi di ottimizzazione come la compilazione *Just-In-Time* (JIT) e *Ahead-Of-Time* (AOT) o una migliore *garbage collection* per migliorare le prestazioni su dispositivi con memoria ridotta;
- *librerie native*: scritte in C o C++, sono utilizzate dai componenti e servizi per interfacciarsi con HAL e ART;
- *framework Java*: contiene le API e le librerie Java che compongono le applicazioni Android, sia quelle di sistema che quelle esterne;
- *applicazioni di sistema*: sono le applicazioni preinstallate sul dispositivo, ad esempio il browser, il gestore di contatti o il calendario.

Un meccanismo di sicurezza Unix è il concetto di separazione dei processi e dei file utente: ciascuna applicazione installata, infatti, viene trattata come un utente separato con un processo dedicato e identificato da un *Unique User ID* (UID), il quale è utilizzato per creare un ambiente isolato, detto *sandbox*, in cui viene eseguita l'applicazione, proteggendo così il sistema stesso e le altre applicazioni da attività pericolose [15].

Normalmente, Android non fornisce all'utente tutti i privilegi possibili (privilegi di *root*), quindi alcune operazioni strettamente legate al sistema operativo, come la gestione dell'hardware e di porzioni riservate di memoria, non sono consentite. Attraverso applicazioni apposite, software per computer o procedure personalizzate, è possibile togliere queste limitazioni per avere accesso al sistema nella sua interezza [16]; tuttavia, si tratta

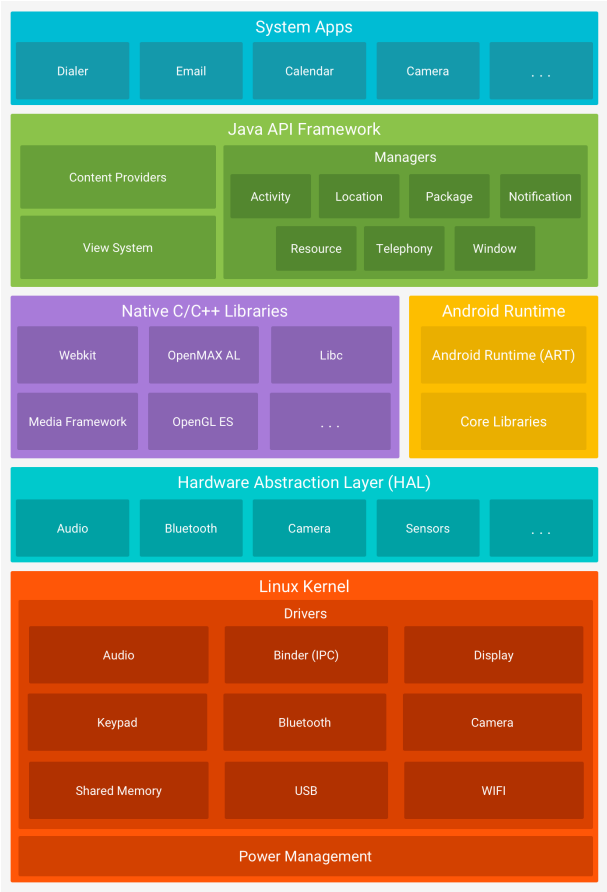

Figura 2.1: Architettura del sistema operativo Android [14].

di un'operazione rischiosa perché potrebbe corrompere il dispositivo se eseguita male e, anche se effettuata con successo, espone maggiormente ai malware e alle debolezze delle applicazioni, dato che un attaccante può uscire dalla sandbox sfruttando più funzionalità e accedendo a più dati.

Gli **APK** delle applicazioni possono essere scaricati da repository online oppure direttamente dagli store ufficiali che scaricano e installano automaticamente nascondendo la procedura all'utente, il più popolare è *Google Play*. Oltre all'ampio catalogo, Google Play offre i vantaggi di una gestione intelligente della distribuzione degli **APK** grazie agli App Bundle e si pone l'obiettivo di mantenere l'utente al sicuro tramite il sistema Play Protect che scansiona giornalmente le applicazioni del dispositivo alla ricerca di malware [17]. Nonostante questi meccanismi di monitoraggio e le ulteriori protezioni del sistema operativo, le **TPL** possono comunque essere una minaccia, specialmente nel caso di un *rooted phone*[1], perché contengono del codice sconosciuto che potrebbe avere delle falle di sicurezza di vari livelli di gravità (*severity*) e sfruttabilità (*exploitability*).

[1]Un dispositivo Android dotato dei privilegi di root

2.3 Vulnerabilità

Per loro natura gli smartphone e i tablet sono più proni ad avere problemi di sicurezza rispetto ai computer poiché nascono come dispositivi il cui obiettivo principale è la portabilità e il risparmio di risorse, inoltre sono diventati il mezzo principale per svolgere numerose operazioni che coinvolgono dati strettamente personali [18]. Le applicazioni mobile devono essere progettate in modo da porre particolare attenzione sulla gestione dei dati e sul corretto utilizzo delle risorse per non esporre componenti sfruttabili per compromettere il sistema. Quando si usano le TPL, l'aspetto della sicurezza viene parzialmente delegato, potenzialmente introducendo maggiori rischi per l'utente. Si parla di vulnerabilità quando vi sono delle debolezze nella logica computazionale di componenti hardware o software che, se sfruttate, possono compromettere i requisiti di confidenzialità, integrità e disponibilità delle informazioni [19].

Il programma *Common Vulnerabilities and Exposures* (CVE) [20] si occupa di raccogliere le segnalazioni di vulnerabilità riscontrate in applicazioni software e librerie, catalogarle ed eseguire test per verificarne le caratteristiche. Una volta conclusa la valutazione, viene pubblicato un CVE, identificato dall'anno di pubblicazione e un numero, contenente la descrizione del problema e corredato da una serie di informazioni utili per stimarne la gravità.

Per lo scopo dello studio condotto, si considererà indicativa la sola presenza di CVE relativi alle librerie, tralasciando parzialmente l'aspetto della criticità.

3 Analisi di applicazioni

Per rilevare le vulnerabilità nelle app Android, il primo passo è quello di reperire i file APK delle applicazioni da analizzare. Una volta ottenuti i file, è necessario individuare le librerie utilizzate, per poi confrontarle con le versioni note per contenere vulnerabilità. Infine, è possibile cercare i CVE relativi alle librerie individuate e valutarne la criticità.

Ciascuna di quese operazioni verrà descritta nelle sezioni seguenti, con particolare attenzione all'approccio utilizzato e alle difficoltà riscontrate. I risultati ottenuti verranno presentati nel capitolo seguente.

3.1 Reperimento degli APK

L'ottenimento degli APK è un'operazione disincentivata da Google, per questo non esistono, ad oggi, metodi ufficiali per scaricare i file direttamente dal Play Store tramite script; anche in qualità di sviluppatore, un utente non può scaricare un APK di cui non è proprietario. Sono stati pubblicati diversi script online che permettono lo scaricamento (*download*) dal Play Store, ma la loro efficacia è limitata dalla continua evoluzione del servizio e dalla presenza di controlli di sicurezza. In particolare, una libreria utilizzata da vari progetti, conosciuta come *gpapi* [21], offre i meccanismi di autenticazione ai server per poter cercare e scaricare app; tuttavia, questo strumento è diventato ormai obsoleto poiché utilizza metodi di autenticazione e servizi deprecati.

Per ovviare al problema si potrebbe decidere di usufruire dei repository online di APK come *APKPure* o simili, siti che mettono a disposizione un vasto catalogo di app Android. Questo metodo è difficilmente automatizzabile poiché la maggioranza dei siti richiedono il superamento delle verifiche di sicurezza per impedire l'accesso ai bot. Inoltre, per la natura dell'analisi svolta, la migliore opzione è attingere dallo store ufficiale per avere certezza dell'autenticità e integrità dei dati, quindi l'utilizzo di terze parti è da considerarsi un'estensione dello studio piuttosto che un'alternativa.

Una soluzione più tecnica, ma con dei limiti, è l'utilizzo degli strumenti di sviluppo e debug forniti da Android su un dispositivo emulato. L'emulazione di un dispositivo Android permette di ottenere naturalmente le app dal Play Store, tuttavia la procedura richiede di completare il download e l'installazione, due operazioni a velocità molto variabile che introducono un tempo di attesa importante, specialmente per applicazioni pesanti, perciò il set di dati sarà fortemente ridotto.

3.1.1 Android Debug Bridge

L'*Android Debug Bridge* (**ADB**) è uno strumento di sviluppo per Android che permette di comunicare con un dispositivo fisico o un emulatore. Si tratta di un programma client-server composto dal client, la parte che invia i comandi, il server, che media la comunicazione con un daemon, il terzo componente, in esecuzione sul dispositivo [22]. Il daemon è chiamato *adbd* e lavora come processo di sistema, quindi ha accesso a funzionalità che non sono disponibili per un'applicazione di terze parti.

Per utilizzare **ADB** è necessario installare il pacchetto `platform-tools` di Android SDK, che contiene il programma `adb`, sul computer usato per l'estrazione. Una volta installato, è possibile usufruire di comandi che coprono una vasta gamma di funzionalità, tra cui la possibilità di scaricare file, inviare comandi shell e molto altro. In particolare, sono notevoli il *package manager* e l'*activity manager*, due componenti per la gestione delle applicazioni e delle attività, rispettivamente. Il package manager permette di eseguire azioni e query sui pacchetti installati, mentre l'activity manager fornisce vari strumenti per eseguire azioni di sistema. Questi due elementi sono fondamentali per l'analisi che si vuole condurre, in quanto permettono di ottenere informazioni sui pacchetti e avviare attività, come quella del Play Store per scaricare applicazioni.

Il metodo per ottenere gli **APK** sfrutta gli *intent*, messaggi utilizzati per richiedere un'azione a un'altra app [23], in combinazione con i *deep link*, **URI**[1] che permettono di navigare direttamente a una pagina specifica di un'app [24], per avviare l'activity del Play Store relativa ai dettagli dell'applicazione desiderata, con il comando mostrato nel listato 3.1, dove è presente il pulsante di download sul quale si simulerà un tocco.

```
1 > adb shell am start -a android.intent.action.VIEW -d
    "https://play.google.com/store/apps/details?id=com.example.app"
```

Listing 3.1: Esempio di utilizzo di **ADB** per avviare l'activity del Play Store.

Questo metodo è stato scelto per la sua semplicità, in quanto non richiede l'installazione di app di terze parti o l'uso di servizi online, tuttavia ha dei limiti: è strettamente legato al posizionamento del pulsante e alla struttura della pagina, perciò se viene modificato il layout, bisogna ricavare nuovamente le coordinate, inoltre bisogna introdurre un piccolo *delay* per essere certi che l'activity abbia terminato il caricamento, altrimenti il tocco (*tap*) sarà inefficace.

Identificativi delle app

Per identificare un'applicazione, si utilizza il nome del pacchetto (*package name*), definito in fase di sviluppo: una stringa in notazione puntata di due o più campi che individua univocamente l'app all'interno del sistema e del Play Store. Per posizionarsi nella schermata di dettaglio bisogna conoscere il package name e passarlo come parametro dell'**URI**. Per

[1] Uniform Resource Idenfitier, sequenza di caratteri che identifica una risorsa.

esempio, l'URI riportato nel listato 3.1 apre la pagina di dettaglio dell'app identificata da `com.example.app`. Non esiste un metodo lineare per ricavare i package name, non è possibile dedurli dal nome dell'app poiché possono essere anche molto diversi quindi la soluzione è collezionarli manualmente, ad esempio tramite il Play Store da *browser* in cui il package name, come nel caso mobile, è presente nell'URI della pagina di dettaglio, oppure sfruttando la ricerca tramite uno *script*.

Quando si effettua una ricerca sullo store da browser, viene compilato un URL[2] della forma `https://play.google.com/store/search?q=query&c=apps`, in cui `query` sono le chiavi di ricerca mentre `c=apps` indica la categoria applicazioni, e la risposta è una pagina *web* che contiene i risultati, in cui i package name sono ritrovabili nei *tag* di *link*, quindi è possibile estrarli con un *parser* HTML[3]. Tramite questo metodo, si ottengono i package name di tutte le app relazionate con la query di ricerca, utile nel caso in cui si voglia ottenere un insieme di applicazioni correlate. Ai fini della ricerca, è preferibile avere un set di dati ben definito perciò si è scelto di utilizzare un insieme di app popolari e di interesse generale, come *social network*, *home banking* e della pubblica amministrazione. Analizzando il sorgente della pagina web si riscontra una ricorrenza utile per individuare solo l'app desiderata: con una ricerca sufficientemente accurata il primo risultato è differente rispetto agli altri in termini di layout perciò ha una classe di stile unica, della quale ci si può servire per leggere il link dal quale ricavare il package name.

Installazione delle app

Una volta aperta la pagina di dettaglio dell'applicazione, è necessario simulare il tocco sul pulsante di installazione per avviare il processo di scaricamento. Per fare ciò, si utilizza il comando `adb shell input tap <x> <y>`, dove `x` e `y` sono le coordinate del punto in cui si vuole toccare. I valori `x` e `y` possono essere ottenuti manualmente, eseguendo un tap con la funzionalità di *overlay* che mostra le coordinate sullo schermo, attivabile dalle impostazioni sviluppatore.

Le pagine di dettaglio del Play Store hanno tutte una struttura simile, ma si può facilmente notare che il pulsante di installazione può subire degli spostamenti più o meno pronunciati in base alla dimensione del nome dell'applicazione, ciò comporta una complicazione perché in caso di variazioni il tocco potrebbe non avviare lo scaricamento. Con qualche tentativo pratico si riescono a stimare delle coordinate corrette nella maggioranza dei casi, perciò questo metodo è da considerarsi affidabile ma non perfetto.

In alternativa, si potrebbe utilizzare un approccio più sofisticato basato sull'analisi della struttura della pagina, per individuare il pulsante di installazione e ottenerne le coordinate, infatti tramite ADB si può accedere al file XML che descrive il layout. Eseguendo `adb shell uiautomator dump` mentre l'emulatore è sulla pagina scelta viene generato il file

[2]Uniform Resource Locator, URI specifico per le risorse web.
[3]HyperText Markup Language.

`window_dump.xml`, salvato nella memoria esterna del dispositivo, quindi si può fare un *pull*, copiandolo in locale tramite il comando `adb pull /sdcard/window_dump.xml`. Il contenuto è suddiviso in nodi (*node*), che rappresentano ciascun elemento della pagina, con delle proprietà, tra cui l'attributo `bounds` che indica la posizione e le dimensioni dell'elemento.

Tra tutti i nodi, si può identificare quello con la scritta corrispondente a quella riportata sul pulsante di installazione, come mostrato nell'esempio 3.2, e ricavare le coordinate.

```
<node index="0" text="Install" class="android.widget.TextView"
    package="com.android.vending" content-desc="Install" ...
    bounds="[490,796][592,849]" />
```

Listing 3.2: Esempio di nodo di un file XML generato da ADB.

Questo metodo introduce una complessità maggiore perché per ogni applicazione sarebbe necessario fare il dump, scorrerlo e trovare il nodo corrispondente al pulsante, perciò è stato scartato in favore di un approccio più semplice che, nel caso specifico, si rivela altrettanto efficace.

Una criticità di questa fase è la disponibilità di spazio di archiviazione sul dispositivo, poiché se la memoria è satura, l'installazione dell'app non è possibile. È opportuno effettuare dei controlli sullo spazio disponibile, oppure rimuovere le app all'occorrenza.

Estrazione degli APK

Considerato che l'APK è necessario per l'esecuzione dell'app, è ragionevole pensare che il file venga mantenuto sul dispositivo in qualche area di memoria riservata, non accedibile dall'utente. Questa area di memoria è la cartella `/data/app` che ha i permessi di lettura e scrittura solo per il sistema operativo, come si può vedere dalla figura 3.1; per aggirare questo limite, che altrimenti richiederebbe un dispositivo con i privilegi di root, si lavora tramite il package manager.

```
drwxrwx--x  50 system system     4096 2024-06-11 16:02 data
```

Figura 3.1: Permessi della cartella in cui sono memorizzate le applicazioni.

Dal package name non è possibile risalire direttamente alla directory in cui è contenuto l'APK perché la struttura in cui vengono memorizzati i file contiene diversi caratteri casuali utilizzati per rendere più difficile l'accesso alla memoria. Per questo motivo bisogna ricorrere package manager, utilizzando il comando `adb shell pm path com.example.app`, mostrato nel listato 3.3, per ottenere il percorso assoluto del file, tramite il quale si può successivamente fare il pull.

```
> adb shell pm path com.example.app
package:/data/app/~~0tDjqm2t_QB92TSnuCyqpg==/
    com.example.app-2pGPttO0rlUtoa4PmnLwXg==/base.apk
```

```
3 package:/data/app/~~0tDjqm2t_QB92TSnuCyqpg==/
    com.example.app-2pGPttO0rlUtoa4PmnLwXg==/split_config.x86_64.apk
4 package:/data/app/~~0tDjqm2t_QB92TSnuCyqpg==/
    com.example.app-2pGPttO0rlUtoa4PmnLwXg==/split_config.xxhdpi.apk
```

Listing 3.3: Esempio di utilizzo del package manager per ottenere i percorsi degli APK.

```
1 > adb pull /data/app/~~0tDjqm2t_QB92TSnuCyqpg==/
    com.example.app-2pGPttO0rlUtoa4PmnLwXg==/base.apk
```

Listing 3.4: Esempio di pull dell'APK.

Per evitare di saturare la memoria del dispositivo, è opportuno rimuovere l'applicazione una volta copiato il file in locale, utilizzando il comando `adb shell pm uninstall com.example.app`, che elimina l'applicazione e tutti i dati a essa associati.

Come si può notare dall'output esemplificativo nel listato 3.3, all'interno della cartella ci possono essere più file: l'APK base e i relativi split. In questo caso, perché l'applicazione funzioni correttamente e possa essere analizzata nelle fasi successive, è necessario fare il *merge* e unirli in un unico pacchetto, ad esempio sfruttando tool come *APKEditor* [25], che permette di fare questa operazione da riga di comando eseguendo un file `.jar`, previa installazione di *Java*. Al termine, si ha a disposizione l'APK dell'applicazione da analizzare e si può procedere con l'analisi delle librerie.

3.2 Individuazione delle librerie

Includere librerie all'interno di un'applicazione permette di semplificare e velocizzare il processo di sviluppo, integrando funzionalità già create e testate da altri. Ai fini dell'analisi, oltre a conoscere quali librerie sono state utilizzate, è importante sapere anche le versioni, poiché le vulnerabilità possono essere presenti solo in alcune e risolte nelle altre, oppure possono essere state introdotte solo in versioni successive a quella scelta.

Per definire dei metodi di rilevazione, si può ragionare partendo dal file di build, in cui sono elencate le coppie libreria-versione: se fosse presente un file analogo all'interno dell'APK l'operazione di rilevazione sarebbe molto più semplice. Analizzando gli archivi con diversi approcci è emerso che questo file non è incluso e non esiste un suo corrispettivo, perciò bisogna ricorrere a metodi più onerosi.

Durante la compilazione, il codice sorgente delle librerie viene compilato e incluso nell'APK finale, perciò potrebbe essere utilizzato per risalire al nome. Se si decomprime l'archivio come un normale file compresso, si ottiene la struttura di directory descritta nella sezione 2.1.2, il cui contenuto è prevalentemente in formato binario, quindi non immediatamente leggibile e che non ha alcun riferimento diretto alle librerie, perciò è necessario utilizzare uno strumento di *reverse engineering* per ottenere maggiori informazioni.

Il *reverse engineering* [26] è il processo di analisi di un sistema per identificare i suoi componenti e le loro relazioni e creare una rappresentazione in un'altra forma o a un più

alto livello di astrazione. Questo metodo permette di risalire a una rappresentazione più comprensibile dell'archivio, incluso il codice sorgente delle librerie.

Attraverso *Apktool* [27], un tool di reverse engineering di applicazioni Android largamente utilizzato, è stato possibile analizzare più approfonditamente il contenuto degli archivi. Con questo tipo di estrazione, la struttura della cartella differisce in parte da quella ottenuta con una normale decompressione, in quanto Apktool si occupa di ricostruire il codice sorgente dai file DEX tramite un processo chiamato *backsmaling*, ottenendo codice in un linguaggio conosciuto come *Smali* [28].

I file `.smali` sono contenuti in percorsi di cartelle che richiamano il package name delle librerie perciò è possibile risalire con un sufficiente grado di accuratezza al nome, come mostrato in figura 3.2. Nell'applicazione mostrata *MyUnibo*, l'app per gli studenti dell'Università di Bologna, si può notare il percorso simile alla struttura di un package name e infatti corrispondono a librerie esterne esistenti, disponibili su Maven Central, ad esempio `com.google.firebase`, `com.google.zxing` e `net.minidev.asm`.

```
└─$ find com.myunibo/smali_classes2/ -mindepth 3
com.myunibo/smali_classes2/com/google/firebase/components/Component.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentContainer$-CC.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentContainer.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentDiscovery$1.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentDiscovery$MetadataRegistrarNameRetriever.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentDiscovery$RegistrarNameRetriever.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentDiscovery.smali
com.myunibo/smali_classes2/com/google/firebase/components/ComponentDiscoveryService.smali
com.myunibo/smali_classes2/net/minidev/asm/BeansAccess.smali
com.myunibo/smali_classes2/net/minidev/asm/BeansAccessBuilder.smali
com.myunibo/smali_classes2/net/minidev/asm/BeansAccessConfig.smali
com.myunibo/smali_classes2/net/minidev/asm/ConvertDate$StringCmpNS.smali
com.myunibo/smali_classes2/net/minidev/asm/ConvertDate.smali
com.myunibo/smali_classes2/net/minidev/asm/DefaultConverter.smali
com.myunibo/smali_classes2/net/minidev/asm/DynamicClassLoader.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/encoder/ByteMatrix.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/encoder/Encoder$1.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/encoder/Encoder.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/encoder/MaskUtil.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/encoder/MatrixUtil.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/encoder/QRCode.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/QRCodeReader.smali
com.myunibo/smali_classes2/com/google/zxing/qrcode/QRCodeWriter.smali
```

Figura 3.2: Esempio di percorsi contenenti dei file Smali.

Anche con questa ricostruzione, non si riesce a risalire direttamente alle versioni, perciò si rende necessario l'utilizzo di tecniche più sofisticate, che vadano ad analizzare approfonditamente le componenti del codice.

3.2.1 Tecniche di offuscamento

Una sfida che si può incontrare con l'utilizzo di queste tecniche è l'offuscamento del codice, ovvero la pratica di rendere il codice sorgente o il bytecode più difficile da comprendere o da decompilare. Questa pratica è molto comune e incoraggiata nelle applicazioni Android poiché, oltre a proteggere il software dal reverse engineering, può anche ridurre le dimensioni del file. Le tecniche si dividono tra quelle che intervengono direttamente sul bytecode e quelle che, invece, si limitano ad eseguire trasformazioni esterne, le più comuni sono [29] [30]:

- rinominazione degli identificatori (*identifier renaming*), che consiste nel cambiare i nomi di variabili, metodi, classi e altro in brevi stringhe prive di significato, per rendere il codice meno leggibile ma anche per ridurre lo spazio occupato da essi;
- cifratura delle stringhe, per nascondere i dati all'interno del codice sostituendoli con stringhe cifrate poi decifrate a runtime;
- *reflection*, che permette di accedere a metodi e attributi di una classe in modo dinamico, senza conoscere il nome a priori, complicando l'analisi del codice;
- *dead code removal* e *code addition*, che consistono rispettivamente nella rimozione di codice inutilizzato che potrebbe fornire particolari informazioni e nell'aggiunta di codice fittizio per confondere il lettore;
- *control flow randomization*, ossia l'introduzione di punti decisionali fittizi che riorganizzano il grafo del flusso di controllo e interferiscono con le rilevazioni;
- *package flattening*, ossia la riduzione della gerarchia delle cartelle, per rendere più difficile la navigazione all'interno del codice;
- *packaging*, tecnica che sfrutta un APK come contenitore di quello reale cifrato;
- cifratura dei DEX, rendendoli inaccessibili da strumenti di reverse engineering, ma ripristinati runtime;
- trasformazione del manifest, aggiungendo, per esempio, ulteriori permessi o componenti.

Osservando la gerarchia di un APK come quella di un'app di home banking possiamo notare la presenza di cartelle con nomi generici, mostrate in figura 3.3, che potrebbero essere risultato di un'operazione di offuscamento di tipo identifier renaming.

Figura 3.3: Esempio di gerarchia di directory con offuscamento.

Sono disponibili strumenti come *ProGuard* [31] o *DexGuard* [32] che si occupano di offuscare il codice in fase di compilazione secondo regole stabilite dallo sviluppatore infatti, per assicurare il corretto funzionamento dell'applicazione anche in seguito all'offuscamento, è necessario configurare il tool in modo che non vengano alterate parti di codice essenziali. Sebbene questa necessità esponga il codice a un livello di modifica inferiore, il grado di mascheramento introdotto da queste tecniche è comunque significativo e richiede l'utilizzo di strumenti che siano in grado di superarlo.

3.2.2 Stato dell'arte

Il problema dell'individuazione delle librerie è già stato affrontato in letteratura e sono stati proposti diversi strumenti che si propongono di risolverlo con un buon livello di accuratezza, superando le limitazioni introdotte dall'uso di tecniche di offuscamento. I tool di rilevazione si dividono in tre categorie, in base alla procedura di ricerca: si possono utilizzare tecniche di *clustering* che raggruppano le librerie in base a delle caratteristiche comuni, di ricerca di similitudini (*similarity-based*) che confrontano le librerie con un database di riferimento e di *learning* che utilizzano algoritmi di apprendimento automatico per riconoscere le librerie. Tra i tool che fanno uso di clustering troviamo *LibRadar* [33], il quale cerca di limitare il dispendio di tempo causato dalle comparazioni in favore di un'analisi più rapida, basata sul confronto e raggruppamento di *fingerprint* delle librerie, ossia di una rappresentazione compatta ottenuta dall'elaborazione delle funzionalità presenti.

Quelle maggiormente utilizzate e sviluppate sono le tecniche similarity-based, che permettono di ottenere risultati più precisi e affidabili, in quanto si basano su un confronto diretto tra le librerie analizzate e quelle note, prese da un database che deve essere mantenuto aggiornato. I tool più recenti che fanno affidamento su questa tecnica sono *LibScout* [30], *LibID* [34] che si propone come migliorativo di LibScout, e *LibScan*, che risulta essere il più recente.

3.2.3 LibScan

La scelta di LibScan è dovuta alla sua recente pubblicazione e ai risultati promettenti descritti nella relativa pubblicazione [35]. LibScan è uno strumento di rilevazione delle librerie che si basa su un database di librerie note, che vengono confrontate con quelle presenti all'interno dell'APK. Il confronto avviene tramite l'analisi delle classi e dei metodi, che vengono raffrontati con quelli delle librerie note attraversando diverse fasi di elaborazione. Il set di TPL da ricercare deve essere fornito al tool in input in formato JAR, eseguibile Java, o AAR, libreria specifica per Android, poi ciascun file verrà convertito in DEX e usato per i confronti.

Inizialmente, l'APK viene decompilato e le classi vengono comparate con ciascuna presente in una TPL definendo un set di relazioni di corrispondenza per coppia di classi, quindi viene calcolato un punteggio di accuratezza, a cui ci si riferirà anche come affidabilità, che fornisce una misura dell'accuratezza della rilevazione: la libreria viene considerata presente se il punteggio è superiore a una soglia definita per ogni step del procedimento. Più in dettaglio, il tool analizza l'app ed estrae le informazioni, incluse le fingerprint, poi continua con il processo di confronto che si divide in tre fasi, ciascuna con un tipo diverso di rilevazione per aumentare la copertura rispetto all'offuscamento: la prima fase si basa sulla firma delle classi, ottenuta in base alle loro caratteristiche, la seconda analizza gli

opcode dei metodi, ovvero le istruzioni macchina che indicano le azioni, mentre la terza fase si concentra sulle chiamate ai metodi, risalendo la catena di invocazioni. Al termine della seconda fase, i risultati vengono combinati e viene calcolato un primo punteggio, che serve per filtrare anticipatamente le coppie e snellire i dati da computare nella terza fase, nella quale si calcola il punteggio finale e si tengono solo le occorrenze che superano una seconda soglia.

Il risultato di questa analisi è un report che contiene le librerie rilevate, con il relativo punteggio di somiglianza, che sarà tanto più vicino a uno quanto più certa è la presenza effettiva della TPL.

```
lib: commons-io-2.5
similarity: 0.5932874516828509

lib: core-3.4.0
similarity: 0.43104514533085964

lib: okhttp-4.2.0
similarity: 0.40401087695445276

lib: okhttp-4.2.1
similarity: 0.4041128484024473

lib: okhttp-4.2.2
similarity: 0.40492694529391776

lib: picasso-2.5.0
similarity: 0.5473217881594845

lib: retrofit-2.5.0
similarity: 0.4694607717863532

lib: retrofit-2.6.0
similarity: 0.6166743224621039

lib: retrofit-2.6.1
similarity: 0.6190799909358713

time: 228s
```

Figura 3.4: File di output di LibScan.

3.3 Ricerca e valutazione dei CVE

Una volta individuate le TPL, bisogna verificare se esistono CVE relativi. Ricercando il nome della libreria e la versione all'interno di database come NVD [36] si può ottenere l'elenco di tutte le vulnerabilità note della libreria o di altre componenti che essa utilizza. Nella repository Maven Central è possibile trovare, se presenti, i CVE associati a ogni versione di una libreria, inclusi quelli relativi alle dipendenze, perciò si può utilizzare questo servizio per ottenere informazioni, oppure per scegliere specificamente solo le librerie note per contenere vulnerabilità.

La struttura di un CVE si compone del suo identificatore, una data di pubblicazione, una descrizione e delle caratteristiche della vulnerabilità. In base alle caratteristiche vengono calcolati dei punteggi secondo il *Common Vulnerabilities Scoring System* (CVSS) [37] che definisce le metriche di valutazione e i coefficienti.

Ci sono tre gruppi di metriche usati per valutare una vulnerabilità, base, temporale e ambientale. Le ultime due introducono elementi aggiuntivi atti a contestualizzare maggiormente e raffinare il punteggio base, che è quello principale. Le metriche base sono:

- metrica di *exploitability*, ovvero quanto è semplice approfittare della falla, composta da:
  - vettore di attacco, che può essere la rete internet, un determinato protocollo, la rete locale o la rete fisica;
  - complessità, definita in base all'esistenza di particolari condizioni necessarie per effettuare un attacco;
  - privilegi richiesti, quindi il grado di privilegio che un attaccante deve ottenere per poter sfruttare la falla;
  - interazione dell'utente, se è necessario l'intervento di un umano oltre l'attaccante;
- ambito (*scope*), se l'attaccante può compromettere anche risorse esterne a quella direttamente interessata dalla vulnerabilità (*changed* o *unchanged*);
- metrica di impatto (*impact*), definisce la misura in cui una vulnerabilità sfruttata compromette i requisiti di confidenzialità, integrità e disponibilità.

Il punteggio che fornisce una misura complessiva della gravità della vulnerabilità è il *base score*, un valore decimale compreso tra 0 e 10 derivato dalle metriche sopraindicate. Ad esempio, una vulnerabilità facilmente sfruttabile via internet, che non richiede permessi né interazioni di terzi avrà un punteggio più alto di una che invece richiede interazioni fisiche per cui servono anche dei permessi particolari. Un altro punteggio significativo è l'*exploitability score* che indica la facilità con cui si può sfruttare la falla di sicurezza, quindi un valore alto indica una vulnerabilità più pericolosa. Questo punteggio si calcola sulle metriche di exploitability e il suo valore massimo è 3,9.

Una volta ottenute le librerie è possibile consultare i CVE relativi, in modo da poter dare un giudizio sulla sicurezza dell'applicazione.

4 Risultati

Per condurre l'analisi, tramite il software *Android Studio*, è stato emulato un dispositivo *Pixel 7a*, con sistema operativo Android 14 equipaggiato con Google Play Store, configurato con un account Google e la lingua italiano. Il dispositivo è stato utilizzato per scaricare le applicazioni da analizzare, che sono state poi trasferite sul computer per l'analisi. Il sistema operativo del computer è *Windows 11*, con processore *Intel Core i7* di ottava generazione e 16 GB di RAM.

Il campione utilizzato consta di 30 applicazioni appartenenti a diverse categorie come social, home banking, shopping, pubblica amministrazione, per verificare le diverse tendenze in base allo scopo dell'app. Il numero ridotto è dovuto alla quantità di tempo necessaria per l'analisi in tutte le sue fasi, ma potrebbe essere ampliato in futuro per avere una visione più completa del panorama delle applicazioni presenti sullo store.

Il reperimento degli APK è stato effettuato tramite un semplice script *Python*, disponibile su GitHub[1], che ricerca le app sul web per ricavare il package name, quindi utilizza ADB per avviare le activity e l'installazione delle applicazioni sul dispositivo emulato. Al termine dell'installazione, sempre tramite ADB, trasferisce gli APK sul computer e utilizza APKEditor per effettuare il merge degli split, qualora ce ne siano.

Il set di librerie in input per il tool è composto da un totale di 20 librerie, ciascuna con diverse versioni per un totale di 315 file. La scelta è stata fatta in base alla loro popolarità e diffusione, facendo riferimento sia ad articoli sul web [38], sia alle informazioni ottenute dall'analisi degli APK processati con Apktool, considerando solo librerie gratuite, i cui file JAR sono facilmente reperibili da Maven Central. La tabella 4.1 mostra le principali librerie utilizzate per i rilevamenti. Per alcune librerie si ha un insieme relativamente ristretto di versioni, poiché lo strumento utilizzato da LibScan per convertire in DEX non riesce a convertire alcuni JAR che risultano di una dimensione ridotta rispetto ad altri della stessa libreria.

Per stimare l'accuratezza di LibScan sono stati inseriti gli APK di applicazioni di controllo non influenzate da tecniche di offuscamento per verificare che il tool rilevasse le librerie e con quale grado di accuratezza. In particolare, un APK di controllo si limita ad includere librerie mentre gli altri tre ne fanno anche utilizzo, ciò è utile per capire in quale misura è importante l'uso delle funzionalità di libreria per il rilevamento. Dai risultati ottenuti, si può notare che è importante un utilizzo significativo delle librerie perché il tool

[1] `https://github.com/dennnanni/adb-apk-downloader`.

Tabella 4.1: Alcune librerie del set di input.

Libreria	**Descrizione**
BouncyCastle	Funzionalità di crittografia
Braintree	Integrazione di pagamenti in app
Coil	Caricamento di immagini
Dagger	Supporto alla dependency injection
Gson, Moshi	Gestione di dati in formato **JSON**
Jackson	Gestione di dati in formato **XML**
Retrofit	Interfaccia per chiamate **HTTP**
Lottie	Inserimento di animazioni
Zxing	Lettura e creazione di codici a barre
Glide, Picasso	Caricamento ottimizzato e *caching* di immagini
OkHttp, Ktor	Gestione della comunicazione **HTTP**
Okio	Supporto alla gestione dei dati
Guava	Gestione estesa di dati e azioni di input/output
Vungle	Inserimento di annunci in app

le identifichi in maniera sufficientemente accurata, tuttavia ci sono dei casi in cui vengono individuate con lo stesso grado di affidabilità anche se non utilizzate. Si noti che i quattro **APK** non contengono le stesse librerie in quanto due di essi, *BudgetBuddy*[2] e *WeatherApp*[3], sono stati prelevati da GitHub.

Tabella 4.2: Risultati con applicazione di controllo senza utilizzo.

Libreria inclusa	**Rilevata dal tool**	**Versione**	**Accuratezza**
Gson 2.8.0	Sì	2.8.0	56,18%
Retrofit 2.9.0	No	-	-
OkHttp 4.11	No	-	-

Nella tabella 4.2 si nota la mancata rilevazione di due **TPL** su tre, mentre nel caso con implementazione, mostrato nella tabella 4.3, si aggiunge una corrispondenza, seppur con affidabilità inferiore al 50%.

Una parte importante per l'utilizzo di LibScan per la rilevazione sono i due valori soglia che permettono di filtrare le corrispondenze in due fasi distinte del processo. Inizialmente erano impostate a 0.7 per il primo passaggio e 0.85 per il secondo, ma sono state modificate

[2]BudgetBuddy su GitHub: `https://github.com/dennnanni/BudgetBuddy`.
[3]WeatherApp su GitHub: `https://github.com/dev-aniketj/WeatherApp-Android`.

Tabella 4.3: Risultati con applicazione di controllo con utilizzo.

Libreria inclusa	**Rilevata dal tool**	**Versione**	**Accuratezza**
Gson 2.8.0	Sì	2.8.0	56,24%
Retrofit 2.9.0	Sì	2.9.0	48,31%
OkHttp 4.11	No	-	-

Tabella 4.4: Risultati per *BudgetBuddy*.

Libreria inclusa	**Rilevata dal tool**	**Versione**	**Accuratezza**
Coil 2.3.0	Sì	2.3.0	79,88%
Ktor 2.3.8	Sì	2.3.5	80,38%

Tabella 4.5: Risultati per *WeatherApp*.

Libreria inclusa	**Rilevata dal tool**	**Versione**	**Accuratezza**
Lottie 5.2.0	Sì	5.2.0	77%

a 0.3 e 0.4, dato che con percentuali di accuratezza più alte il tool non rilevava librerie. In questo modo si può avere una stima delle TPL presenti anche se con un margine di errore maggiore. I valori sono stati scelti in base ai risultati ottenuti con le applicazioni di controllo, in modo da raggiungere un compromesso tra accuratezza e rilevazione.

Prima di sottoporre gli APK ottenuti dal Play Store al tool, è stata effettuata una ricerca manuale per individuare le librerie presenti secondo la struttura di cartelle, utilizzando Apktool per ottenere la gerarchia. Questo passaggio richiede diverso tempo data la necessità di effettuare il reverse engineering di ciascun'app, ma è servito per valutare i risultati ottenuti con LibScan. I tempi di esecuzione del tool variano in base a fattori come la dimensione dell'applicazione, il numero di librerie presenti, il grado di offuscamento, ma in media si aggirano sui 340 secondi per ogni applicazione, quindi circa 3 ore complessive, escludendo il tempo impiegato per convertire i JAR delle librerie in DEX. Considerando sia le tempistiche di reperimento che quelle di analisi, l'intera durata del processo è approssimabile alle 5 ore per solo 30 applicazioni.

Per quanto riguarda le applicazioni del Play Store, i risultati variano molto: per più della metà di esse non è stato possibile individuare alcuna libreria, mentre per le restanti si sono ottenuti risultati parziali, riuscendo a ottenere solo un quarto delle TPL trovate tramite ricerca manuale e, a volte, si aggiungono alcune non riscontrate. Il motivo di questa discrepanza potrebbe dipendere sia dalle similitudini tra le invocazioni, causando dei falsi positivi, sia dall'utilizzo della libreria come dipendenza di altre, includendola quindi nelle rilevazioni senza inclusioni dirette. Nella maggioranza dei casi, l'accuratezza finale non supera il 50%, con alcune eccezioni che raggiungono il 70%.

In generale, si delinea una predisposizione maggiore delle applicazioni sviluppate per il settore pubblico a essere più trasparenti all'analisi, come quelle di *Poste Italiane* o del Fascicolo Sanitario Elettronico, ma anche nel caso di alcune app di home banking si è ottenuto qualche risultato. Inoltre, le app attribuibili a grandi aziende sono quelle che hanno fornito meno corrispondenze. Ciò potrebbe essere dovuto al fatto che le librerie utilizzate sono state sviluppate internamente o che sono state modificate per adattarsi alle esigenze dell'applicazione, rendendo più difficile il rilevamento. Nel caso delle applicazioni di proprietà di *Meta*, ad esempio, guardando gli esiti dell'analisi manuale si riscontrano molti percorsi con package name di librerie proprietarie.

Alcuni risultati fanno pensare che il tool riesca realmente a superare le tecniche di offuscamento di tipo identifier renaming, ad esempio nell'app *PosteID*, come si può notare dalla tabella 4.6, poiché si ottengono riscontri per librerie inaspettate.

Tabella 4.6: Risultati per *PosteID*.

Libreria	**Rilevata manualmente**	**Rilevata con tool**	**Versione**	**Accuratezza**
BouncyCastle	Sì	Sì	1.57	46%
Retrofit	Sì	Sì	2.0.0	53%
Zxing	No	Sì	3.3.0	45%
Glide	No	Sì	4.14.0	56%

Se si guarda la gerarchia di cartelle nella figura 4.1, si ritrovano dei package con nomi composti da una sola lettera, che evidenziano la presenza di offuscamento.

```
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/i
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/j
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/k
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/k/i
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/l
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/m
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/n
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/o
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/o/v
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/o/w
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/o/x
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/o/y
posteitaliane.posteapp.appposteid/smali_classes2/n/a/a/o/z
```

Figura 4.1: Offuscamento di *PosteID*.

Non è possibile però affermare che questo valga in generale, poiché ci sono casi con offuscamento, mostrato nella figura 3.3 nella sezione 3.2.1, in cui il tool non rileva alcuna libreria.

Questi esiti non permettono di affidarsi completamente a LibScan, in quanto è chiaro che ci siano delle forti limitazioni, ma può essere comunque utile ai fini dello studio perché, nonostante le poche corrispondenze, si può avere un'idea delle librerie maggiormente

Figura 4.2: CVE mostrati da Maven Central per la libreria *Retrofit* versione 2.0.0.

utilizzate e riuscire a fare valutazioni sulla sicurezza delle stesse ricercando le vulnerabilità note.

Per valutare la sicurezza di una libreria, bisogna considerare anche i CVE relativi alle sue dipendenze, oltre quelli che la riguardano direttamente. Grazie alle informazioni di Maven Central, è facile risalire alle problematiche note di ciascuna versione.

Prendendo in esame sempre *PosteID*, si può vedere nella figura 4.2 che la versione 2.0.0 di *Retrofit* presenta 6 vulnerabilità note, di cui due sono direttamente correlate alla libreria, mentre le altre riguardano le dipendenze. Oltretutto, questa versione di *Retrofit* è stata pubblicata nel 2016 e sono disponibili numerose nuove versioni che non contengono tali problemi. Questo esempio mostra come sia importante tenere aggiornate le librerie per evitare problemi di sicurezza. Eccetto questa applicazione, tutte le altre contenenti *Retrofit* utilizzano delle *release* più recenti, che però presentano altre vulnerabilità relative alle dipendenze.

Ciò vale per la maggioranza delle librerie rilevate, le quali non sono aggiornate all'ultima versione disponibile, anche se sono segnalati CVE. In diversi casi, le vulnerabilità associate o relative alle dipendenze hanno un base score abbastanza alto in quanto riguardano problemi che possono essere sfruttati via rete senza alcun privilegio particolare e che impattano significativamente sui requisiti di confidenzialità, integrità e disponibilità, come nel caso del CVE-2021-3711 per *Retrofit* 2.0.0 con un base score di 9,1 su 10, oppure il CVE-2018-1000613 per *BouncyCastle* con punteggio di 9,8. Altre vulnerabilità, pur avendo un base score più basso, possono comunque essere sfruttate facilmente, ciò è indicato dal punteggio di exploitability[4] che spesso raggiunge il valore massimo 3,9.

Di tutte le applicazioni analizzate, nessuna di quelle di cui sono disponibili i risultati può dirsi priva di vulnerabilità, anche se non tutte sono critiche. In generale, si può affermare

[4]Punteggio che misura la facilità di sfruttamento della vulnerabilità.

che la sicurezza delle applicazioni mobile non è un aspetto di primaria importanza, poiché non si presta attenzione alle librerie utilizzate e alle loro vulnerabilità, nonostante siano facilmente risolvibili con l'aggiornamento delle stesse.

5 Conclusioni

La metodologia definita nell'ambito di questo studio si basa sui princìpi dell'analisi statica, il tipo di analisi che si concentra sull'esaminazione del codice sorgente di un'applicazione senza eseguirla, a cui si aggiunge una complicazione ulteriore dovuta al fatto che il codice di partenza è in formato bytecode, quindi non facilmente osservabile. Partendo dal reperimento degli APK da analizzare, tramite l'utilizzo di un emulatore di supporto e gli strumenti di debug Android Debug Bridge, si è proceduto con l'analisi dei file in diversi modi, studiando la struttura e il contenuto degli archivi decompilati e usando strumenti di analisi statica più avanzati come LibScan, quindi si è terminato con la ricerca dei CVE. La parte più critica del processo è l'identificazione delle librerie di terze parti, in quanto è necessario riconoscere i pattern che le caratterizzano, i quali possono essere estremamente simili tra una versione e l'altra, anche tra major e patch, e possono essere soggetti a tecniche di offuscamento e ottimizzazione che li oscurano. I risultati, seppur non perfetti, sono stati sufficienti a osservare la tendenza degli sviluppatori a trascurare la sicurezza, che spesso utilizzano librerie non sempre obsolete ma vulnerabili.

L'analisi statica è soggetta a diverse limitazioni che ne riducono l'accuratezza. La principale è la mancanza di esecuzione del codice, dato che diversi metodi di offuscamento si basano su tecniche che applicano modifiche che devono poi essere ripristinate in fase di esecuzione, quindi elementi cifrati non possono essere analizzati correttamente prima della fase di runtime.

Per ovviare ai problemi direttamente collegati alla staticità della tecnica, si potrebbe pensare di utilizzare un approccio dinamico o ibrido. L'analisi dinamica consiste nel mettere in esecuzione l'applicazione su un emulatore in ambiente controllato per monitorare i suoi comportamenti per mezzo di log. Questo metodo è solitamente utilizzato per rilevare malware, ma potrebbe essere sfruttato per identificare le librerie di terze parti, per esempio isolando determinati comportamenti caratteristici di una libreria. Le azioni che potrebbero essere monitorate sono, ad esempio, chiamate ad API esterne, le classi e i metodi caricati in memoria, le chiamate di sistema. Alzaylaee et al. [39] discutono dell'ambiente da creare per l'analisi dinamica, proponendo un sistema che usufruisce di un emulatore Android eseguito in una sandbox intermedia personalizzata per intercettare dei log prodotti grazie all'*instrumentation* dell'APK in esame, ovvero l'inserimento di codice dedicato al logging di azioni specifiche. La categorizzazione dei log potrebbe essere un'operazione utile per identificare le librerie, sfruttando comportamenti tipici di esse.

Un approccio diverso, invece, potrebbe basarsi sull'ottenimento di dump della memoria **RAM** mentre l'app è in esecuzione, poi esaminati per riconoscere, anche in questo caso, pattern comuni; infatti, quando i dati sono caricati in memoria devono essere decifrati, altrimenti le prestazioni ne risentirebbero considerevolmente [40], ciò permetterebbe di superare l'offuscamento che si avvale di decifratura runtime. Questo metodo è più intricato perché richiede un sistema con modifiche complesse, un dispositivo rooted, un kernel modificato, dato che l'accesso integrale alla memoria **RAM** non è un'operazione permessa a livello utente [41], e una conoscenza approfondita di molti aspetti di basso livello che riguardano la memoria e l'organizzazione delle informazioni in essa contenute.

In tutti gli scenari, però, l'unico metodo per garantire la rilevazione corretta della versione di una libreria è utilizzando dei confronti, poiché anche nel caso dinamico si deve ricorrere a pattern e comportamenti che possono essere soggetti a corrispondenze errate. Inoltre, il tempo di attesa continua a essere consistente anche nelle tecniche dinamiche perché è necessario eseguire l'applicazione per un certo periodo di tempo per rilevare i comportamenti tipici delle librerie, oltre al tempo impiegato per l'instrumentation, l'installazione, l'analisi dei log e i confronti.

Entrambe le tecniche potrebbero essere ampliate inserendo degli algoritmi di *machine learning* (**ML**) per la classificazione, come nel caso di *DroidDolphin* [42], in cui la penultima fase di analisi effettua un'estrazione di *features*, poi inserite in fase finale nel contesto del **ML**.

La difficoltà nell'estrapolare informazioni dagli **APK** è un'arma a doppio taglio, poiché da un lato non si può valutare la sicurezza dell'applicazione in maniera semplice, dovendo ricorrere a tool la cui affidabilità può essere messa in discussione, dall'altro è considerabile un vantaggio per l'utente, dato che potenziali attaccanti dovranno fare lo stesso sforzo per trovare informazioni utili per un attacco. La possibilità di oscurare le informazioni può essere anche un disincentivo all'uso di librerie aggiornate e sicure, visto che non è facile condurre degli studi approfonditi. Alcune delle applicazioni analizzate in questo studio operano in ambiti in cui la riservatezza delle informazioni e la continuità di servizio sono fondamentali, perciò sarebbe auspicabile che l'aspetto della sicurezza venisse tenuto in maggior considerazione.

In conclusione, nonostante il cospicuo numero di tentativi fatti negli anni, la rilevazione delle versioni di libreria è un problema ancora non risolto e che richiede ulteriori ricerche per essere affrontato in modo più accurato, considerando tutti i fronti di analisi. Questa lacuna compromette la capacità degli utilizzatori di valutare correttamente la *software supply chain*, un aspetto sempre più cruciale secondo le normative e le certificazioni di settore. Grazie alla rapida evoluzione nell'ambito dell'intelligenza artificiale e del machine learning potrebbero essere sviluppati nuovi metodi più efficaci, migliorando le rilevazioni e raffinando la ricerca, ma l'accuratezza della rilevazione rimarrà un aspetto critico.

Ringraziamenti

Al termine di questo percorso triennale, desidero ringraziare chi mi ha sostenuto e accompagnato in questa avventura, permettendomi di raggiungere un traguardo importante. Un ringraziamento particolare va al mio relatore, il professor Gabriele D'Angelo, per la gentilezza e la disponibilità con cui mi ha seguita durante il progetto e la stesura della tesi.

Bibliografia

[1] A. Turner, *How Many Smartphones Are In The World?*, mag. 2024. indirizzo: https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[2] Mag. 2024. indirizzo: https://gs.statcounter.com/os-market-share/mobile/worldwide/.

[3] A. Turner, *How Many Apps In Google Play Store?*, apr. 2024. indirizzo: https://www.bankmycell.com/blog/number-of-google-play-store-apps/.

[4] H. Wang e Y. Guo, «Understanding Third-Party Libraries in Mobile App Analysis», in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 515–516. DOI: 10.1109/ICSE-C.2017.161.

[5] M. Lieshout, «The Value of Personal Data», vol. 457, mag. 2015, pp. 26–38, ISBN: 978-3-319-18620-7. DOI: 10.1007/978-3-319-18621-4_3.

[6] M. Backes, S. Bugiel e E. Derr, «Reliable Third-Party Library Detection in Android and its Security Applications», in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 356–367, ISBN: 9781450341394. DOI: 10.1145/2976749.2978333. indirizzo: https://doi.org/10.1145/2976749.2978333.

[7] P. Salza, F. Palomba, D. Di Nucci, A. De Lucia e F. Ferrucci, «Third-party libraries in mobile apps», *Empirical Software Engineering*, vol. 25, n. 3, pp. 2341–2377, mag. 2020, ISSN: 1573-7616. DOI: 10.1007/s10664-019-09754-1. indirizzo: https://doi.org/10.1007/s10664-019-09754-1.

[8] E. Derr, S. Bugiel, S. Fahl, Y. Acar e M. Backes, «Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android», in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2187–2200, ISBN: 9781450349468. DOI: 10.1145/3133956.3134059. indirizzo: https://doi.org/10.1145/3133956.3134059.

[9] *Application Fundamentals*, Android, ott. 2023. indirizzo: https://developer.android.com/guide/components/fundamentals.

[10] T. Preston-Werner, *Semantic Versioning 2.0.0*. indirizzo: https://semver.org.

[11] *Structure of an Android App Binary (.apk)*, set. 2023. indirizzo: https://www.appdome.com/how-to/devsecops-automation-mobile-cicd/appdome-basics/structure-of-an-android-app-binary-apk/.

[12] *Enable multidex for apps with over 64K methods - About the 64K reference limit*, Android, mag. 2024. indirizzo: https://developer.android.com/build/multidex#about.

[13] *About Android App Bundles*, Android, mag. 2024. indirizzo: https://developer.android.com/guide/app-bundle.

[14] *Platform architecture*, Android, mag. 2024. indirizzo: https://developer.android.com/guide/platform.

[15] *Application Sandbox*, Android, apr. 2024. indirizzo: https://source.android.com/docs/security/app-sandbox?hl=en.

[16] A. Sinicki, *Root Android: Everything you need to know!*, mag. 2024. indirizzo: https://www.androidauthority.com/root-android-277350/.

[17] *On-device protections*, apr. 2024. indirizzo: https://developers.google.com/android/play-protect/client-protections?hl=en.

[18] Y. Elsantil, «User Perceptions of the Security of Mobile Applications», *Int. J. E-Services Mob. Appl.*, vol. 12, n. 4, pp. 24–41, ott. 2020, ISSN: 1941-627X. DOI: 10.4018/IJESMA.2020100102. indirizzo: https://doi.org/10.4018/IJESMA.2020100102.

[19] *Vulnerabilities*, National Institute of Standards e Technology, ago. 2023. indirizzo: https://nvd.nist.gov/vuln.

[20] *CVE Program*. indirizzo: https://www.cve.org/.

[21] D. Iezzi, *googleplay-api*, accessed: 2024-04-03, 2020. indirizzo: https://github.com/NoMore201/googleplay-api.

[22] *Android Debug Bdrige (adb)*, Android Developers, 2024. indirizzo: https://developer.android.com/tools/adb?hl=en.

[23] *Intents and intent-filter*, Android, giu. 2024. indirizzo: https://developer.android.com/guide/components/intents-filters.

[24] *Handling Android App Links*, Android, giu. 2024. indirizzo: https://developer.android.com/training/app-links.

[25] *APKEditor*, mag. 2024. indirizzo: https://github.com/REAndroid/APKEditor.

[26] E. Chikofsky e J. Cross, «Reverse engineering and design recovery: a taxonomy», *IEEE Software*, vol. 7, n. 1, pp. 13–17, 1990. DOI: 10.1109/52.43044.

[27] *APKTool*, gen. 2024. indirizzo: https://apktool.org/.

[28] D. Suganthi, J. Paulose, N. Girikumar, M. Sundaramoorthi e T. Tsultrim, «Smali Code Based Protection for Android Package», *International Journal of Innovative Science and Research Technology*, mar. 2017. indirizzo: https://ijisrt.com/wp-content/uploads/2017/04/Smali-Code-Based-Protection-for-Android-Package.pdf.

[29] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang e K. Zhang, «Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild», in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li e S. Zhu, cur., Cham: Springer International Publishing, 2018, pp. 172–192, ISBN: 978-3-030-01701-9. indirizzo: https://link.springer.com/chapter/10.1007/978-3-030-01701-9_10.

[30] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo e Y. Liu, «Automated Third-Party Library Detection for Android Applications: Are We There Yet?», in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 919–930.

[31] *ProGuard*. indirizzo: https://www.guardsquare.com/proguard.

[32] *DexGuard*. indirizzo: https://www.guardsquare.com/dexguard.

[33] Z. Ma, H. Wang, Y. Guo e X. Chen, «LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps», in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 653–656. indirizzo: https://ieeexplore.ieee.org/document/7883363.

[34] J. Zhang, A. R. Beresford e S. A. Kollmann, «LibID: reliable identification of obfuscated third-party Android libraries», in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 55–65, ISBN: 9781450362245. DOI: 10.1145/3293882.3330563. indirizzo: https://doi.org/10.1145/3293882.3330563.

[35] Yafei Wu and Cong Sun and Dongrui Zeng and Gang Tan and Siqi Ma and Peicheng Wang, «LibScan: Towards More Precise Third-Party Library Identification for Android Applications», in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, ago. 2023, pp. 3385–3402, ISBN: 978-1-939133-37-3. indirizzo: https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei.

[36] *National Vulnerability Database*, mag. 2024. indirizzo: https://nvd.nist.gov/.

[37] *Common Vulnerability Scoring System v3.1: Specification Document*, First. indirizzo: https://www.first.org/cvss/v3.1/specification-document.

[38] H. Dungriyal, *13 Essential Third Party Android Libraries for Efficient App Development*, set. 2023. indirizzo: https://medium.com/@dugguRK/13-essential-third-party-android-libraries-for-efficient-app-development-b406cc299ed8.

[39] M. K. Alzaylaee, S. Y. Yerima e S. Sezer, «Dynalog: an automated dynamic analysis framework for characterizing android applications», in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, 2016, pp. 1–8. DOI: 10.1109/CyberSecPODS.2016.7502337.

[40] C. Hilgers, H. Macht, T. Müller e M. Spreitzenbarth, «Post-Mortem Memory Analysis of Cold-Booted Android Devices», in *2014 Eighth International Conference on IT Security Incident Management and IT Forensics*, 2014, pp. 62–75. DOI: 10.1109/IMF.2011.8.

[41] J. Sylve, A. Case, L. Marziale e G. G. Richard, «Acquisition and analysis of volatile memory from android devices», *Digital Investigation*, vol. 8, n. 3, pp. 175–184, 2012, ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2011.10.003. indirizzo: https://www.sciencedirect.com/science/article/pii/S1742287611000879.

[42] W.-C. Wu e S.-H. Hung, «DroidDolphin: a dynamic Android malware detection framework using big data and machine learning», in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, ser. RACS '14, Towson, Maryland: Association for Computing Machinery, 2014, pp. 247–252, ISBN: 9781450330602. DOI: 10.1145/2663761.2664223. indirizzo: https://doi.org/10.1145/2663761.2664223.