Investigating Suspected Background Processes in Android Malware Classification through Dynamic Automated Reverse Engineering and Semi-Automated Debugging

by

Laya Taheri

B.Sc. degree in Computer Engineering, Shahid Bahonar University of Kerman (SBUK), Iran, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor:	Ali A. Ghorbani, Ph.D, Computer Science
Examining Board:	Rongxing Lu, Ph.D, Computer Science, UNB, Chair
	Arash Habibi Lashkari, Ph.D, Computer Science, UNB
	Brent Petersen, Ph.D, Electrical and Computer Engineering, UNB

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April, 2020

© Laya Taheri, 2020

Abstract

Android malware detection is one of the enthusiastic research domains in recent years. Despite researchers' admirable attempts in malware detection, malicious applications keep becoming resistant every year. Attackers develop sophisticated Apps to conceal malicious intentions on the background to be tolerant against naive malware detection methodologies.

To fill the gap in the lack of background malware analysis, we present the novel 3-layered malware analysis framework. We designate the proposed framework with the assistance of automated reverse-engineering and dynamic semi-automated Debugging methods. Our APK repository samples are divided into two groups, based on the existence of particular background processes in their source files. We use two separate activation procedures that differ for each group. Here, we generate our Android malware captured dataset consisted of static features, such as permissions, Intents, and metrics and dynamic features, such as network traffic and background services. Finally, we utilize two machine learning models to evaluate our framework.

We have aggregated our APK repository samples from two resources, CICAndMal2017 [30]-CICInvesAndMal2019 [39] and Android Wake Lock Research [1]. Through the evaluation experiments of the proposed framework, we have succeeded in achieving 85% accuracy and 88% precision in classifying malware categories and benign samples with Random-Forest model.

[1] http://sccpu2.cse.ust.hk/elite/dataset.html

Dedication

I want to dedicate my thesis to my lovely family, my father Mohammadhossein Taheri, my mother Maryam Hejazigol, and my sister Mona Taheri, who always support me and never left my side. Particular gratitude to my significant other, Hooman Khosravi, whose words of encouragement accompanied me during my thesis journey.

Acknowledgements

I would like to express my sincere appreciation to my supervisor Prof. Ali Ghorbani, for his generous support, encouragement, and guidance during my thesis journey. It has been my honor to have the opportunity of collaborating with the Canadian Institute for Cybersecurity (CIC). I would like to express my thanks to my intelligence colleagues and my project manager, Dr. Arash Habibi Lashkari, for their supports.

Table of Contents

Abstract ii

Dedication iii

Acknowledgments iv

Table of Contents v

List of Tables x

List of Figures xii

Abbreviations xiv

1 INTRODUCTION 1

1.1 Introduction 1

1.2 Summary of Contributions 4

1.3 Thesis Organization 5

2 BACKGROUND AND LITERATURE REVIEW 7

2.1 Android OS 7

2.1.1 Android Architecture 8

2.1.2 Android User Interaction 9

2.1.3 Rooting Android 10

2.1.4 Background Services [8] 11

2.1.4.1 WakeLock 11
2.1.4.2 Alternative Background Tasks 12
2.1.4.3 GetSystemService 13
2.2 Android App Analysis 14
2.2.1 Static Analysis 14
2.2.1.1 Reverse Engineering 14
2.2.2 Dynamic Analysis 15
2.2.2.1 Sandboxing 16
2.2.2.2 Debugging 16
2.2.2.3 Instrumentation 16
2.3 Android Malware Behavioral Features 17
2.3.1 Permission-based Features 17
2.3.2 Intent-based Features 18
2.3.3 Traffic-based Features 19
2.3.4 Battery-based Features 20
2.4 Android Malware Analysis 21
2.4.1 Static Analysis 21
2.4.2 Dynamic Analysis 23
2.4.3 Battery Feature-based Analysis 24
2.5 Android Malware Available Analysis Platforms 25
2.6 Android Malware Available Datasets 26
2.7 Concluding Remarks 32

3 PROPOSED FRAMEWORK 33

3.1 Overview 34
3.2 Static Reversing Analyzer (SRA) 34
3.2.1 Virus Total Labelling 36
3.2.2 Extracted Background WakeLock List 36

3.2.3 Preprocessor . . . 38
3.3 Dynamic Debugging Analyzer (DDA) . . . 38
3.3.1 UI Input Testing . . . 39
3.3.2 Network Traffic Monitoring . . . 39
3.3.3 APK Resigning . . . 39
3.3.4 Installation Checking . . . 40
3.3.5 Extracted WakeLock List . . . 40
3.3.6 Pure Capturing . . . 40
3.3.7 Suspected Capturing . . . 42
3.4 Learning Classifier (LC) . . . 44
3.4.1 Parsers . . . 45
3.4.1.1 Static Feature Extractor . . . 45
3.4.1.2 Dynamic Feature Extractor . . . 45
3.4.2 Machine Learning Methods . . . 45
3.5 Concluding Remarks . . . 46

4 IMPLEMENTATION 48

4.1 UML Diagram Types . . . 48
4.1.1 Structural UML - Component Diagram . . . 48
4.1.2 Behavioral UML - Sequence Diagram . . . 49
4.2 Function Call Graph (FCG) . . . 51
4.3 Configurations and Setups . . . 52
4.3.1 Android Device . . . 52
4.3.2 Droidbot . . . 54
4.3.3 SSH Connection . . . 54
4.3.4 APK Resigning . . . 55
4.4 Static Reversing Analyzer (SRA) . . . 55
4.4.1 Tools . . . 56

4.4.2 Coding Details . . . 57
4.5 Dynamic Debugging Analyzer (DDA) . . . 57
4.5.1 Tools . . . 57
4.5.2 Coding Details . . . 60
4.6 Learning Classifier (LC) . . . 60
4.6.1 Tools . . . 60
4.7 Concluding Remarks . . . 61

5 Experiments and Results **64**
5.1 Data Repository . . . 64
5.1.1 Data Distribution . . . 65
5.2 Evaluations . . . 67
5.2.1 Experimental Results . . . 67
5.3 Results Comparison . . . 69
5.4 Concluding Remarks . . . 71

6 Conclusions and Future Works **72**
6.1 Conclusions . . . 72
6.2 Future Works . . . 73

Bibliography **82**

A GNU Debugger (GDB) **83**
A.1 GDB Installation Procedure . . . 83
A.1.1 Installing GDBserver on the phone . . . 84
A.1.2 Running GDB and GDBserver . . . 84

B All WakeLock List Tables **88**

C Procedure of Rooting Android **91**

Glossary **94**

Vita

List of Tables

2.1 Details regarding currently available Android malware datasets 29

2.2 Comparison of the publicly available Android malware datasets . . . 31

3.1 The final list of background WakeLock signatures in Smali code format achieved through Javap 37

3.2 The final list of background WakeLock signatures in Smali code format achieved through IDA pro 41

4.1 Customer profiling setups for the Nexus 5 phone 53

4.2 Device profiling setups for the Nexus 5 phone 53

4.3 Configuration setups for using Droidbot 54

4.4 Certified passwords for resigning APKs 55

5.1 Number of benign samples that have been allocated to each *PureSamples* and *SuspectedSamples* group in Component 3.2 65

5.2 Number of malware samples that have been allocated to each *PureSamples* and *SuspectedSamples* group in Component 3.2 65

5.3 Distributions of the *SuspectedSamples* benign APKs at the four stages in Component 3.3 66

5.4 Distributions of the *SuspectedSamples* malware APKs at the four stages in Component 3.3 66

5.5 Number of each feature set extracted through the developed parsers explained in Section 3.4.1 68

5.6 The best results of the NN and RF classifications using specific variable parameters 69

List of Figures

2.1 Android architecture from elinux.org 9

2.2 Requesting *WAKE-LOCK* permission inside a manifest file from software.intel.com 11

2.3 Sample code for creating a WakeLock object in Android application from software.intel.com 12

2.4 The compiling versus decompiling sequences in App developing 15

2.5 An example of Intent usage in an App's manifest file 18

2.6 The battery log .txt output file related to an SMS malware sample from *bigge* family in the CICAndMal2017 Dataset 21

2.7 Radar visualization regarding the results of Table 2.2 31

3.1 Overview of the proposed background malware analysis framework . . 35

3.2 Flowchart of the *Static Reversing Analyzer* Component 36

3.3 Procedure of the Pure Capturing 3.3.6 module from Dynamic Debugging Analyzer component 3.3 41

3.4 Procedure of Suspected Capturing module 3.3.7 from Component 3.3. 47

4.1 Component view of the proposed framework for reversing and debugging Android Apps in Android malware analysis 49

4.2 Sequence diagram of Pure Capturing module 3.3.6 50

4.3 Sequence diagram of Suspected Capturing module 3.3.7 51

4.4 Function call graph related to the Suspected Capturing module 3.3.7. 52

4.5 Droidbot Shell Code 54

4.6 SSH TCP dump request Shell code for capturing Pcaps 54
4.7 Automated APK Resigning Shell code 56
4.8 Shell code of the Static Reversing Analyzer component 3.2 58
4.9 Python script for the Pure Capturing module 3.3.6 61
4.10 IDAPython script of the Main function for the Suspected Capturing module 3.3.7 63

5.1 distribution of final analyzed APKs used in the evaluation procedure, component 3.4 67
5.2 Comparison among the accuracy results of testing methods (Testing-20% and Cross-Validation) for MLP and RF models 68
5.3 Corresponding confusion matrices regarding the best results of the trained classifications in Table 5.6 69
5.4 ROC curves regarding the trained binary RF and NN classifications . 71

A.1 Overall connection structure between GDB and GDBserver 83
A.2 PS command to retrieve App PID 85
A.3 A suspended App by GDBserver waiting for a signal of a remote GDB 85
A.4 Running GDB on PC 86
A.5 Examples of commands of GDB 87

C.1 A Successful booting with Odin3 to Achieve the root access on the Nexus 5 Android device 92
C.2 The RootChecker App screenshot for testing the achieved root access on the device 93

List of Abbreviations

ACC	Accuracy
ADB	Android Debug Bridge
API	Application Programming Interface
APK	Android Application Package
App	Application
ART	Android Run Time
AUC	Area Under the ROC Curve
BPT	Breakpoint
CPU	Central Processing Unit
DDA	Dynamic Debugging Analyzer DVM Dalvik Virtual Machine
GDB	GNU Debugger
FCG	Function Call Graph
FP	False Positive
LC	Learning Classifier
ML	Machine Learning
MLP	Multi Layer Perceptron
NN	Neural Network
PID	Process ID
RF	Random Forest
ROC	Receiver Operating Characteristic
ROM	Read-Only Memory
SRA	Static Reversing Analyzer
SVM	Support Vector Machine
TP	True Positive
UI	User Interface
UID	User ID
UML	Unified Modeling Language
VT	VirusTotal

Chapter 1

INTRODUCTION

1.1 Introduction

Since the emersion of Android in 2008, it has become the center of attention among technologies. Android OS provides enormous functionalities to facilitate its customers' daily tasks such as banking, communication, Internet connection, and navigation. New capabilities appear with every latest version of Android OS. For privacy concerns, Android OS requires developers to use Intents and request read/write permissions for accessing personal information. Even though granting these accessibilities are necessary to launch innovative capabilities for Android users, they give opportunities to malicious applications in accomplishing their intentions. Through further analysis of Intents and requested permissions retrieved from APKs, we can investigate their acquired privileges.

The more the usage of Android devices is growing among people, the bigger its community is becoming. From the attacker's point of view, significant group members are supplying more probable vulnerable targets. Currently, we observe the existence of diverse abnormal Android applications among regular Apps. Any suspicious App with strange behavior might be malware. Researchers have divided malware into

a variety of categories, including Adware, SMSware, and Scareware. One way to categorize malware is according to its malicious behavior.

From the beginning of Android generations, studies on Android malware detection began. In previous years, several studies succeeded in gaining satisfactory results in detecting Android malware. Each of these studies focused on different methodologies. In general, we categorize them as follow: 1) *Signature-based detection*, defines signatures based on features obtained from APK files; 2) *Similarity-based detection*, compares a similarity score to a threshold value based on extracted features for identifying the malicious targets; 3) *Learning-based detection*, uses either static or dynamic features to train a decision model. Recently, the majority of researchers prefer the third methodology in their studies.

Researchers attempt to perceive the malicious activities of applications while examining their behavior. For observing App behavior, researchers gather a variety of characteristics to analyze in their proposed Android malware detection methodologies. These attributes are varied from static features acquired from an APK file such as permissions and Intents to dynamic features captured while running Apps on emulators and smartphones, such as network flows and API calls.

One of the challenges in dynamic feature analysis is performing user-interaction with Apps. In general, an installed Android App stays idle until the user starts interacting with it. Since Android applications require user-interaction to be activated, the researchers need to interact with them through simulated real-world scenarios. There are several tools available to perform App UI input testing.

Some malware are Single Mission. For example, when ransomware prompts a victim with blackmail, the life-cycle of ransomware would usually expire. As soon as the victim spots the malware, it troubleshoots the glitch permanently with possible mitigation solutions. On the other hand, some malware want to maintain its presence once installed on the phone without being detected. The Multi-Mission malware

conceals its actions on the background, away from a user's sight, and still is active even if the screen is off and the user is not working with the phone. Once Multi-Mission malware infects smartphones, it is more difficult to remove than Single-Mission malware since Multi-Mission malware works in the background.

In this study, we assume the suspected malicious App wants to hide its activities by conducting malicious tasks in the background. We believe advanced malware works in the background without the user's knowledge and exploits the device while it is in sleep mode. Hence, we observe captured background processes to inspect hidden background tasks of malware extracted from Apps' battery logs.

Through a captured battery log of an executed App, we would find several interesting details regarding the App's procedures, including its background tasks, App's services, and WakeLocks. Even though there are a few studies on battery features, they only focused on power usage and battery draining. In 2013, Hoffmann et al. [20] used battery **power usage** to distinguish malware and benign samples. Other previous research focused on detecting misused WakeLock usage that resulted in the **battery draining** [32], [27].

An analysis evaluation is trustworthy if it utilizes a well-known dataset. There are several static and dynamic Android malware datasets available for researchers' benefits. Here, we generate the captured logs while investigating suspected background processes.

To fill in the gap of background analysis of Android Apps, we provide a novel Android framework analysis based on the *Reverse-Engineering* and *Dynamic Semi-Automated Debugging* methods. In our three-layered framework, initially, we divide APKs through source code reverse-engineering approach into *SuspectedSamples* that conduct particular background malicious activities and *PureSamples* that have no background activities. To observe these groups' behavior, we have separate activation analysis methods. For the *SuspectedSamples* group, we activate Apps with

Semi-Smart UI Input Testing method. At the same time, we run Apps on a real smartphone and capture the battery logs with dynamic debugging techniques at three stages: after installation, at a possible reached breakpoint, and before the uninstallation. Also, we monitor Apps' network traffic flows during analysis. On the other hand, for the *PureSamples* group, we conduct a simple behavioral observation with *Semi-Smart UI Input Testing* while running on the phone. Finally, we design machine learning-based models for malware classification purposes based on retrieved static and dynamic features from the generated logs. The proposed analysis framework broadens our knowledge horizons regarding deceitful malware, which accomplishes its malicious intentions on the background without the user's awareness.

1.2 Summary of Contributions

The main contributions of this thesis can be summarized as follow:

- Proposing 15 criteria for comparing available Android datasets
- Collect malware samples that include critical hidden malicious processes, such as WakeLock and JobScheduler, in their Smali codes.
- Separate our APK repository into *PureSamples* with no background tasks and *SuspectedSamples* with specified background tasks.
- Trigger particular hidden tasks while observing their breakpoints with dynamic debugging methods.
- Collect more comprehensive battery logs according to our dynamic debugging techniques.

- Generate a unique Android dataset while activating repository samples in two separate ways depending on their group types, either *PureSamples* or *SuspectedSamples*.

- Detect and analyze background malware for the first time in Android malware classification.

- Accelerate background malware analysis by focusing only on specific background actions.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- First, Chapter 2 describes general knowledge and definitions regarding the research domain in Android malware analysis to clarify concepts in the thesis progress. Also, reviews various noticeable studies that have similar research domains, including Android malware analysis techniques and platforms. Later, it summarizes several available Android datasets from 2012 to 2019 and presents a comparison among them based on 15 criteria.

- Chapter 3 describes our three-layered analysis framework in continuous orders. First, it explains details regarding the first layer, Static Reversing Analyzer (SRA). Then, it reveals more information regarding the second layer, Dynamic Debugging Analyzer (DDA). Finally, it mentions the primary objectives of the third layer, Learning Classifier (LC).

- Chapter 4 outlines all the tools, software, and setup configurations regarding the implementations of each layer related to the proposed analysis framework.

- Chapter 5 presents the evaluation results according to the examination circumstances of the proposed thesis framework. It provides learning accuracy and visualizes results.

- Finally, Chapter 6 concludes the thesis by summarizing contributions and providing possible improvements and future works.

Chapter 2

BACKGROUND AND LITERATURE REVIEW

In this chapter, we describe prior knowledge regarding Android OS required in our research. Each section presents a comprehensive definition of a concept related to the Android App analyses that we will refer to later throughout the thesis.
Here, the chapter highlights several previous works in the Android malware research domain. First, the chapter divides some studies based on the types of features used in proposed Android malware detection systems. Second, some available malware analysis platforms are presented. Finally, we end with a review and comparison of currently available Android malware datasets.

2.1 Android OS

This section introduces prior knowledge regarding Android OS required in our research.

2.1.1 Android Architecture

Android has a hierarchical structure, and each layer has different responsibilities. Figure 2.1 shows the Android hierarchical architecture, consisting of the following layers:

- *Linux Kernel Layer*, includes device drivers and memory management.

- *Libraries Layer*, includes all the essential native libraries for launching tasks on the kernel layer. Android App developers avoid coding some general tasks by loading Native libraries already located in Android OS. In this case, they propose light-weight Apps for smartphones. Also, Android run time core libraries and Dalvik VM are located in this layer. Each App starts a separate DVM at run-time that executes the App in an isolated environment (sandbox) separated from other running Apps. The DVM, which is working only at App running time, aids a smartphone in handling its limited resources such as memory and battery life, and provides efficient performance.

- *Application Framework Layer*, includes all the application frameworks such as thread framework and activity framework. Each framework uses an event-driven programming model to integrate App coding. The event can be a user's actions, such as pressing buttons or sensor inputs. Android developers reuse these frameworks for their coding instead of working with native libraries. Android coding looks like an iceberg: the top is the developer's code, and the hidden part is the framework code. Each framework works with its corresponding native libraries.

- *Applications Layer*, includes manufactured basic applications that come with the device.

Any other developed third-party Apps execute over the top layer.

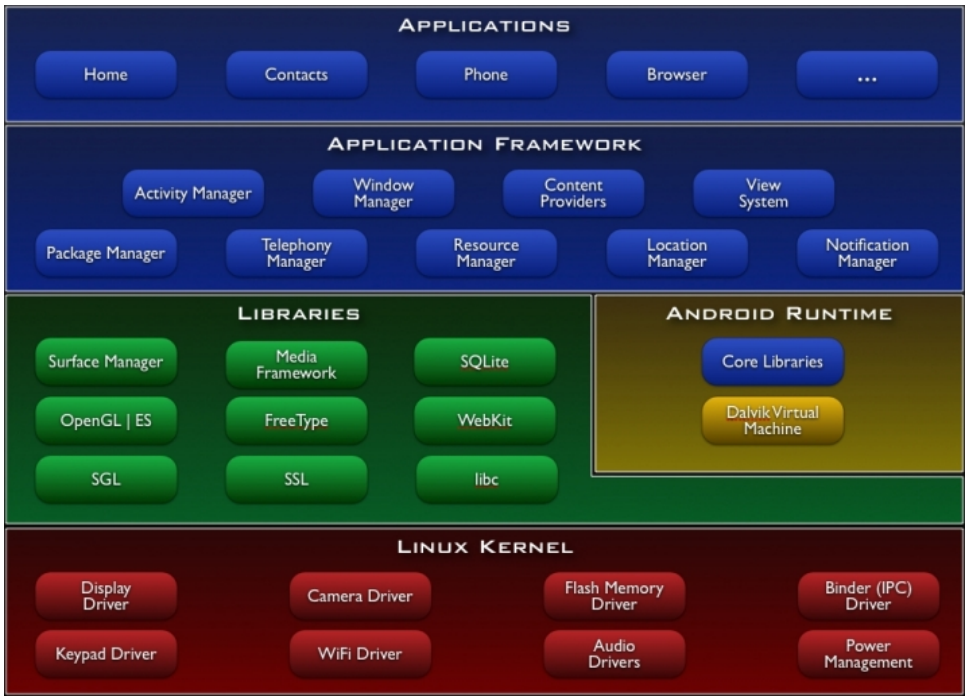


Figure 2.1: Android architecture from elinux.org

2.1.2 Android User Interaction

As we mentioned in Section 2.1.1 - App Framework Layer, Android Apps are using an event-driven model in which an App stays silent until a user input is received. Apps are demand-based and work as users' requests. Even though malware might act against users' requests or might perform extra tasks, it still might require the user's interaction to trigger its actions (except some background processes). When analyzing Android malware, researchers need to simulate user-interaction inputs to activate the desired actions of the malware sample. One of the challenges in capturing dynamic features is performing efficient automated UI interaction scenarios. The automated UI interaction methods are categorized into three general types:

- *Random UI Input*, is the process of randomly pressing any pixel on the phone screen such as Monkey tool [1] and MonkeyRunner tool [2]. It is a Black-Box UI testing method that does not require any App source code.

[1]https://developer.android.com/studio/test/monkey
[2]https://developer.android.com/studio/test/monkeyrunner

- *Semi-Smart UI Input*, follows the UI forward path by pressing relevant App buttons. It is a Gray-Box UI testing method that uses the App source code partially. Previous studies had provided some tools to conduct semi-smart user interaction such as SmartDroid [51], Droidbot [3] [31], and Dynodroid [4] [33].

- *Smart UI Input*, extracts the UI backward path by tracing the App's UI functions from the target UI element until the initial matching UI element, such as IntelliDroid [5] [47]. It is a White-Box UI testing method that requires the App's source code.

2.1.3 Rooting Android

To run a remote debugger on an Android device, we need to have root privileges on the device. Technically, rooting a device means flashing a rooted kernel taken from stock ROM which includes a security glitch [1]. A rooted device allows us to conduct tasks on the device system files that are typically bounded by a secure Android OS. There are two ways to have root access on Android: using rooting Apps and manually booting from scratch. Rooting Apps such as One-Click Root [6], Kingo [7], iRoot [8] usually work with the lower versions of Android with known exploit bugs. To do manual rooting, we need to flash Android with its rooted kernel from stock ROM. For more details regarding rooting Android Samsung Galaxy A5 and Nexus 5 devices, see Appendix C.

[3] https://github.com/honeynet/droidbot
[4] https://github.com/dynodroid/dynodroid
[5] https://github.com/miwong/IntelliDroid
[6] https://www.oneclickroot.com
[7] https://www.kingoapp.com
[8] https://www.malavida.com/en/soft/iroot/android/#gref

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="android.intel.sdp.WakeLock"
    android:versionCode="1"
    android:versionName="1.0" >
    ...
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    ...
</manifest>
```

Figure 2.2: Requesting *WAKE-LOCK* permission inside a manifest file from software.intel.com

2.1.4 Background Services [8]

Since smartphones come with limited power resources, by default, Android OS suspends CPU processes when no one is using the device and turns the screen off after a few seconds for efficient daily battery usage. However, some applications need to run operations continuously; as a countermeasure, Android designated WakeLock API that can run tasks even when the device is in sleep mode.

2.1.4.1 WakeLock

WakeLock is one way to have background processes on phones. WakeLock keeps the CPU awake and prevents the device from entering complete sleep mode. When an application wants to launch a task while the phone is in sleep mode, it must wake up the CPU by acquiring the *powermanager.WakeLock* API. Figure 2.2 shows that developers need to request the *WAKE-LOCK* permission within the AndroidManifest.xml file to utilize WakeLocks.

There are two types of WakeLocks: partial WakeLock and full WakeLock. The partial WakeLock keeps CPU running while the screen is off. A full WakeLock keeps both CPU and screen running. It is the developer's responsibility to stop a WakeLock process. Otherwise, it keeps running in the background and consumes battery and CPU resources. Developers need to create a WakeLock object and use *acquire()* and *release()* functions for controlling it. Figure 2.3 provides an example of the WakeLock code. Even though developers are cautious while acquiring WakeLocks, some tasks

```
01 public void onCreate(Bundle savedInstanceState) {
02     ...
03     mPowerManager = (PowerManager)
   getSystemService(Context.POWER_SERVICE);
04     ...
05     mWakeLock = mPowerManager.newWakeLock(mWakeLockState,
06                         "UMSE PowerTest");
07     if (mWakeLock != null) {
08         mWakeLock.acquire();
09     ...
10     }
11 }
12
13 protected void onDestroy() {
14     if (mWakeLock != null) {
15         mWakeLock.release();
16         mWakeLock = null;
17     }
18     ...
19 }
```

Figure 2.3: Sample code for creating a WakeLock object in Android application from software.intel.com

might not be completed in certain circumstances and cause WakeLocks not to be released. For instance, an application may allocate a WakeLock API for uploading an image, but the task will not complete if the server crashes. In these cases, using a *WakeLock.acquire* with timeout is helpful.

Besides WakeLock, Android proposes *WindowManager*, which has a *FLAG-KEEP-SCREEN-ON* variable to prevent the screen from turning off. *WindowManager* differs with the simple WakeLock in not requiring special permission, and whenever a user switches to another application, the *WindowManager* releases the WakeLock automatically.

2.1.4.2 Alternative Background Tasks

In addition to WakeLock, there are other ways to keep the phone awake as follows:

- *DownloadManger*, which performs a long-running HTTP download in the background. It takes charge of HTTP interactions and download failures.
- *SyncAdapter*, which synchronizes data transferring between an Android App

and an external server. It also coordinates synchronized operations across different Apps, where developers can aggregate their Android-based Apps' data and web-based App's data.

- *JobScheduler*, which enables developers to schedule their applications' background tasks asynchronously to improve the phone's performance and duration battery life duration. To use it, developers need to register a job and specify requirements and timing. The system schedules all jobs for execution at appropriate times.

- *AlarmManager*, which is for scheduling applications to wake CPU up in a specific time to perform some tasks. For instance, when an app wants to post a notification or set up an alarm at a particular time.

The *WorkManager* API in Android handles the mentioned alternative background processes, and developers need not be concerned with scheduling. The API decides to whether run background processes immediately or at an appropriate later time.

2.1.4.3 GetSystemService

GetSystemService retrieves the states of Android system-level services (such as LocationManager and WifiManager) for conducting application-level operations such as launching activities, broadcasting, and receiving Intents [9]. This method runs on background and acquires a WakeLock to keep updating states of the desired system services even if the device is in sleep mode. The *GetSystemService* function handles communications among system services and the application framework layer and works as a bridge among software (App layer) and hardware (kernel) layers [7].
These system-level services are allocated to a unique context instance, and they are reusable and permanent within an App. Hence, developers will obtain the same service by recalling them within the App code. The context instance keeps updat-

ing its corresponding requested service. For example, utilizing the *getSystemService(java.lang.String)* method for an alarm service enables the App to receive the updated state at the selected time [53]. Since this context object is created once and stays alive while the App is running, the developers should be careful while using related services [7].

2.2 Android App Analysis

This section presents details regarding techniques for analyzing Android Apps at different levels.

2.2.1 Static Analysis

A mobile application is released as an Android package (APK), which consists of the dex file, meta-data, manifest file. Android Apps are developed with Java and compiled into Java byte code (.class format) as an intermediate representation program. The byte code is machine independent that is transformed into a binary Dalvik native code (.dex format) by the Dalvik virtual machine (DVM), as explained in Section 2.1.1 - Libraries Layer. There are several tools to reverse an APK's compiled chain to generate Smali code, such as APKtool [9]. Smali code is the assembly language for Android Apps and a human-readable format of a .dex file.

2.2.1.1 Reverse Engineering

One way to analyze an APK statically is by inspecting its binary files. In Android static analysis, any backward attempts to create high-level source files from compiled APK are considered to reverse engineering methods. These methods are similar to the White-Box testing method since they require APK source files.

[9]https://ibotpeaches.github.io/Apktool/

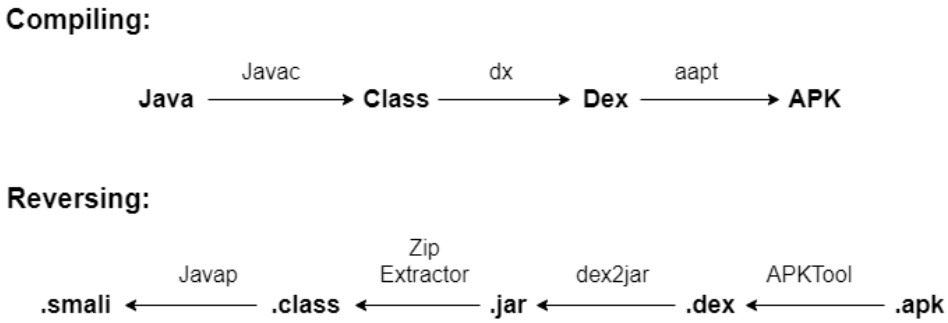


Figure 2.4: The compiling versus decompiling sequences in App developing

In static reverse engineering, researchers investigate APKs at different levels, including the Java, Byte code, Smali, and Binary levels. Analyses at each level have their advantages and disadvantages. For instance, malware developers usually use obfuscating techniques such as encrypting, optimizing, and packing to challenge researchers in reversing malware to higher code levels, which would cause data losses while decompiling. Hence, analyzing the source code at higher levels can not be as precise at lower levels.

On the other hand, it is harder to investigate dex instructions (low-level code) than Java code (high-level code) lines in a static code-based analysis since the machine code is not human-readable. However, it is more accurate to analyze the App source code closer to the machine level (dex/Smali). Figure 2.4 depicts the compiling versus decompiling orders in developing Android APKs.

2.2.2 Dynamic Analysis

In more precise malware analysis, researchers execute Apps on emulators or devices while monitoring their behavior dynamically. Dynamic analysis has gained Android malware analysts' interest. Dynamic analysis is categorized into three techniques [2]: Sandboxing, Debugging, and Instrumentation.

2.2.2.1 Sandboxing

In Sandboxing, researchers monitor an App behavior while running it in an isolated environment on an emulator or a real smartphone. Researchers capture Apps' dynamic behavior for further inspection. It is a Black-Box testing method in which analysis is carried on without investigating source codes.

2.2.2.2 Debugging

In dynamic debugging, researchers debug Apps' binary files while running them on real smartphones/emulators. It traces an App's behaviors in real time through setting breakpoints on assembly instructions and monitors their triggered states. There are some software programs such as Android Studio [10], GDB [11], IDA Pro [12], and OllyDbg [13] for debugging Apps' executable files. Debugging method is a White-Box testing method since it uses Apps' source codes. We provide more details regarding GDB in Appendix A and IDA Pro in Chapter 4.

2.2.2.3 Instrumentation

In this research area, Android analyzers insert scripts around critical regions in a binary file to change the App behavior at run time and bypass security points. It hooks functions and monitors their input and output arguments while the program is executing. It is an advanced level of dynamic analysis for triggering desired API calls while observing the results and tracing their data flows. In an automated instrumentation method, the registers of the App's source code are monitored when dynamically generated instruments feed into the binary executable code and manipulate the registers. There are some toolkits available for instrumentation using

[10] https://developer.android.com/studio/?gclid=Cj0KCQiAiZPvBRDZARIsAORkq7cCDUgwXUxOupOqs-8rAS6ikGvmqBVwnTR0ToYYSwmvAvcNLTEY79AaAhPIEALw_wcB

[11] https://www.gnu.org/software/gdb/

[12] https://www.hex-rays.com/products/ida/

[13] http://www.ollydbg.de/

White-Box testing techniques such as Frida [14] that assist in hooking and inputting functions and API calls to observe their input/output argument changes.

2.3 Android Malware Behavioral Features

Behavioral patterns of malware Apps can be investigated either with static features such as permissions and Intents, or dynamic features retrieved from battery logs and network flows.

2.3.1 Permission-based Features

To provide security, Android grants no privileges to any App and prompts the user to approve the requested permissions before installing Apps. Android requires developers to demand permissions regarding the facilities that the App wants to access. Developers use the *uses-permission* tag in the App's manifest to request permissions from OS.

At the beginning of the installation (for Android 5.1.1 and below) or run-time (for Android 6.0 and higher), Android prompts the user to accept the risk of installing Apps by approving the application's requested permissions. Even though this security mechanism is based on doubting installing Apps, it somehow trusts developers and users. Most of the time, users accept the requested permission without reading it. Also, not all the users are security experts familiar with the actual meaning of Apps' permissions. There is a high possibility that a simple user approves requested permissions without knowing its practical purposes.

Furthermore, Android is trusting developers by permitting them to ask for the least permissions. Permissions work with each other, and this collaboration can result in some consequences. For instance, granting Internet access and personal information

[14] https://frida.re/

```
- <receiver android:name="com.agilebinary.mobilemonitor.device.android.device.receivers.ScheduledActionReceiver">
  - <intent-filter>
      <action android:name="com.agilebinary.mobilemonitor.SCHEDULED_ACTION"/>
      <data android:scheme="alarm"/>
    </intent-filter>
  </receiver>
```

Figure 2.5: An example of Intent usage in an App's manifest file

access permissions apart causes no problem, but giving both could cause sensitive information leakage problems through connecting to the Internet. As a result, an attacker can ask a combination of grants to complete his/her missions.

Permissions are categorized into the following five types [11]:

- *Normal Permission*, poses minimal risk to the user's privacy and is approved automatically by Android OS, such as the permission to set the time zone.
- *Signature Permission*, is granted if the App that wants to use signature permissions is signed by the same certificate as the App that defines the signature permissions, such as *BIND-ACCESSIBILITY-SERVICE* permission.
- *Dangerous Permission*, can potentially affect the user's privacy or the device's regular operation, such as *READ-PHONE-STATE* permission.
- *Optional Hardware Permission*, appears with <*uses-feature*> tag in the manifest file that declares whether or not the device has requested a hardware feature, such as *android.hardware.camera* permission.
- *security Permission*, declares restricted access to a component or features and appears with <*permission*> tag in the manifest file.

2.3.2 Intent-based Features

Intent in Android defines an abstract description of an operation to be performed later, see Figure 2.5 for an example of Intent usage in an App's manifest file. Intents handle intra-application and inter-application communications in Android. Each

application runs in a specified Dalvik environment, mentioned in Section 2.1.1, and Intent is responsible for exchanging data between components of one particular application or between different applications. Intent filters specify which type of data to trade. Intent has three components: *action*, a passive task to be performed; *category*, a group that Intent belongs to; *data*, an input on which operate to.

There are two kinds of Intent, Explicit and Implicit. The Explicit Intent specifies what component or application to call. For instance, an App that opens a map direction only with Google Map App is using the Explicit Intent. On the other hand, the Implicit Intent does not call a specific App's name, and As a response, Android offers some applications based on the Intent filter type. An example of the Implicit Intent is an App that opens a video with three options of currently installed video player Apps on the phone.

Android's kernel uses the binder module for handling the Intent's data switching process. Whenever App A calls a component of App B, the binder checks data types regarding Intent filters and allocates memory space in App B for copying input data from App A.

Intent provides three fundamental use cases [10]:

- *Activity*, which is any entity of the user interface related to an App in the Android framework;
- *Service*, which allows developers to perform long-running operations in the background without a user interface;
- *Broadcast*, which is a system event as a message that Apps can receive.

2.3.3 Traffic-based Features

Some malware applications try to connect to a remote server to download an updated malicious patch or to leak the user's information. Monitoring in/out network flows

is an essential method to conduct malware behavior analysis.

There are some tools available to extract features of captured Pcap files such as CICFLOWMETER [15] [28], [13], which retrieves 84 elements from both in and out streams.

2.3.4 Battery-based Features

Providing an efficient and longer battery life with limited resources is one of the greatest successes achieved by Android devices. The captured battery logs include interesting data, which are outlined in the following.

There are some available tools for dumping battery logs, such as Android Debug Bridge (ADB) [16].In ADB, the *dumpsys batterystate* command captures the battery states of a connected device. The battery data are based on a unique User ID (UID) allocated to each installed application. Its output includes a history of battery-related events, battery statistics, approximate power usage per UID, and system components.

Figure 2.6 depicts a sample of battery states related to an SMS malware from the *bigge* (MD5= 46a4dae8d679775cdbde69b31d975f17) family collected from the CICAndMal2017 dataset [30]. The battery log provides related battery statistics, battery usage of each program running on the phone along with its UID, battery usage of each process with its PID, statistics since the last charge, estimated power use (mAh), all kernel wake locks, all partial wake locks, details of installed applications separated with unique UID including their package names, CPU usage, wake locks, scheduled jobs, and duration of started services.

[15] http://netflowmeter.ca

[16] https://developer.android.com/studio/command-line/adb

```
Battery History (0% used, 1368 used of 256KB, 40 strings using 4992):
                   0 (9) RESET:TIME: 2017-06-07-10-35-46
                   0 (2) 100 status=discharging health=good plug=none temp=256 volt=4275 +
                   0 (2) 100 user=0:"0"
                   0 (2) 100 userfg=0:"0"
                +111ms (2) 100 volt=4241
                +111ms (2) 100 volt=4219
              +17s999ms (2) 100 volt=4250
              +27s981ms (2) 099
              +30s677ms (2) 099 -screen
              +37s969ms (2) 099 volt=4282
            +1m07s671ms (1) 099 -wake_lock
            +1m13s568ms (1) 099 -running
            +1m13s770ms (2) 099 +running +wake_lock=u0a9:"CMWakeLock"
            +1m13s941ms (1) 099 -wake_lock
            +1m33s472ms (1) 099 -running
            +1m33s527ms (3) 099 +running +wake_lock=1000:"PhoneWindowManager.mPowerKeyWakeLoc
            +1m35s118ms (1) 099 +wifi_scan
            +1m35s693ms (1) 099 -wifi_scan
            +1m37s327ms (2) 099 -top=u0a90:"com.whatsapp"
            +1m37s328ms (2) 099 +top=u0a16:"com.android.launcher"
            +1m38s229ms (3) 099 volt=4230 -top=u0a16:"com.android.launcher"
            +1m38s865ms (2) 099 +wifi_full_lock +top=u0a95:"com.facebook.katana"
```

Figure 2.6: The battery log .txt output file related to an SMS malware sample from *bigge* family in the CICAndMal2017 Dataset

2.4 Android Malware Analysis

In this section, we review several previous studies using static features, including Permissions and Intents; and dynamic features, including Network Flows, API Calls, and Battery features in Android malware analysis. Due to our study domain, we dedicate a separate sub-section to Battery feature-based analysis. In the end, we review some available Android malware datasets.

2.4.1 Static Analysis

Static-based malware analyzers have examined permission features for some time. In 2015, researchers established APK Auditor [40], a server-side malware detector that consisted of three components: *Android Client*, *Central Server*, *Signature Database*. The Android Client was a user-interface App that enabled users to request the server to analyze an application or monitor the behavior of currently installed applications. The Central server was a link between clients and the database to conduct all the analyses and update the database. The Signature database stored the App's re-

quested permissions, services, and receivers as signature patterns. The APK Auditor reached 88.28% malware detection accuracy with 30% of a dataset, which included 6909 malware from Contagio [17], Genome [18], Drebin [19] sources, and 1853 benign from Google Play Store.

In 2017, Sokolova *et al.* [37] extracted permission-based graph patterns for benign applications and used them in an Android malware risk warning system. They analyzed their generated graph patterns with seven metrics, including Node degree, Weighted degree, Betweenness centrality, Closeness centrality, Page Rank score, Hub, and Authority scores. They trained their ML model with these metrics of nine thousand benign applications in 35 categories from Google Play Store and obtained 80.86% accuracy.

In some previous studies, researchers combined several static features to improve their analyzers. For example, Milosevic *et al.* [34] proposed two Android malware classification models trained with static permission and source code features. The permission-based model resulted in an F-score of 89% classification, and the source code-based model resulted in an F-score of 95.1% classification.

The AndroDialysis framework [18] extracted Explicit and Implicit Intents, Intent filters, and permissions for a Bayesian Network learner. It compared the impact of Intent features with permission features in Android malware classification. Based on this comparison, the top 10 Intent features between benign and malware differed more than the top 10 permission features for a dataset with 1846 Google Play benign and 5560 Drebin malware. Also, the Intent usage was more frequent than permission usage in malware samples. The AndroDialysis classified Android malware with 91% accuracy while using only Intent features, 83% accuracy while using only permission features, but increased up to 95.5% accuracy while using both types.

[17] http://contagiodump.blogspot.com/
[18] http://www.malgenomeproject.org
[19] https://www.sec.cs.tu-bs.de/ danarp/drebin/

Zhang *et al.* [49] utilized n-gram feature vectors data related to Intents and components. They trained the model in parallel with several n-gram sub-fingerprints. Most new learning-based analyzers employed a batch classifier method to retrain their models. A batch classifier is a simple model that needs to be retrained with all samples for adding newly updated ones. However, the authors in [49] utilized a parallel classifier which aided them in analyzing a large scale of malware samples and dealing with model retraining obstacles.

Some attackers try to mislead naive static-based Android analyzers. One of the recent attempts to defend against these attackers' attempts is KUAFUDET [5], a poisoning attack detector. The poisoning attack targets Android malware classifiers. In this attack, the malware developers add benign code and normal permissions to mislead static-based Android malware analyzers. KUAFUDET used the similarity score method to filter infected samples before passing them to the training phase of a malware classifier. To evaluate KUAFUDET, the authors manually infected their testing samples with the poisoning attack while only adding or removing a restricted number of benign features at three levels: weak, strong, and sophisticated.

2.4.2 Dynamic Analysis

To broaden Android malware analyses, researchers started to focus on dynamic features besides static features. For instance, Wang *et al.* [45] proposed an automated malware detection based on an SVM model and used it in an intelligent vaccine defense framework for malware immunization. They developed digital vaccine signatures and their antidote recovery scenarios. The digital vaccine deployment had three primary functions: 1) *System Monitoring*, which detected changes in specified registries and system files; 2) *System Backup*, restored a previous safe state whenever a malware infects the system; 3) *System Recovery*, which included reverse-engineering scenarios according to the malicious infections. Researchers evaluated their model

with 60 malware families from Contagio [20] and benign samples from Google Play. They extracted static source code features, such as byte signature and hash signature logs, and dynamic features, including network traffic and memory disk accesses of encrypted samples while running on the DroidBox [21] emulator.

Paper [50] proposed DroidSIFT, an Android malware classifier based on a weighted contextual API dependency graph. For a given App, they generated a weighted security-related API graph. They extracted a feature vector associated with each graph to use in training and testing sets.

In 2018, Karbab *et al.* [26] gathered sequences of API calls from Apps' DEX files for deep learning-based malware detection and family attribution purposes in MalDozer. The authors built a dictionary for mapping each series of API calls to its unique unified identifier. MalDozer was inspired by the Natural Language Processing (NLP) model according to similar definitions among the Android API call concept and the NLP models. Karbab *et al.* claimed that MalDozer is resilient against API call evolution and API call order changing over time.

2.4.3 Battery Feature-based Analysis

So far, not many researchers have studied battery features, and if they did, they mostly focused on power usage and misused WakeLocks. Back in 2012 [36], researchers developed their basic malware to test their anomaly detector model. The model used CPU consumption, battery usage levels, and the number of sending packets through WiFi as features. One of the first studies was the WakeScope platform [27] that continually tracked WakeLock acquiring and releasing at run time. They evaluated the fact that Android developers are responsible for releasing the acquired WakeLock whenever they are using it; otherwise, the battery will drain quickly. The WakeScope [27] was able to handle WakeLock misuses at Android run

[20] http://contagiodump.blogspot.com/
[21] https://droidbox.co.uk/how-to/tag/emulators/

time with low CPU overhead. In 2016, Liu *et al.* [32] examined real-world Android applications to understand how Android developers use WakeLocks. Based on their observation, they extracted eight patterns of real-world WakeLock misuses. They proposed a static analysis technique, called ELITE, to overcome two of the discovered misused WakeLock patterns.

2.5 Android Malware Available Analysis Platforms

Here, we introduce several available platforms for Android malware analyzing:
In a novel study in 2015, Boxify [4] was deployed as a new methodology for monitoring installed applications. Its idea was launching Boxify as the main application on the phone and running other Apps inside the Boxify. Boxify executes each application in an isolated process called the target, and it redirects its inter-process communications and system calls to Android OS. Researchers [4] evaluated the Boxify with 1079 Apps while interacting through the MonkeyRunner tool [22], a random UI event generator. Besides, they evaluated the overhead time of system calls and redirection of API calls. The results presented that the overhead time is not significant to affect overall performance.
In 2015, researchers [41] ran unmodified Android OS on modified CopperDroid emulators and captured system calls and Binder interactions. The CopperDroid intercepts a system call to retrieve its return values and analyze data dependency between its system calls. To evaluate the CopperDroid, authors designed an App simulation with 2900 samples from Genom [23], Contagio [24], and McAfee datasets, interacting through the MonkeyRunner tool [25]. According to the results, the simulation process aided in observing additional existed behavior and previously unseen behavior of

[22]https://developer.android.com/studio/test/monkeyrunner
[23]http://www.malgenomeproject.org
[24]http://contagiodump.blogspot.com/
[25]https://developer.android.com/studio/test/monkeyrunner

samples.

Glassbox [21] emerged as a grey box testing platform for dynamically capturing system calls, Java calls, and network traffic of Apps. Besides, Glassbox achieves system calls by interacting with the given application. The researcher made a smarter MonkeyRunner tool by dumping the hierarchy trees of App UI elements. This approach increased the average of basic code block's coverage up to 13.52% compared to a simple Monkey program. They presented the AndroCoverage [26] dataset for evaluations. The Glassbox platform is suitable for capturing Android App behavior, which can be used further in classification methods.

Malton [48] is an on-device malware detection method for ART. Malton monitored malware behavior at multiple Android architecture layers dynamically, including Java and Native codes, and it explored their possible bidirectional communication paths. This method can bridge the semantic gap between layers to track their data transmission. Compared to the other studies in the domain of Android malware detection, Malton can capture more sensitive operations and propose an effective path exploration method.

2.6 Android Malware Available Datasets

The first Android dataset emerged in 2012, and since then, several Android datasets have been published. Some of these datasets include static features, and some others installed applications on smartphones/emulators to capture their dynamic characteristics. This section provides a review of available Android datasets and compares them based on 15 criteria.

The Genome [27] [52] project is one of the first publicly available attempts in 2012. In this project, the authors collected 1260 malware samples from 2010 to 2011 of

[26] https://github.com/paul-irolla/androcoverage

[27] http://www.malgenomeproject.org

Android malware vendors. They used static analysis methods to scan permission lists and malicious source codes, track API calls. They executed Genome samples on real smartphones to examine the effectiveness of the existing anti-virus software. Afterward, the Drebin dataset [3] was created, which provided 5560 malware samples across 20 families and 123,453 benign samples from 2010 to 2012. To evaluate Drebin, researchers trained a classification system based on static features, including hardware components, requested permissions, App components, filtered Intents (extracted from manifest files, suspicious/restricted API calls, used permissions), and network addresses (extracted from disassembled code).

The Hacking and Countermeasure Research Lab (HCRL) [28] produced four Android malware datasets: AndroTracker, SAPIMMDS, Andro-Dumpsys, Andro-Profiler. In the AndroTracker dataset [25], they assumed that a single creator had written each malware family, and he/she has used his/her certification to develop malware Apps. They used a similarity score associated with creator information and other static features such as Intent, critical permissions, and suspicious API calls to classify malware samples.

SAPIMMDS [23] Dataset supported by the Korea Internet Security Agency (KISA), consists of 13 malware families with 906 samples and 1776 benign samples. Its samples were collected from March to December of 2014. They tracked suspicious sequences of API calls from byte code with memory dump techniques to extract API call patterns for certain malicious functionality.

The contribution in gathering Andro-Dumpsys dataset [22] combined malware attributes with Intent-based features for malware detection and classification. It includes 906 malware and 1776 benign samples. The authors adopted the serial number of certificates, suspicious API call sequences, permission distributions, Intents, and the usage of system commands for executing forged files as feature vectors. Based

[28]http://ocslab.hksecurity.net

on these feature vectors, they conducted profiling patterns according to the relationships between opcode and byte code (captured by memory dumping techniques) for each malware family. In the detection and classification engines on the server side, they calculated a similarity score among APK requests from the client side and their profiling patterns.

In the Andro-Profiler dataset [24], the authors used the concept of behavior profiling by extracting system calls and system logs during malware executions on an emulator. They developed a hybrid anti-malware system inside the client and server components.

In 2017, Lashkari *et al.* [29] provided the AAGM dataset with 1700 benign samples plus 426 malware samples from 2008 to 2016 in 12 families of three categories (benign, general, and adware). They installed samples on real smartphones and started performing user-interaction scenarios to capture network traffic. Their analysis method was based on machine learning algorithms. Also, they used the Droidkin project [19] for finding clustered relationships among Apps based on similar source code characteristics.

The AMD dataset[46] is another publicly available dataset since 2017 that includes profiles of Android malware. They gathered 405 Android malware samples of 4 categories within 71 families. Their systematic methodology lies in evaluating malware behavior with the analysis of malicious components based on their priorities.

The CICAndMal2017 [30] - CICInvesAndMal2019 [39] dataset tested Android malware and benign applications from several sources on real smartphones. The authors created a user profile for each smartphone. To provide this dataset, the researchers simulated a specific scenario for each of the malware categories. They collected two groups of features, namely continuous, such as Network Traffic, and discrete, such as Permissions and Battery Usages.

Researchers utilize available Android malware datasets for conducting evaluations

Table 2.1: Details regarding currently available Android malware datasets

Dataset Name	Published Year	Number of Benignware	Number of Malware	Captured Static Features						Captured Dynamic Features					Installed On
				States	Permission	Intent	Component	Certification	SourceCode	APICall	Network	Sys.Call	Infoflow	Log	
Genome [52]	2012	-	1260	No	Yes	No	Yes	No	Yes	No	No	No	No	No	-
Drebin [3]	2014	123,453	5560	No	Yes	Yes	Yes	No	Yes	No	No	No	No	No	-
AndroTracker [25]	2015	51,179	4,554	No	Yes	Yes	No	Yes	Yes	No	No	No	No	No	-
SAPIMMDS [23]	2016	1776	906	No	No	No	No	No	No	Yes	No	No	No	No	Emulator
Andro-Dumpsys [22]	2016	1776	906	No	Yes	Yes	No	Yes	Yes	Yes	No	No	No	No	Emulator
Andro-Profiler [24]	2016	8840	643	No	No	No	No	No	No	No	No	Yes	No	Yes	Emulator
Kharon [6]	2016	-	7	No	No	No	No	No	No	No	No	No	Yes	No	RealPhone
AAGM [29]	2017	1500	400	No	No	No	No	No	No	No	Yes	No	No	No	RealPhone
AMD [46]	2017	-	405	No	No	No	Yes	No	Yes	No	No	No	No	No	-
MalDozer [26]	2018	38,000	33,000	No	No	No	No	No	Yes	No	No	No	No	No	-
UCL [38]	2018	1.2M	-	No	No	No	Yes	No	Yes	No	No	No	No	No	-
CICAndMal2017 [30]	2018	1700	426	Yes	Yes	Yes	No	No	No	Yes	Yes	No	No	No	RealPhone

over academia or industrial malware analyzers. These datasets consist of a variety of extracted static data and captured dynamic behavior. Table 2.1 summarizes several available Android malware datasets from 2012 to 2019. Table 2.2 provides the comparison of the Android datasets reviewed in Table 2.1 based on our proposed 15 evaluation criteria as follows:

- C1, includes three tags: the static (S) tag represents that the corresponding dataset consists of extracted static features; dynamic (D) tag shows application installation either on emulators or smartphones while capturing its dynamic behaviors; both static and dynamic (B) tag states conducting both static and dynamic analyses for the generated dataset.

- C2, specifies whether the datasets with dynamic analyses had installed applications on real-phones or emulators.

- C3, clarifies whether the dataset has designed an analysis framework for the setup experiment or not.

- C4, states whether the corresponding dataset includes real-world malware samples or not.

- C5, expresses the presence of activation scenarios for the dynamic-based Android datasets.

- C6, shows that the corresponding dataset consists of multi capturing steps.

The capturing can start with installing the App while conducting activation scenarios and continues with another capturing after rebooting the device.

- C7, is the proof of initializing application labeling based on multiple trustable anti-virus sources.
- C8, demonstrates that the corresponding dataset consists of diverse malware categories and families.
- C9, stands for having a reasonable, balanced number between malicious and benign samples in the Android datasets.
- C10, shows that the dataset shares all of its information. Some datasets do not release all data regarding security and governmental issues.
- C11, stands for collecting samples from heterogeneous resources.
- C12, specifies that the dataset captured a variety of feature sets such as permission, Intents, states, log, API, network traffic.
- C13, declares that the dataset creators provided proper documentation for later potential researchers.
- C14, expresses the presence of malware taxonomy for providing an overall view of each malware categorical behavior.
- C15, demonstrates that the related dataset conducted its analyses with up-to-date applications.

In Table 2.2, the C2, C3, C5, and C6 columns are irrelevant to datasets generated by static analyses. The C9 column is unrelated to datasets that do not include benign samples. If datasets have both benign and malware Apps, they must have the real-world average distribution balance, which includes 80% benign and 20% malware to

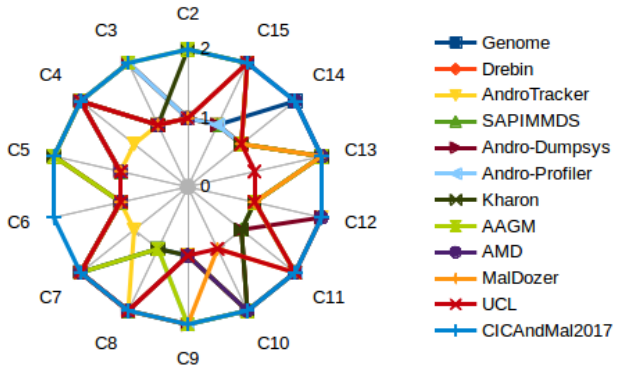


Figure 2.7: Radar visualization regarding the results of Table 2.2

Table 2.2: Comparison of the publicly available Android malware datasets

Year	Dataset	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
2012	Genome [52]	S	-	-	yes	-	-	yes	yes	-	yes	yes	no	yes	yes	no
2014	Drebin [3]	S	-	-	yes	-	-	yes	yes	no	yes	yes	yes	yes	no	no
2015	AndroTracker [25]	S	-	-	yes	-	-	no	yes	no	yes	yes	yes	yes	no	no
2016	SAPIMMDS [23]	B	no	yes	yes	no	no	yes	yes	yes	yes	no	no	yes	no	no
2016	Andro-Dumpsys [22]	B	no	yes	yes	no	no	yes	yes	yes	yes	no	yes	yes	no	no
2016	Andro-Profiler [24]	B	no	yes	yes	no	no	yes	no	yes	yes	yes	no	yes	no	no
2016	Kharon [6]	B	yes	no	yes	yes	no	yes	no	-	yes	no	no	yes	no	yes
2017	AAGM [29]	D	yes	yes	yes	yes	no	yes	no	yes	yes	yes	no	yes	no	yes
2017	AMD [46]	S	-	-	yes	-	-	yes	yes	-	yes	yes	yes	yes	yes	yes
2018	MalDozer [26]	S	no	no	yes	no	no	yes	yes	yes	no	yes	no	yes	no	yes
2018	UCL [38]	S	no	no	yes	no	no	yes	yes	no	no	yes	no	no	no	yes
2017	CICAndMal2017 [30]	B	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes

Legend:
C1: Type of data capturing, **static(S) or dynamic(D) or both(B)**.
C2: Utilizing **Real-Phone** devices instead of emulators.
C3: Having **network architecture** for the experiment set up.
C4: Examining **real-world malware** samples.
C5: Having **malware activation scenario**.
C6: Defining multiple states of **data capturing**.
C7: Having trust-able **fully-labeled malware samples**.
C8: Including **diverse malware** categories and families.
C9: Keeping **balance** between malicious and benign samples.
C10: Avoiding **anonymity** and preserving all captured data.
C11: Containing a **heterogeneous** set of resources.
C12: Providing a variety of **feature sets** for other researchers.
C13: For **meta-data**, includes a proper documentation.
C14: Including malware **taxonomy**.
C15: Being **up-to-date**.

set the C9 to (yes) element. All datasets have at least one kind of extracted feature sets, but the (no) element in the C12 column means that the dataset does not contain more than one type of feature set.

In Figure 2.7, we visualize the comparison of the aforementioned Android datasets from Table 2.2. In Figure 2.7, the value 1 is allocated to (no) and (-) elements, and value 2 is allocated to (yes) element. Figure 2.7 demonstrates that the more a dataset crosses over the value 1 for each criterion, the more it lacks in those criteria.

2.7 Concluding Remarks

Since 2012 the analysis of Android malware Apps has emerged, and researchers present remarkable studies every year. Initially, this chapter presents crucial information about Android OS to accomplish the thesis. Each section compares available methods related to an Android analysis concept that is referred to in the following chapters for additional clarification. In this chapter, we highlight details regarding previous novel studies in some domains, including Android malware detection, Android Analysis platform, and Android malware dataset. The next chapter presents the proposed Android malware analysis framework and its components' details.

Chapter 3

PROPOSED FRAMEWORK

Previous studies on battery analysis of mobile devices have mostly focused on battery draining and enhancing battery life in mobile devices [27], [32]. Researchers have neglected battery features in the Android malware analysis area, except Hoffmann *et al.* [20], who used App's power usage in malware detection, which was not efficient because benign samples use the same amount of battery levels as malware.

Android architecture is designated to go on sleep mode after a few seconds of nonuse. However, Android provides sleeping mode rescue plans by suggesting WakeLocks, described in Chapter 2 - Section 2.1.4.1. Android acquires WakeLocks for particular services through *GetSystemService* to provide background processes for an App. Sleep mode is a good time for hiding activities since the user is not working with the phone. Deceitful malware might exploit a phone through concealing activities by executing background processes on a device without the user's knowledge. This chapter is based on the assumption that the captured background processes are useful for inspecting malware's hidden background tasks extracted from Apps battery logs. Chapter 2 - Section 2.1.4.3 explains details of the *GetSystemService* concept in Android. Also, other types of Background processes that are analyzed in the research are explained in Chapter 2 - Section 2.1.4.2.

Here, the chapter presents the proposed analysis framework. The analysis framework reverses APKs to separate Apps, which are running background processes on Android by acquiring WakeLocks. The framework debugs the separated APKs while executing on a real smartphone, and interacts through a semi-smart UI input generator, introduced in Chapter 2 - Section 2.1.2, to trigger the observed background processes. The framework evaluates retrieved static and dynamic features in classification models. This chapter presents a comprehensive model that monitors App behavior through an in-depth analysis of APKs' source codes.

3.1 Overview

The proposed framework consists of three main components. Figure 3.1 provides an overview of the framework. The principal goal of these components are stated below:

- *1st Component - Static Reversing Analyzer (SRA)* separates APKs through reversing analysis based on the existence of particular signatures in source codes.
- *2nd Component - Dynamic Debugging Analyzer (DDA)* conducts behavioral analysis by capturing dynamic features with the assistance of debugging methods.
- *3rd Component - Learning Classifier (LC)* evaluates features retrieved from prior captured data in learning classification methods.

The output of each component is the resource of inputs for the subsequent parts.

3.2 Static Reversing Analyzer (SRA)

The first component reverses Android APKs collected from multiple sources consisting of both benign and malware samples. SRA focuses on looking up the allocated

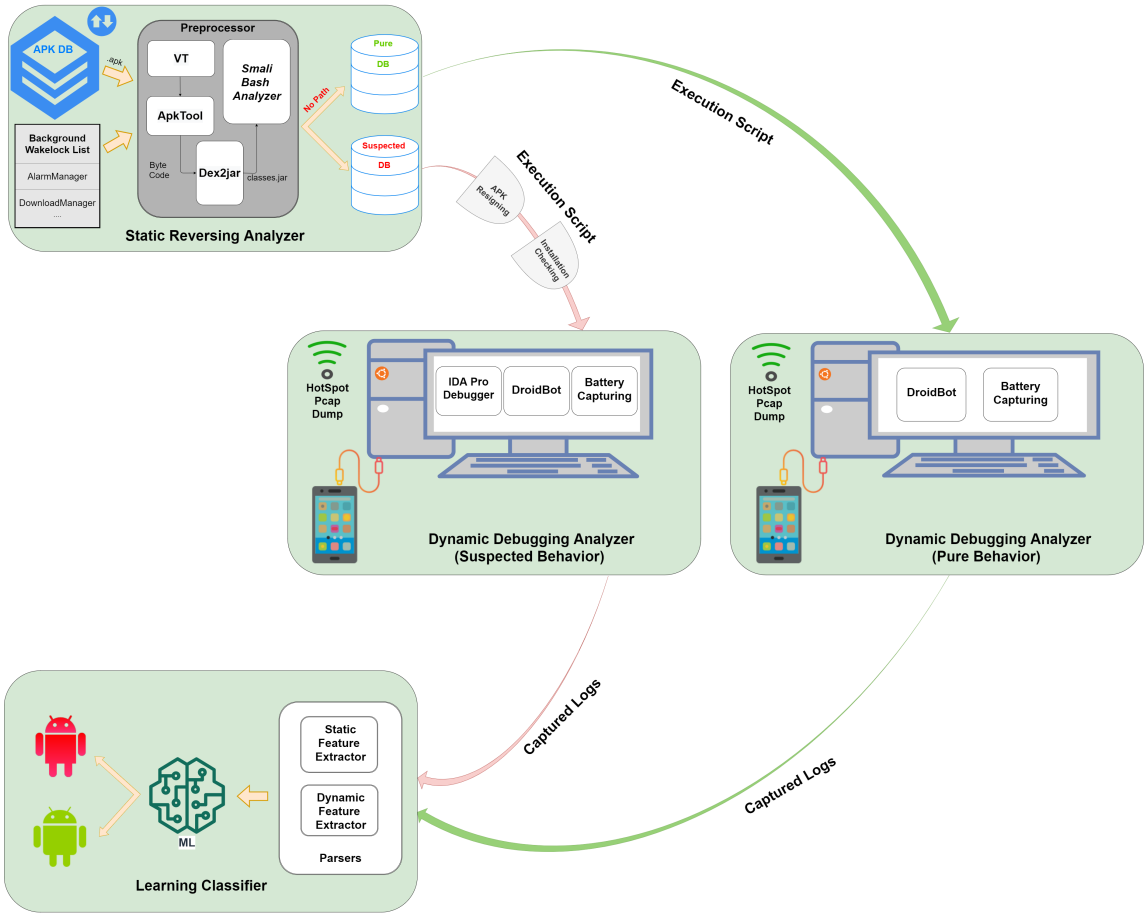


Figure 3.1: Overview of the proposed background malware analysis framework

signatures in APK files to distinguish them based on the results of the exploration. It investigates Smali codes of samples to extract the predefined Smali patterns related to the specified WakeLock list. SRA splits the labeled Apps' repository into two groups after it has observed the background processes in the source codes. One group, named the *PureSamples*, has no sign of the signatures in the Smali codes; the other group called the *SuspectedSamples* consists of the signatures in the APKs. Each group may contain both malware and benign samples from the repository. Figure 3.2 illustrates the sequential module procedure of the *Static Reversing Analyzer* component. This component completes its missions by modules below, provided

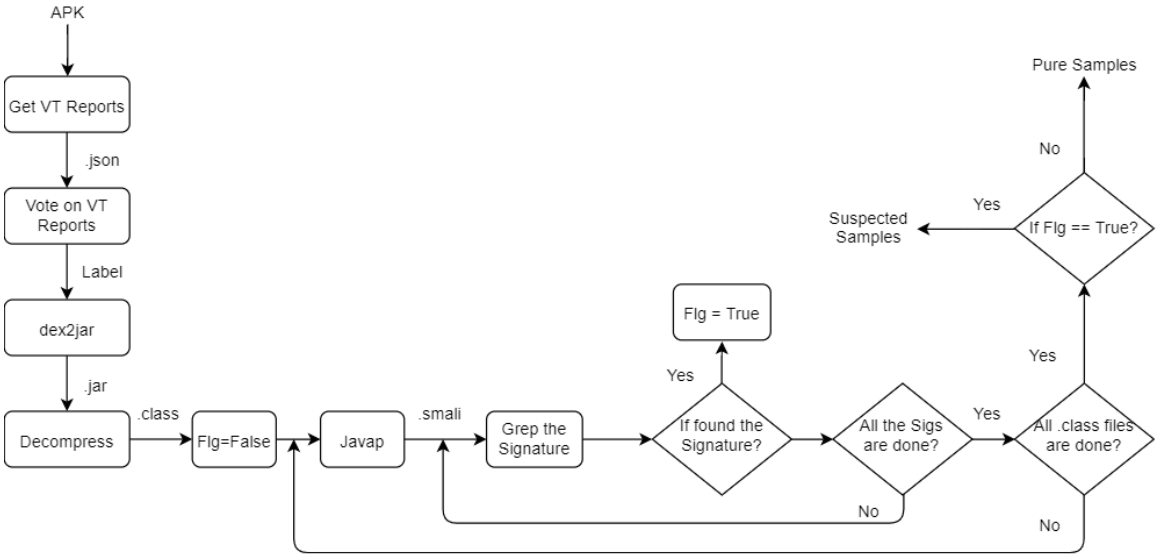


Figure 3.2: Flowchart of the *Static Reversing Analyzer* Component

in 3.2.1, 3.2.2, and 3.2.3.

Also, the directory path of each seen signature in an App's source code is restored along with the number of times they have been seen. These metrics are used as static features in the 3rd component 3.4.

3.2.1 Virus Total Labelling

It labels benign APKs, which are collected from the open App source [1], with VirusTotal (VT) private API to raise the assurance about benign ground-truth labels in the study. The module obtains VT reports and votes among them to label samples.

3.2.2 Extracted Background WakeLock List

Previously in the reading course studies, we have investigated captured battery states from the CICAndMal2017 dataset [30] and finalized a list of potential suspicious WakeLocks. We have selected several malware samples to research their decompiled Java and Smali codes. As a result, a list has been created that includes suspicious

[1] http://sccpu2.cse.ust.hk/elite/dataset.html

Table 3.1: The final list of background WakeLock signatures in Smali code format achieved through Javap

Background WakeLock Signatures - Javap Smali Version
checkcast // class android/app/AlarmManager
checkcast // class android/media/AudioManager
checkcast // class android/app/DownloadManager
checkcast // class GCM_LIB
checkcast // class android/location/LocationManager
checkcast // class PenetrateDictionariesLock
checkcast // class android/os/Vibrator
checkcast // class android/view/WindowManager

WakeLocks in the Smali code format.

In this module, the initial list includes the extracted WakeLock services from the captured battery logs related to malware samples. The battery logs are searched for the list's elements as keywords to find caller functions in the decompiled Java code of the App. Afterward, the module generates the Smali code of each APK by Javap [2] and looks up corresponding caller functions in Apps' Smali codes.

The extracted unique signatures matching WakeLock services are used as inputs in the Preprocessor Module 3.2.3. Table 3.1 shows the final list of background WakeLock signatures in Javap Smali code format. Comprehensive tables of these signatures in other coding forms (Java, Apktool Smali, Javap Smali, IDA Smali) are provided in Appendix B, which includes the source codes of some analyzed malware samples from the CICAndMal2017 [30] dataset.

One of the interesting observations while creating this WakeLock List is that all of the selected captured battery services from Table 3.1 ended up in a similar decompiled Java code function call, named *GetSystemService*, described in Chapter 2 - Section 2.1.4.3.

[2]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html

3.2.3 Preprocessor

In this module, a Shell script identifies the APKs that include particular WakeLock signatures from Table 3.1 in their source codes. Initially, the module decompiles the executable APK with dex2jar [3] tool, which generates a .jar file containing Java classes for each APK. Subsequently, the Preprocessor module decompresses .jar to .class files and utilizes Javap [4] to decompile .class to the Smali code format. The Preprocessor Module looks for the specified signatures from Table 3.1 in the final Smali files and separates the APKs that contain signatures (allocated to the *SuspectedSampels* group) from those with no signature (allocated to *PureSamples* group).

3.3 Dynamic Debugging Analyzer (DDA)

The second component conducts App behavioral dynamic monitoring. This component is a semi-automated dynamic analyzer and considers two different procedures for activating and analyzing samples from output groups of the previous Component 3.2.

It has two sub-components that can work in parallel. One component interacts and activates the *PureSamples* group by using a UI input testing tool while capturing battery logs.

The other sub-component activates and communicates with the *SuspectedSamples* group to trigger observed background processes retrieved from the specified WakeLock list within malware's source codes. This component uses dynamic debugging methods to set breakpoints on the specified background tasks in Apps' source code. While it monitors the breakpoints for a running App on a real smartphone, it runs a UI input testing tool to generate inputs for interacting with the App's UI. As soon as a breakpoint is reached, DDA captures the battery log of the corresponding

[3]https://github.com/pxb1988/dex2jar

[4]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html

suspected App.

The purpose of the Dynamic Debugging Analyzer component is to present a unique semi-automated model consisted of activation, UI interaction, battery behavioral capturing, network traffic monitoring, and debugging modules. Component 3.3 debugs real-world malware samples successfully. More details of developed modules in this component are described below.

3.3.1 UI Input Testing

Android applications are developed with an event-driven model and wait for user interaction as an input event to respond. This module utilizes a Shell code to work with a UI input generator tool for testing App samples while running on the phone. This tool uses a Semi-Smart User Input testing method, described in Chapter 2 - Section 2.1.2, that follows UI tree elements of each App.

3.3.2 Network Traffic Monitoring

Since some Apps try to synchronize with a remote server, this module simulates the Internet connection for the phone. It provides a HotSpot WiFi connection for connecting smartphones to the unprotected Internet. The module keeps track of all in and out packets on the HotSpot joint. An SSH port is opened on the remote HotSpot machine to listen to requests that start Pcap capturing. In this Module, a developed Shell code sends the TCPdump Pcap capturing query through the opened SSH connection while analyzing APKs.

3.3.3 APK Resigning

For attaching a debugger to an installed APK, the APK needs to be built in the debug mode version. APK Resigning module is a developed Bash script that decompiles each APK from the "SuspectedSample" group. Next, this module searches the App's

Manifest.xml file for the "android: debuggable" flag. If can not find any, it adds the flag with *True* value to the *Manifest.xml* file. Afterward, the script rebuilds and resigns the APK with our testing certificate.

3.3.4 Installation Checking

Some APK might not be compatible with the experimental device model and its Android version. This module is a developed Bash script that tests the APK repository against installation failures. Here, the module installs the APK, and retrieves the package and activity names, to launch App's *MainActivity*. If the APK is not consistent with the experimental environment, the module removes APK from the analysis repository.

3.3.5 Extracted WakeLock List

The DDA considers the IDA Smali code format in its debugging procedure. Since the *SuspectedSamples* group have some signs of signatures from Table 3.1 in the source codes, this module uses the IDA Smali form of these signatures through debugging analysis. Table 3.2 provides the Smali format of IDA assembly instructions related to similar signatures from Table 3.1. Later on, IDA Pro [5] debugger looks for the signatures of Table 3.2 in each APK binary file and sets breakpoints on their spots.

3.3.6 Pure Capturing

The Pure Capturing module analyzes the *PureSamples* group. Figure 3.3 depicts the sequence of its procedure. This module is developed as a Python script that starts with a parallel remote Pcap capturing thread, before installing an APK, until the end of execution of the App. The module obtains battery logs twice; the 1st time after App installation, and the 2nd time before the uninstallation. In the end, the

[5]https://www.hex-rays.com/products/ida/

Table 3.2: The final list of background WakeLock signatures in Smali code format achieved through IDA pro

Background WakeLock Signatures - IDA Smali Version
check-cast v0, <t: AlarmManager>
check-cast v0, <t: AudioManager>
check-cast v0, <t: DownloadManager>
const-string v3, aGcmLib # “GCM_LIB”
check-cast v0, <t: LocationManager>
const-string v2, aPenetratedicti # “PenetrateDictionariesLock”
check-cast v0, <t: Vibrator>
check-cast v0, <t: WindowManager>

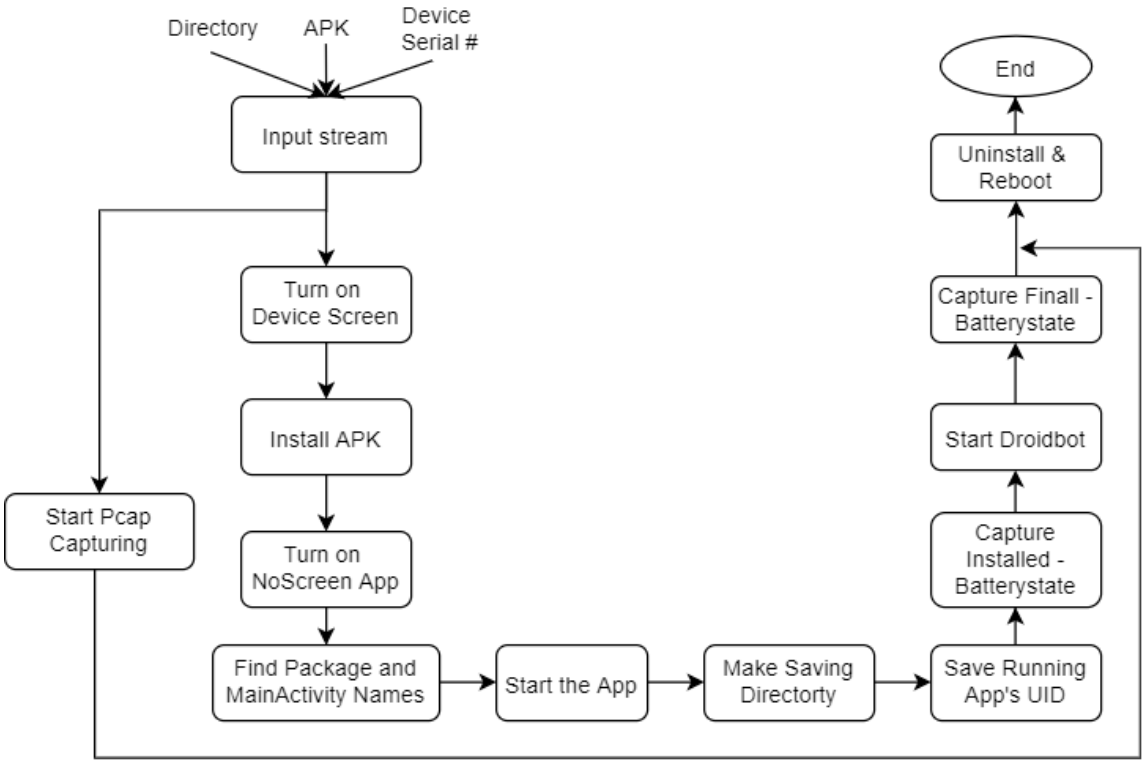


Figure 3.3: Procedure of the Pure Capturing 3.3.6 module from Dynamic Debugging Analyzer component 3.3

module uninstalls the APK and reboots the phone to its initial state. The Pure Capturing module includes UI Input Testing 3.3.1 and Network Traffic Monitoring 3.3.2 modules.

3.3.7 Suspected Capturing

Figure 3.4 illustrates the procedure of analyzing the "SuspectedSample" group inherited from the Static Reversing Analyzer 3.2 component. It is a semi-automated dynamic debugging method.

In the Suspected Capturing module, after enabling the "android: debuggable" flag and resigning suspected APKs, it runs the Installation Checking 3.3.4 module to ensure the resigned suspected APKs are compatible with the device. The procedure starts with initializing input variables and beginning a parallel Pcap capturing 3.3.2 on a remote SSH machine. Afterward, it forbids the device's screen to turn-off by running the "NoScreen" App. If the device is ready, it installs a suspected APK and retrieves its package and launchable *MainActivity* names. The next step sets breakpoints on all the observed assembly instructions corresponding to the *background WakeLock signatures* from Table 3.2 in IDA Smali format. The pseudo-code for setting breakpoints on targeted assembly instructions is shown in Algorithm 1.

Later on, this module restores the APK details along with App UID and attaches a debuggable process to the running App's PID on the real Android device. If hooked successfully, it initializes the UI Input Testing 3.3.1 module on a separate thread to interact with the App in parallel. The analysis method begins with continuing the attached process while it keeps interacting with App's UI and monitoring its settled up breakpoints with the IDA API's debugging capabilities. The debugging thread would interrupt the program with a returned debugging code, either if the breakpoints are hit or if the attaching timeout is reached. The module examines the returned debugging code to determine if the execution process has hit a breakpoint. If a breakpoint is met, the module captures a battery log of the device; otherwise, the debugging thread has suspended the continuing process because of the considered timeout. Algorithm 2 presents the pseudo-code for capturing battery logs based on the value of the returned debugging code.

Algorithm 1 Breakpoint Setting Algorithm that assigns BPTs on assembly instructions corresponding to the Background WakeLock Signatures of Table 3.2 developed in IDAPython coding

Ensure: B_List AND Num_BPT
1: $Num_BPT = 0$
2: $funcs_List = idautils.Functions()$
3: **for** $func_k$ in $funcs_List$ **do**
4: $\quad Insts_List = idautils.FuncItems(func_k)$
5: $\quad$ **for** $Inst_i$ in $Insts_List$ **do**
6: $\quad\quad$ **for** Sig_j in $Table$ 3.2 **do**
7: $\quad\quad\quad Num_Oprnd = len(Sig_j.split('\%', 3)) - 1$
8: $\quad\quad\quad$ **if** $Num_Oprnd == 1$ AND
9: $\quad\quad\quad\quad GetMnem(Inst_i) == Operator(Sig_j)$ AND
10: $\quad\quad\quad\quad GetOpnd(Inst_i, 0) == Operand(Sig_j, 0)$ **then**
11: $\quad\quad\quad\quad

At this stage, the module checks the final PID of the App; if it is not the same as the initial PID, the module drops the APK analysis because the App is probably a corrupted sample due to the decompiling procedure.

In the end, the program restores a final battery log, uninstalls the App, saves all the breakpoint details, and de-attaches the execution process. The program reboots the phone to its initial stage with recovered backup files. We will repeat the same procedure for the next APK sample from the *SuspectedSamples* group.

In the Suspected Capturing procedure, the modules above are used, including UI Input Testing 3.3.1, Network Traffic Monitoring 3.3.2, APK Resigning 3.3.3, Install Checking 3.3.4, and Extracted WakeLock List 3.3.5.

3.4 Learning Classifier (LC)

The last component is the Learning Classifier consisted of static and dynamic parsers and ML evaluation methods. The developed parsers aggregate unique features to be used in ML models. The learning models distinguish malware from benign samples according to the ground-truth labels. LC evaluates the Dynamic Debugging Analyzer 3.3 component in activating specified malicious background activities.

Algorithm 2 Switch Case Algorithm in IDAPython coding for capturing battery logs based on the value of the returned debugging code

Require: *input_code*
Ensure: *Reached_BPT_List*
1: $reached_ip = idaapi.get_ip_val()$ ▷ get instruction pointer
2: **if** $input_code == 16$ AND $reached_ip$ not in $Reached_BPT_List$ **then**
3: ▷ 16 is hit breakpoint code
4: $capture_battery(str(reached_ip))$
5: $Reached_BPT_List.append(reached_ip)$
6: **end if**
7: **if** $input_code < 0$ **then** ▷ zero is error code
8: $print(''$Attached Process is Suspended by Timeout!$'')$
9: **end if**
10: return Reached_BPT_List

3.4.1 Parsers

Each parser script has two essential parts; one is for retrieving unique features, and the other is for creating tables for APKs according to the unique features. Parsers are developed to extract static and dynamic features.

3.4.1.1 Static Feature Extractor

This module extracts two types of static features: permissions and Intents from the *Manifest.xml* file of each APK. Permission and Intent concepts in Android are introduced in Chapter 2 - Sections 2.3.1 and 2.3.2. Also, it collects metrics retrieved from the analysis of Component 3.2. The metrics are the numbers of matching signatures related to Table 3.1 observed in each App's source code.

3.4.1.2 Dynamic Feature Extractor

This module extracts dynamic features, including network traffic features (see details in Chapter 2 - Section 2.3.3) and background WakeLock services (see details in Chapter 2 - Section 2.1.4). The module achieves WakeLock characteristics based on each App UID from captured battery logs.

3.4.2 Machine Learning Methods

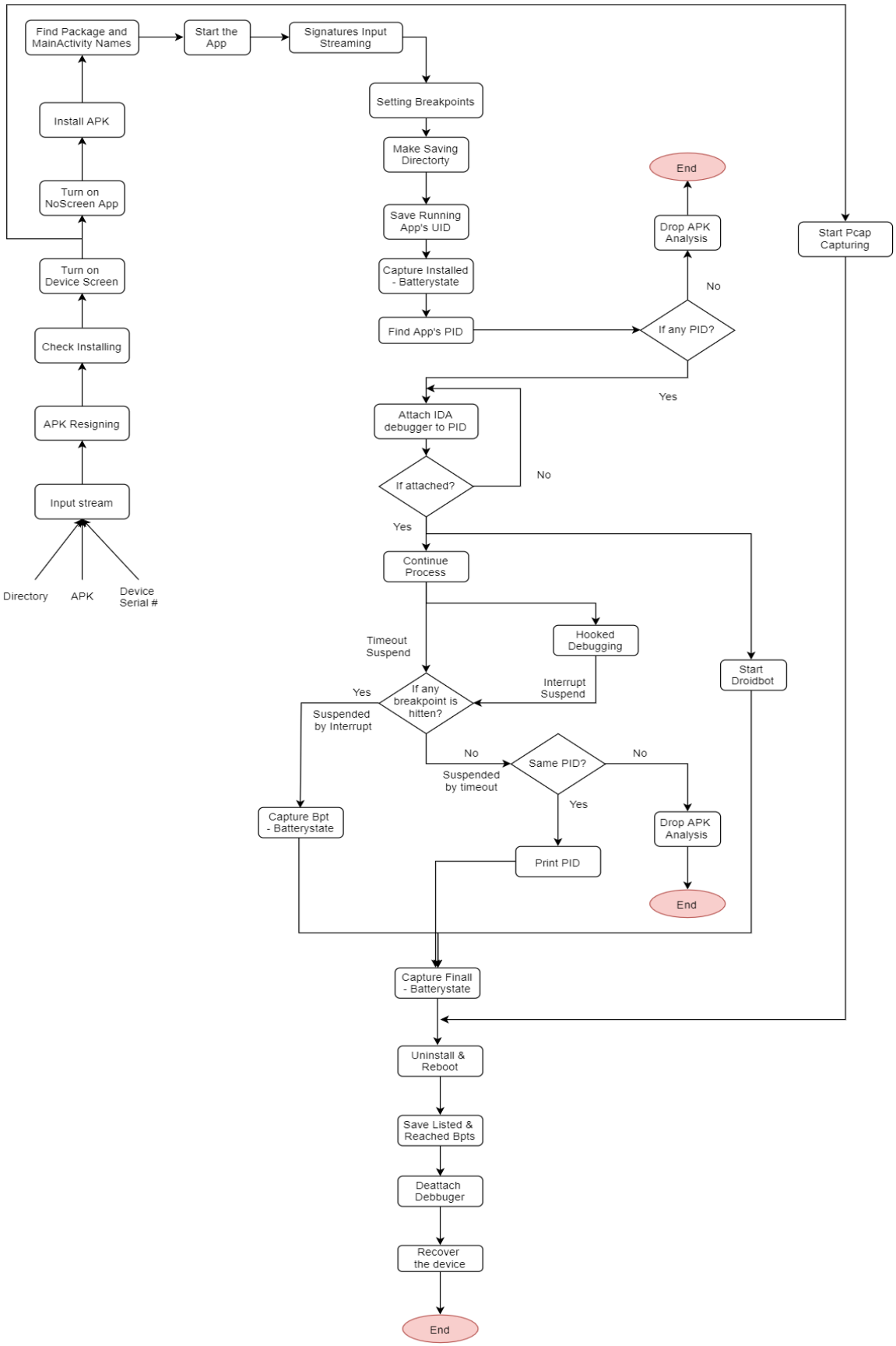
This module classifies Apps, according to their original labels based on the extracted features retrieved from the captured static source files and dynamic logs. The module uses Random Forest (RF) and Neural Network (NN) learning models. The potential high accuracy indicates the achievement in activating the selected background activities of malware samples.

3.5 Concluding Remarks

Running process on the background is one of the essential capabilities that Android OS provides for App developers to avoid interrupting users. Some real-world Android applications conduct activities on the background without users' awareness. However, not all of these Apps are reputed as benign samples. With the assumption of malware samples concealing their operations through the Android background processes, we offer the semi-automated analysis framework in this chapter. Also, in the chapter, if an APK includes suspected codes, that APK is investigated deeper by setting breakpoints and debugging methods.

It is essential to focus further on the *SuspectedSamples* Apps and attempt to conduct behavioral monitoring separately from the *PureSamples* Apps. The proposed analysis framework increases the chance of more centralized observation on the background suspected Apps. Later on, Chapter 4 presents the implementation details regarding the proposed analysis framework.

Figure 3.4: Procedure of Suspected Capturing module 3.3.7 from Component 3.3.



Chapter 4

IMPLEMENTATION

This chapter provides the implementation details of the framework, presented in Chapter 3. The implementation details consist of the developed automated static reversing modules 3.2, semi-automated dynamic debugging modules 3.3, and learning classifiers 3.4. Here, the chapter states the configuration details regarding Android device profiling, Android user profiling, and tools used in this research. To clarify the framework complexity, Chapter 4 presents several depictions, including the FC graph, UML Component View diagram, and UML sequence diagram.

4.1 UML Diagram Types

The Unified Modeling Language (UML) is a standard language to visualize the architecture design of a system [42]. UML is categorized into two diagram types explained below: Structural UML Diagram and Behavioral UML Diagram.

4.1.1 Structural UML - Component Diagram

Several components and sub-components (modules) are designated to form an intelligent Framework. These elements collaborate for successfully conducting actions of the framework's purposes [43]. A Component Diagram illustrates these relations in

detail. Figure 4.1 presents the Component diagram of the framework in this study. It shows all transferring directions of data files, such as APKs, transformed APKs, commands, and captured logs. The transferring paths are either between the data repository and modules, or between the modules.

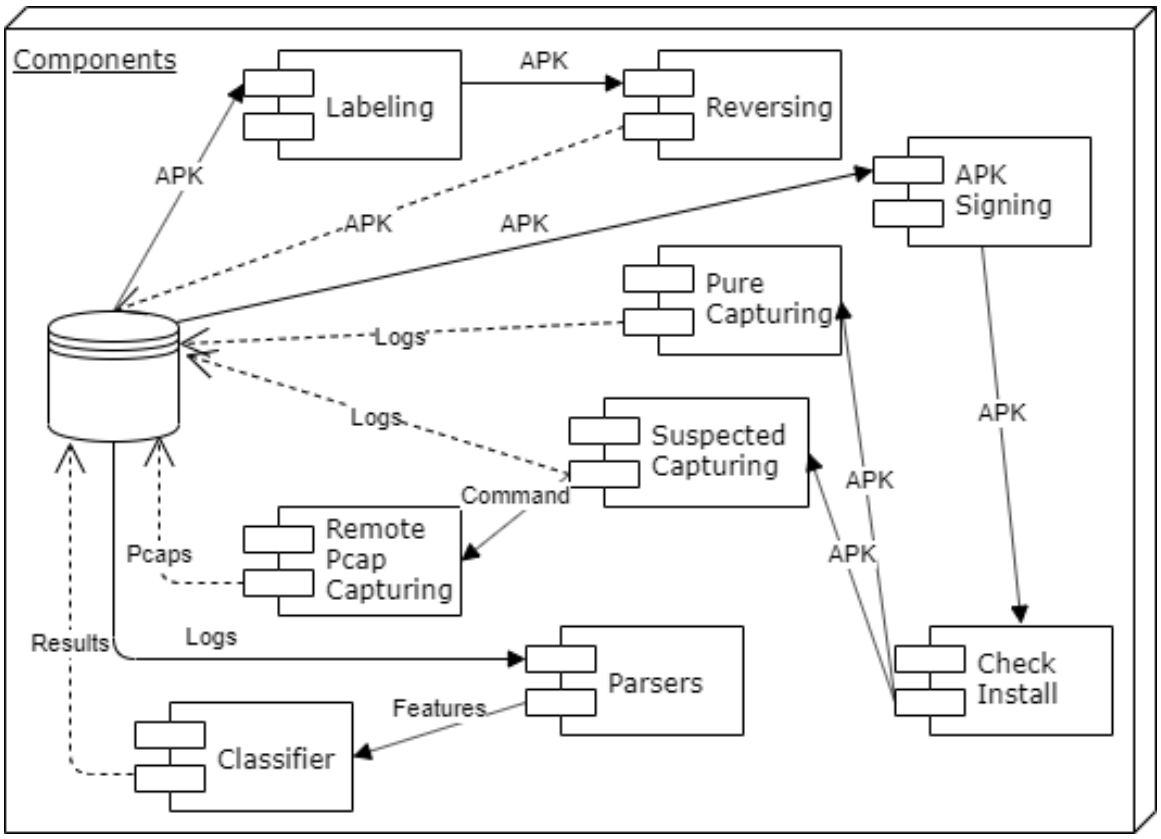


Figure 4.1: Component view of the proposed framework for reversing and debugging Android Apps in Android malware analysis

4.1.2 Behavioral UML - Sequence Diagram

A sequence diagram is a behavioral model that illustrates the interactions among the elements of a system in a sequence of time [44]. This diagram assists in clarifying the complexity of what action sequences are performing on which element and how long they are executing, especially when we have multiple elements in a design. In the Dynamic Debugging Analyzer component 3.3, we developed two modules named

Pure Capturing 3.3.6 and Suspected Capturing 3.3.7, which consisted of multiple elements such as Local PC, Remote PC, and the Mobile Device.

Figure 4.2 provides a sequence diagram regarding Pure Capturing module 3.3.6 that shows how different elements exchange requests and responses in time concepts. Figure 4.2 shows Local PC handles Pcap Capturing SSH requests in parallel through a new thread. In our original design, the Python Script element is executing inside the Local PC element, but we separate them here to reduce complexity in the diagram. The phone is connected to Local PC by USB and connected to Remote PC through HotSpot point.

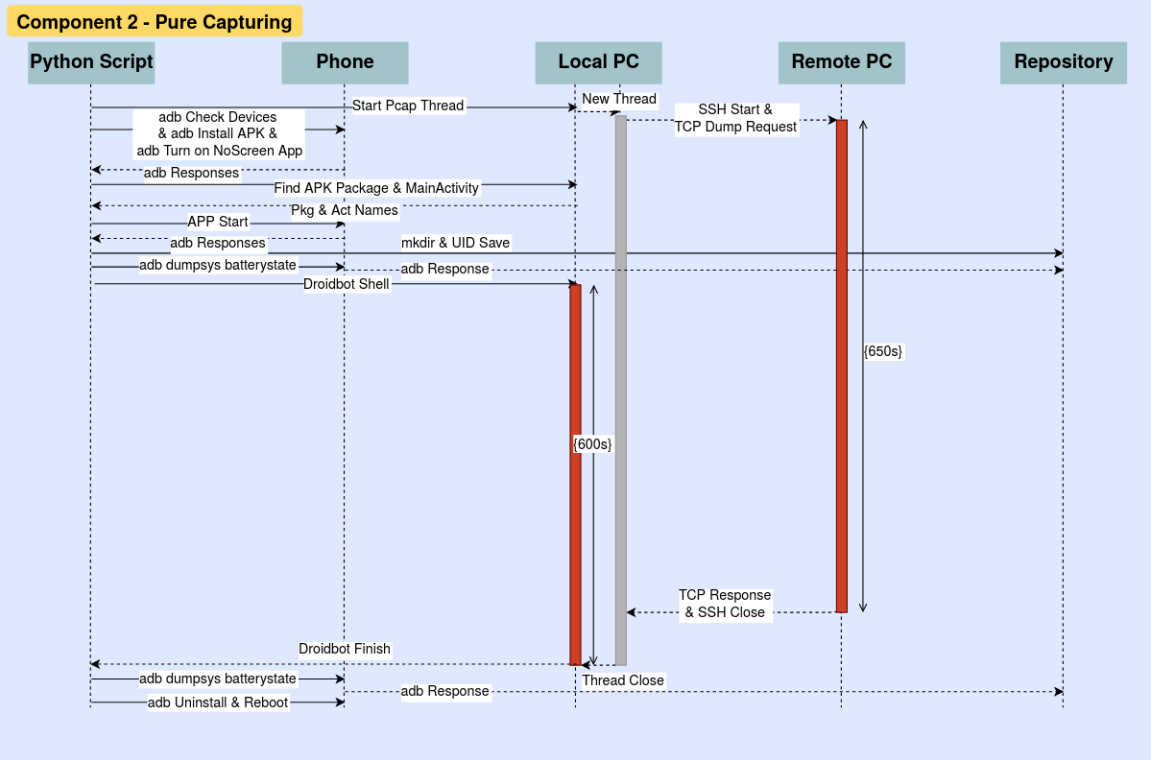


Figure 4.2: Sequence diagram of Pure Capturing module 3.3.6

Figure 4.3 illustrates a UML sequence diagram related to Suspected Capturing module 3.3.7. It represents two additional elements; IDA Pro and IDA Python. In the original design, the IDA Pro runs in Local PC, and IDA Python executes inside IDA Pro. Figure 4.3 depicts that besides the Pcap thread, Droidbot is also running

on a separate thread since the IDA debugger must monitor the attached process in parallel.

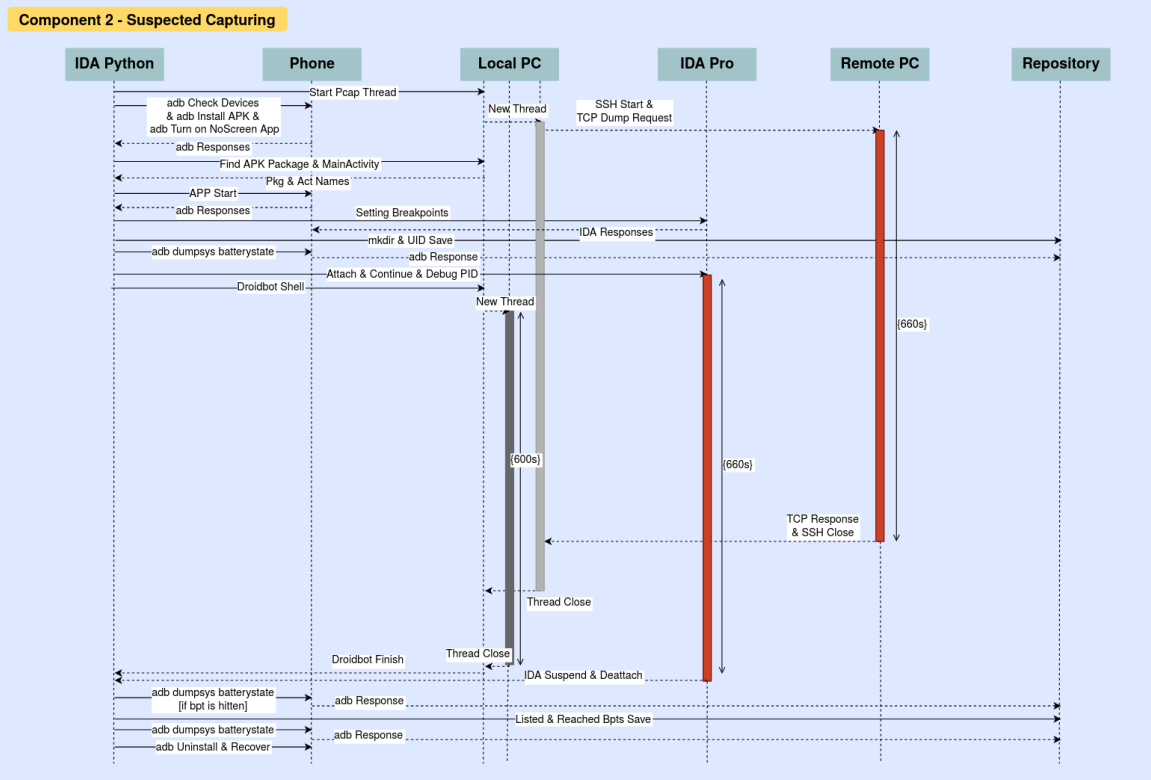


Figure 4.3: Sequence diagram of Suspected Capturing module 3.3.7

In the actual code running, there are two other time factors, including time delays caused by phone-hardware and time delays caused by software that has been omitted from these diagrams to simplify the model.

4.2 Function Call Graph (FCG)

FCG is a graph to draw function calling series in a developed code such as Python code. FCG represents the sequences of calling functions when function B is called inside function C ($C \rightarrow B$). FCG also shows the number of times a function is called. Figure 4.4 is an FCG that illustrates the function calls regarding Python code developed in the Dynamic Debugging Analyzer component 3.3 - Suspected

Capturing module 3.3.7. In Figure 4.4, the Main function revokes other functions, and the order of these callings is from left to right.

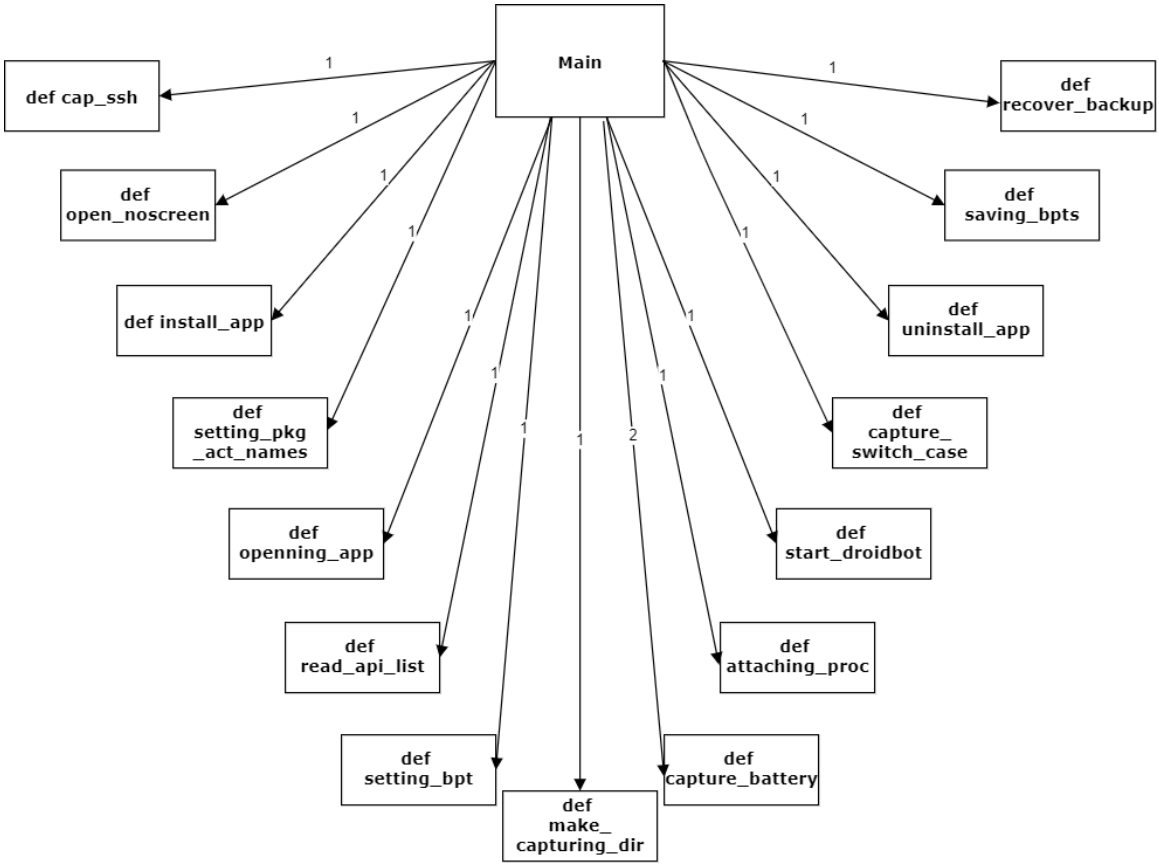


Figure 4.4: Function call graph related to the Suspected Capturing module 3.3.7.

4.3 Configurations and Setups

The configuration setups for the tools and the phone device are documented as below:

4.3.1 Android Device

Each Android user has a unique behavior while using a phone. The study uses the Nexus 5 Android mobile device. Here, we simulate real-world user's behavior by providing two profiling methods:

- *Costumer Profiling*: A real user logs in to social accounts and saves pictures and data on the phone, which represents the customer's behavioral profile. Table 4.1 lists considered customer profiling details in this study.

Table 4.1: Customer profiling setups for the Nexus 5 phone

Customer Profiles
Gmail Account (Thesis.Proj.cic@gmail.com)
Facebook Account (User: Lili ThesisProj)
Personal Photos
Random Contacts

- *Device Profiling*: The Nexus device reverts to specified setups after each rebooting. Table 4.2 mentions the considered setups for this device.

Table 4.2: Device profiling setups for the Nexus 5 phone

Device Profiling
Have Root Access
Installed Apps: 1- NoScreen 2- RootChecker 3- Droidbot
Turned on: Developer Options, USB Debugging, MEO, MTP File Transfer
Turned on: Location
Turned on: Bluetooth
Turned on: WiFi, Connected to CIC HotSpot
Turned off: Google Play Auto Update
Turned off: Lock Screen
Recover to the Last Android Updated Version

4.3.2 Droidbot

Module 3.3.1 of Chapter 3 utilizes Droidbot tool [1] [31] to generate inputs for interacting with the App's UI. Droidbot requires Python 3.6 environment variables with an updated Anroguard [2] version. Table 4.3 provides some configuration factors that have been considered while working with Droidbot.

Table 4.3: Configuration setups for using Droidbot

Input	Setup1	Setup2	Timeout
random	keep_env	keep_app	600 sec

Figure 4.5 provides the Shell code for using Droidbot automatically on a parallel thread.

```
export PATH="~/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"

pyenv activate droidbot && droidbot -d $(echo "$4") -a $(echo "$1")/$(echo "$2") -keep_env -keep_app -
random -timeout 600 -o $(echo "$1")/$(echo $(echo "$2") |cut -d '.' -f 1)/CaptureOutput/droidbot-out-
$(echo "$3")/
```

Figure 4.5: Droidbot Shell Code

4.3.3 SSH Connection

For each APK, an automate network flow monitoring on the remote Hotspot is requested, lasting for **660 seconds** with ***Sudo*** privileges through Shell code provided in Figure 4.6.

```
sshpass -p $UserPassword ssh -tt cic@$ip -p $port 'echo $RootPassword | sudo -S -s /bin/bash -c
"timeout 660 tcpdump -i wlp1s0 -n -s 64 -w ~/Desktop/Remote_pcap_Captured/$(echo '$1').pcap; bash -i"'
```

Figure 4.6: SSH TCP dump request Shell code for capturing Pcaps

[1] https://github.com/honeynet/droidbot
[2] https://github.com/androguard/androguard

4.3.4 APK Resigning

To attach a debugger to an executed APK, a debuggable flag in the APK's Manifest file is required. For this purpose, a Bash code is developed with the following APK resigning procedure:

1) Decompiling APK with Apktool [3]

2) Seeking for Debuggable flag in Manifest file

3) Adding Debuggable flag to Manifest file

4) Rebuilding APK with Apktool

5) Aligning APK with Zipalign [4] to optimize built APK

6) Signing new APK with our certified password from Table 4.4 by APKSigner [5]

7) Verifying new APK with APKSigner

Table 4.4: Certified passwords for resigning APKs

Password 1:	"l1234t"
Password 2:	"l5678t"

Figure 4.7 presents the developed Shell code to automate the mentioned procedure of APK Resigning module 3.3.3.

4.4 Static Reversing Analyzer (SRA)

The Static Reversing Analyzer component 3.2 conducts a static reversing analysis on APKs to search for the specified signatures from Table 3.1 in their source codes. In this component, the APK samples are labeled.

[3]https://ibotpeaches.github.io/Apktool/
[4]https://developer.android.com/studio/command-line/zipalign
[5]https://developer.android.com/studio/command-line/apksigner

```
direc=$(echo "$1")

# mkdir $direc/apktool/

for aapk in $(ls $direc/ | grep ".apk");
do

        # decompile
        echo "************************************************** Decompiling ..."+$aapk
        apktool d $direc/$aapk -o $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')

        # remove debuggable False or True if it has already
        echo "************************************************** Editing Manifest ..."+$aapk
        sed -i 's/android:debuggable="true"//g' $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/AndroidManifest.xml
        sed -i 's/android:debuggable="false"//g' $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/AndroidManifest.xml


        # add debuggable flag to manifest
        echo "************************************************** Editing ..."+$aapk
        sed -i 's/<application/& android:debuggable="true"/' $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/AndroidManifest.xml

        # build apk
        echo "************************************************** Building ..."+$aapk
        apktool b $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/

        # align apk
        echo "************************************************** Aligning ..."+$aapk
        ~/Android/Sdk/build-tools/29.0.2/zipalign -v -p 4 $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/dist/$aapk   $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../
\1/')/dist/unsigned_aligned_$aapk

        # sign apk + enter passwords
        echo "************************************************** Signing ..."+$aapk
        pass1="l1234t"
        pass2="l5678t"
        echo -e $pass1'\n'$pass2 | ~/Android/Sdk/build-tools/29.0.2/apksigner sign --ks laya-release-key.jks --out $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/dist/
app_release_$aapk   $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/dist/unsigned_aligned_$aapk


        # test signed apk ?? check output
        echo "************************************************** verifying ..."+$aapk
        ~/Android/Sdk/build-tools/29.0.2/apksigner verify $direc/apktool/$(echo $aapk | sed 's/\(.*\)..../\1/')/dist/app_release_$aapk

done
```

Figure 4.7: Automated APK Resigning Shell code

4.4.1 Tools

The first component 3.2 uses Bash coding and the following tools:

- *APKtool* [6] assists in unpacking an APK's constituents. APKtool uses *backSmali* [7] for disassembling the binary dex file of an APK to its corresponding Smali code.

- *VirusTotal-API-Tool* [8] is a tool for downloading the corresponding Anti-Viruses' JSON reports of VirusTotal [9] regarding Apps.

- *AVClass* [10] [35] votes among the outcome reports from different anti-viruses to finalize Apps' label-tags.

- *dex2jar* [11] is a tool to convert an Android Dalvik Executable (.dex/.odex) format to its corresponding .jar file which is a compressed package file consisting of all Java .class and metadata resources.

[6] https://ibotpeaches.github.io/Apktool/
[7] https://github.com/JesusFreke/smali
[8] https://github.com/Xen0ph0n/VirusTotal_API_Tool
[9] https://www.virustotal.com/gui/home/upload
[10] https://github.com/malicialab/avclass
[11] https://sourceforge.net/projects/dex2jar/files/

- *Java Class Disassembler (Javap)* [12] is a command utility to disassemble compiled byte code Java class files (.class) to human-readable format.

4.4.2 Coding Details

Figure 4.8 presents a Shell code for the Static Reversing Analyzer component 3.2 that automatically reverts all APKs and divides them into two *PureSamples* and *SuspectedSamples* groups based on the observed signatures.

4.5 Dynamic Debugging Analyzer (DDA)

The second component 3.3 includes the Pure Capturing module 3.3.6 and the Suspected Capturing module 3.3.7 for APK debugging. In the following, additional details regarding the implementations of these modules are provided.

4.5.1 Tools

Component 3.3 is developed with the Python and Bash Coding. Besides Droidbot 4.3.2, SSH Connection 4.3.3, and APK Resigning 4.3.4 tools, Component 3.3 utilizes the following tools:

- *Android Debug Bridge (ADB)* [13] is a command-line tool for interacting with a plugged-in mobile device. ADB assists in working with a smartphone to install or debug Apps. It has three components: a client that is running on the machine to send a command to the phone, a daemon (ADBD) that is run on the phone as a background process to obey the received commands from the machine, and a server that manages multiple connections between the different clients and several daemons on different phones.

[12] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html
[13] https://developer.android.com/studio/command-line/adb

```
# for each apk
for aapk in $(ls $(echo "$1")/); #??? test??
do
        #name of the apk and its folder
        tempDir=$(echo $aapk |cut -d '.' -f 1)

        # run dex2jar
        cp $(echo "$1")/$aapk dex2jar-2.0/
        sh dex2jar-2.0/d2j-dex2jar.sh -f dex2jar-2.0/$aapk -o dex2jar-2.0/$tempDir.jar

        # extract jar file
        cp -r dex2jar-2.0/$tempDir.jar $(echo "$1")/Jaroutputs/
        rm -r dex2jar-2.0/$tempDir*

        # Unzip classes.jar for each APK
        mkdir $(echo "$1")/Jaroutputs/$tempDir
        unzip $(echo "$1")/Jaroutputs/$tempDir.jar -d $(echo "$1")/Jaroutputs/$tempDir/

        found_api_name=''
        # loop over java classes and folders
        class_list=$(find $(echo "$1")/Jaroutputs/$tempDir/ -name "*.class")
        flg_fnd_api=0
        for fold in $class_list;
        do
                # get the class name from its fold address
                fil=$(echo $fold | rev | cut -d '/' -f 1 | rev) # name of the .class file
                #path_out=$(echo $fold | rev | cut -d '.' -f 2 | rev) #path name

                # make a folder for each class
                tmp_name=$(echo $fold | tr '.' '-' )
                mkdir $tmp_name

                # Convert java.class to smali
                $(javap -c $fold > $tmp_name/$(echo $fil | cut -d '.' -f 1).txt)
                echo '%%%'
                echo $fold

                # search into smali files for APIs and save their address and kind
                # in array from list
                for api in ${API_List[@]};
                do
                        tmp_found_api=$(cat $tmp_name/*.txt | grep -c $api)

                        if [ $tmp_found_api -ne 0 ]
                        then
                                found_api_name+='{'
                                found_api_name+=$fold
                                found_api_name+=' % '
                                found_api_name+=$api
                                found_api_name+=' % '
                                found_api_name+=$tmp_found_api
                                found_api_name+='}'
                                found_api_name+=' % '

                                flg_fnd_api=1
                                echo $api
                                echo '%%%'

                        else
                                echo "output is null"
                                echo '%%%'

                        fi

                done
        done
        if [ $flg_fnd_api -ne 0 ] ;
        then
                cp $(echo "$1")/$aapk $(echo "$1")/SuspectedSamples/
        else
                cp $(echo "$1")/$aapk $(echo "$1")/PureSamples/
        fi

        #save founded API array in a file
        echo $found_api_name > $(echo "$1")/Jaroutputs/$tempDir/FoundedAPIName.txt
done
```

Figure 4.8: Shell code of the Static Reversing Analyzer component 3.2

- *Interactive Disassembler (IDA)* [14], is software for disassembling executable machine codes into Assembly languages. IDA Pro supports a variety of platforms, including Windows, Linux, and Dalvik. Through IDA Pro, analyzers can elaborate on a program while executing it. It cooperates with different debuggers, such as GDB [15]. Appendix A introduces GDB and its installation procedure.

  IDA API provides IDC capabilities, including asynchronous debugging extensions. It hooks to HT_DBG for receiving debugger notifications. Commands to work with the debugger in IDA API come in pair functions [14]; one is the synchronous function (as the command() function) in the script, and the counterpart is asynchronous (as request_command function). The request commands are queued for later processing, and the HT_DBG receives the corresponding debugger notification events. Because IDA API demands users to debug through the event-base model program, users cannot develop a simple sequential program and must handle the event notifications with an event-handler design to suspend or terminate the attached process. Hence, the IDC interface provides *GetDebuggerEvent()* function that enables the debugger to wait for an event with a timeout value.

- *IDAServer* [16], is a program that works with IDA Pro and allows users to debug another program remotely, installed on a connected phone.

- *IDAPython* [17], [18], is a plugin that adds Python programming language to IDA Pro. With the assistance of IDAPython, users can write Python scripts to automate the IDA Pro capabilities. It provides all debugging functionalities, including BPT manipulations (setting BPT and deleting BPT), and debugger

[14]https://www.hex-rays.com/products/ida/index.shtml
[15]https://www.gnu.org/software/gdb/
[16]https://www.hex-rays.com/products/ida/debugger/
[17]https://www.hex-rays.com/products/ida/support/idapython_docs/
[18]https://web.archive.org/web/20060116123556/http://d-dome.net/idapython/

event-handling (starting a process, attaching to a process, continuing a process, and changing the program's IP register). Users can develop static python scripts that retrieve some information regarding the executable source code, including the number of function calls, function call graphs, and searching assembly instructions (operation or operands). Also, users can develop dynamic python scripts that semi-automate the procedure of debugging and monitoring registers and breakpoints.

- *Android SDK* [19], is a set of tools for developing Android Apps.

4.5.2 Coding Details

The Python script for the Pure Capturing Module 3.3.6 of the 2nd Component 3.3 is provided in Figure 4.9. Figure 4.10 states the IDAPython script code of the Main function for the Suspected Capturing module 3.3.7.

4.6 Learning Classifier (LC)

Python and Bash codes are used for developing the 3rd component 3.4 and its parsers. Component 3.4 extracted battery, Intent, permission, and metric features for training the learning models. Also, the machine learning algorithms are expanded in Python programming language with the assistance of the *scikit-learn* [20] ML Python library.

4.6.1 Tools

The following tool is used in Component 3.4:

- *CICFlowMeter tool* [21] [28], [13], is a network traffic analysis tool that is used to obtain 84 dynamic network traffic features from Pcap files.

[19] https://developer.android.com/studio
[20] https://scikit-learn.org/stable/user_guide.html
[21] https://github.com/ahlashkari/CICFlowMeter

```
# start pcap capturing
print("start pcap capturing")
t = threading.Thread(target=cap_ssh,name="starting_ssh_pcap",args=())
t.daemon = True
t.start()

# Install APK
print("Installing: ...")
cmd_install= "adb -s {0} install {1}/{2}".format(serial,direct,apk)
result= os.popen(cmd_install).read()
# wait to install
time.sleep(5)
print(result)

# open no screen off app
print("Openning no screen off App: ...")
cmd_noscrn= "adb -s {0} shell am start -n com.k2.no.screen.off/com.k2.no.screen.off.NoScreenOffActivity".format(serial)
os.system(cmd_noscrn)

# <manifest android:installLocation="auto" package="com.bysczh.converter"> -> pkg
cmd_pkg = "aapt dump badging {0}/{1} | grep 'package: name='".format(direct,apk)
pkg_name= os.popen(cmd_pkg).read()
pkg_name= pkg_name.split('package: name=', 1)[1].split(' ', 1)[0].replace("'", '')
print("package name is: " + pkg_name)

# <activity android:label="@string/app_name" android:name=".Converter" android:theme="@style/Theme.NoTitleBar.WithColoredSpinners" -> activity
cmd_act= "aapt dump badging {0}/{1} | grep 'launchable-activity: name='".format(direct,apk)
act_name= os.popen(cmd_act).read()
if act_name:
        act_name= act_name.split('launchable-activity: name=', 1)[1].split(' ', 1)[0].replace("'", '')
        print("Activity name is: " + act_name)

        # open target app
        print("Openning App: ...")
        cmd_strt= "adb -s {0} shell am start -n {1}/{2}".format(serial, pkg_name, act_name)
        os.system(cmd_strt)
        time.sleep(5)

        #make capturing directory
        cmd_hash= "sha1sum {0}/{1} | cut -d ' ' -f 1".format(direct,apk)
        apk_hash= os.popen(cmd_hash).read()
        apk_hash = apk_hash.split('\n', 1)[0]

        folder = apk.split('.apk', 1)[0]
        cmd_folder = "mkdir -p {0}/{1}".format(direct,folder)
        os.system(cmd_folder)

        cmd_mkdir= "mkdir -p {0}/{1}/CaptureOutput/{2}-{3}-{4}".format(direct,folder,pkg_name,act_name,apk_hash)
        os.system(cmd_mkdir)

        # # save UID
        print("UID saving: ...")
        cmd_uid= " adb -s %s shell ps | grep %s"%(serial, pkg_name)
        app_uid= str(os.popen(cmd_uid).read())
        print(app_uid)
        cmd_save = "echo '{0}' > {1}/{2}/CaptureOutput/{3}-{4}-{5}/AppPID.txt".format(app_uid,direct,folder,pkg_name,act_name,apk_hash)
        os.system(cmd_save)

        # capture battery state installed
        print("Capturing Battery: Installed")
        cmd_dump= "adb -s {0} shell dumpsys batterystats> {1}/{2}/CaptureOutput/{3}-{4}-{5}/Battery-Installed.log".format(serial,direct,folder,pkg_name,act_name,apk_hash)
        os.system(cmd_dump)

        # start droidbot
        # droidbit with random timeout -> multi thread
        print("starting Droidbot")
        cmd_droidbot = "/home/itaheri/Taheri2/itaheri/Documents/Thesis/Coding/module2/droidbot-master/./DroidBot {0} {1} 1 {2}".format(direct,apk,serial)
        os.system(cmd_droidbot)
        print("Stopping Droidbot")
        time.sleep(5)

        # capture battery state final
        print("Capturing Battery: final")
        cmd_dump= "adb -s {0} shell dumpsys batterystats> {1}/{2}/CaptureOutput/{3}-{4}-{5}/Battery-Final.log".format(serial,direct,folder,pkg_name,act_name,apk_hash)
        os.system(cmd_dump)
        time.sleep(5)


else:
        print("activity not found")

# ##########################

# # uninstall app
print("Uninstalling App ...")
cmd_uninstall= "adb -s {0} uninstall {1}".format(serial,pkg_name)
un_res= os.popen(cmd_uninstall).read()
print(un_res)
time.sleep(5)

# restart phone
cmd_restart= "adb -s {0} reboot".format(serial)
os.system(cmd_restart)
time.sleep(60)

# ####################################
cmd_finish = "echo 'APK {0} is finished'".format(apk)
os.system(cmd_finish)
print("%%%
```

ates the framework with learning models and presents the evaluation results.

```
##################
# Main:
##################
# start pcap capturing
print("start pcap capturing")
t1 = threading.Thread(target=cap_ssh,name="starting_ssh_pcap",args=())
t1.daemon = True
t1.start()

# open no screen off app
open_noscreen()

# Install APK
install_app()

# setting pkg and act names
pkg_name, act_name = setting_pkg_act_names()

if not act_name:
        print("activity not found**********************************")

# open target app
openning_app()
#####################################
# reading API List
API_Assenble = read_api_list()

#####################################
# setting bpts
bpt_list, bpt_num = setting_bpt()

#####################################
# make directory and save UID
folder, apk_hash = make_capturing_dir()

# first capturing after installing the apk
capture_battery("Installed")

#####################################
# Attaching to a running app
print("Attacheing to process")
app_pid= attaching_proc()
print(app_pid)
res_attach= AttachProcess(app_pid, -1)

if res_attach <= 0:
        print(res_attach)
else: # it is attached now
        capturing_attempts = 1
        t = threading.Thread(target=start_droidbot,name="starting_droidbot",args=(direct, apk, folder, capturing_attempts,serial))
        t.daemon = True
        t.start()

        # Continue process while debugging
        # if no bpt is hitten by running, we will suspend process and there
        #       will be no capturing
        print("Start monitoring ...")
        x= idaapi.continue_process()
        code = idaapi.wait_for_next_event(WFNE_SUSP, 660) # -1

        time.sleep(10)

        #  while ip update or code == 16
        if code != 16: #
                print("suspended by timeout!")
                idaapi.suspend_process()
                print("code: " + str(code))
                # check final PID, if reached bpt the pid gonna change anyway
                pid_f= attaching_proc()
                print("Final PID: ")
                print(pid_f)
        else:
                #  here means code = 16
                # same PID
                print(x)
                print("code: " + str(code))
                bpt_reached = capture_switch_case(code)

#####################################
# final capturing
capture_battery('Final')

#####################################
# uninstall app
uninstall_app()
#####################################
saving_bpts()

#deattach
print("Deattaching")
rr = detach_process()
print(rr)
# #####################################
print "APK %s finished!" % (apk)
print("%%%
```

Chapter 5

Experiments and Results

In this chapter, the Learning Classifier component 3.4 evaluates the framework regarding eliciting malware samples that are conducting malicious activities through specified background services. We observe the background services through debugging techniques while executing the App on a real smartphone and capturing battery logs. The background signatures of malware samples are monitored to reduce the FP rate and promote learning models to accomplish malware classification based on the generated data. The higher a classifier's accuracy, the more successful the framework is in activating and extracting hidden background processes of the malicious samples. Here, the chapter presents details regarding the experimental environment, including the repository, evaluation, visualization graphs, and comparison result descriptions.

5.1 Data Repository

Using a well-known dataset enriches results in research. However, the currently available Android datasets, reviewed in Section 2.6, are not able to fulfill the needs of data in this study. As a result, new data are generated, included battery logs, network traffic, and static features, while debugging installed Apps. The executed samples are gathered from two well-known repositories, named CICAndMal2017 [30]-

CICInvesAndMal2019 [39] and Android Wake Lock Research [1].

The CICAndMal2017 - CICInvesAndMal2019 consists of 426 malware and 1700 benign in five malware categories, including Adware, Ransomware, Scareware, SMS-Malwre, PremiumSMS, along with 42 malware families. The Android Wake Lock Research provides 44,736 unlabeled open-source Android Apps that are using Wake-Locks in their codes.

5.1.1 Data Distribution

In the APK repository, we have randomly selected 319 malware samples in three categories, including Adware, Scareware, and SMS (SMSMalwre and PremiumSMS) from CICAndMal2017, and 408 benign samples from Android Wake Lock Research. The benign samples are identified with VT labels, as explained in Module 3.2.1.

The collected APKs of the repository pass to procedures through the framework's components. As the output of the first component 3.2, the APK repository is divided into two groups, *PureSamples* and *SuspectedSamples*. Tables 5.1 and 5.2 show the number of benign and malware samples that have been allocated to each group, respectively.

Table 5.1: Number of benign samples that have been allocated to each *PureSamples* and *SuspectedSamples* group in Component 3.2

	Benign	
Number Types	Count/All Benign	Percentage
PureSamples	14/408	3.43%
SuspectedSamples	394/408	96.57%

Table 5.2: Number of malware samples that have been allocated to each *PureSamples* and *SuspectedSamples* group in Component 3.2

Malware Category	**Adware**		**Scareware**		**SMSware**	
Number Types	Count/All Malware	Percentage	Count/All Malware	Percentage	Count/All Malware	Percentage
PureSamples	14/319	4.39%	8/319	2.51%	18/319	5.64%
SuspectedSamples	86/319	26.96%	102/319	31.97%	91/319	28.53%

[1]http://sccpu2.cse.ust.hk/elite/dataset.html

The second component 3.3 executes Apps of the *PureSamples* group on the phone while capturing logs. Also, Component 3.3 debugs Apps of the *SuspectedSamples* group after passing the following four stages. First, APKs of the *SuspectedSamples* that are not compatible with the used real smartphone (Nexus 5) are separated (allocated to *NotCompatible* folder). Second, the APKs of *SuspectedSamples* that crash during debugging procedures are placed in the *DebugCrashed* folder. The remained APKs of *SuspectedSamples* finish debugging procedure successfully. Third, if these APKs end with a nonidentical PID compared to their initial PID, they are allocated to the *DebugWrongPID* folder. Otherwise forth, they belong to the *DebugCorrectPID* folder. Tables 5.3 and 5.4 summarizes the distributions of *SuspectedSamples* APKs in the analysis stages for benign and malware samples, respectively.

Table 5.3: Distributions of the *SuspectedSamples* benign APKs at the four stages in Component 3.3

	Analysis Stages			
Benign	NotCompatible	DebugCrashed	DebugWrongPID	DebugCorrectPID
SuspectedSamples	246/394, 62.44%	57/394, 14.47%	59/394, 14.97%	32/394, 8.12%
	Count/All SuspectedSamples, Percentage			

Table 5.4: Distributions of the *SuspectedSamples* malware APKs at the four stages in Component 3.3

	Analysis Stages			
Malware	NotCompatible	DebugCrashed	DebugWrongPID	DebugCorrectPID
Adware	14/86, 16.28%	19/86, 22.09%	40/86, 46.51%	13/86, 15.12%
Scareware	17/102, 16.67%	18/102, 17.64%	51/102, 50.00%	16/102, 15.69%
SMSware	9/91, 9.89%	14/91, 15.39%	44/91, 48.35%	24/91, 26.37%
	Count/All SuspectedSamples, Percentage			

To evaluate the proposed framework in the third component 3.4, we have utilized *PureSample* of Table 5.1 (total 14) and Table 5.2 (total 40), and benign *SuspectedSamples* of *DebugWrongPID*, *DebugCorrectPID* - Table 5.3 (total 91) and malware *SuspectedSamples* of *DebugCorrectPID* - Table 5.4 (total 53). Figure 5.1 illustrates the distribution of final analyzed APKs used in the evaluation procedure of Component 3.4.

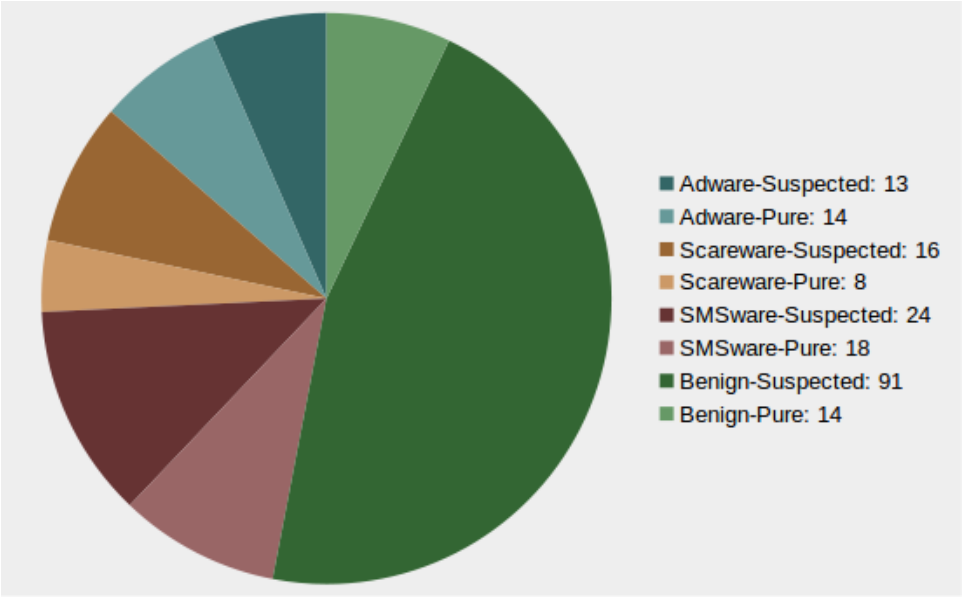


Figure 5.1: distribution of final analyzed APKs used in the evaluation procedure, component 3.4

5.2 Evaluations

In this section, several learning classifiers are conducted to examine the proposed framework with the assistance of the final selected samples in Figure 5.1.

5.2.1 Experimental Results

Two learning classifiers models are used in the experiment procedure, including Random Forest (RF) and Multi-Layer Perceptron (MLP) neural network. These models are trained several times with different variable parameters, including the number of trees in RF and the number of perceptrons in each layer of MLP. Later on, each trained model is tested twice, one time with 20% of the dataset (remaining 80% for the training phase), another time with the 5-fold cross-validation method. As Figure 5.1 delineates in more detail, we have used 105 benign and 93 malware samples in total to examine the framework. Table 5.5 states the number of each feature set extracted through the developed parsers explained in Module 3.4.1.

The MLP model has three layers, with variable parameters as the number of nodes

Table 5.5: Number of each feature set extracted through the developed parsers explained in Section 3.4.1

Features	**Permission**	**Intent**	**Metric**	**Pcap**	**Battery**
Numbers	304	351	8	76	62

for each layer. We have trained the models with different node numbers for all three-layers, which is in the range of 10, 20, 30, 40, and 50. Also, the features are normalized with *sklearn.preprocessing.StandardScaler* method before the training phase in MLP models.

The RF model is trained several times, with different variable parameters as the number of trees in the forest, which is in the range of 100, 200, 300, 400, and 500. Figure 5.2 depicts the comparison among the accuracy results of two ways of learning (80%Training-20%Testing and Cross-Validation) for MLP and RF models with variable parameters. Table 5.6 provides the best results of the NN and RF classifications using specific variable parameters. Figure 5.3 presents the corresponding confusion matrices regarding the best results of the trained classifications in Table 5.6.

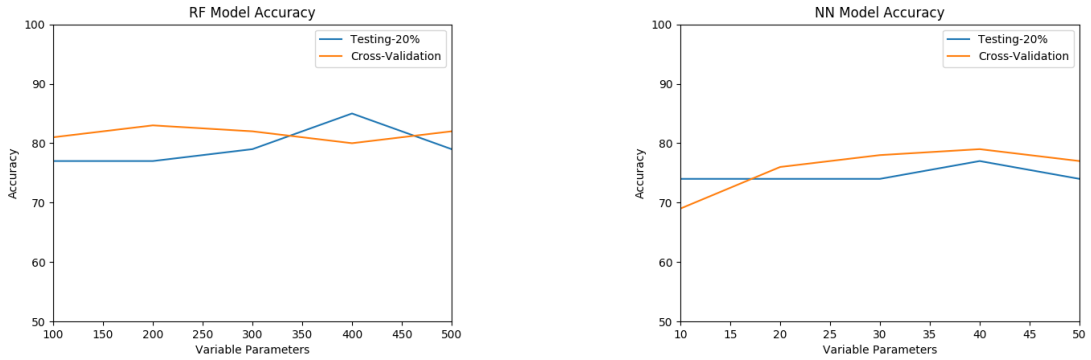


Figure 5.2: Comparison among the accuracy results of testing methods (Testing-20% and Cross-Validation) for MLP and RF models

Table 5.6: The best results of the NN and RF classifications using specific variable parameters

Classifier	**RF**	**MLP-3Layers**
Variable Parameters	n_estimators=400 random_state=400	nodes=(40, 40, 40)
Learning Design	80%Training-20%Testing	80%Training-20%Testing
Accuracy	0.85	0.77
Precision	0.88	0.78
Recall	0.85	0.77
F1-Score	0.83	0.76

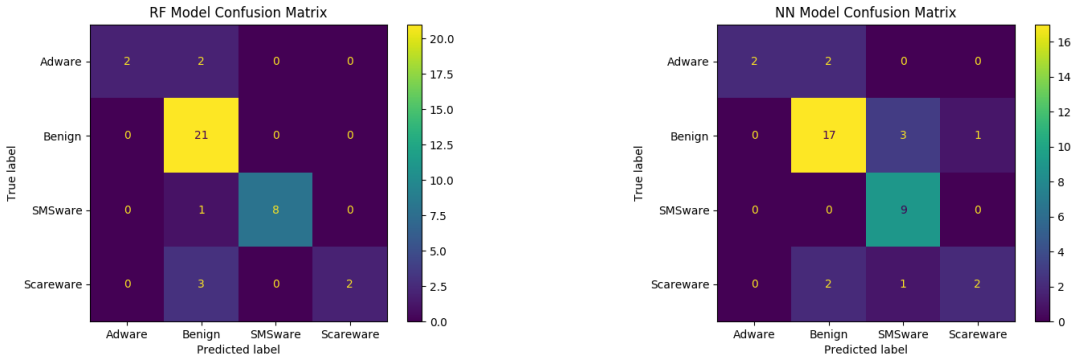


Figure 5.3: Corresponding confusion matrices regarding the best results of the trained classifications in Table 5.6

5.3 Results Comparison

As Figure 5.2 presents, the average accuracy of RF classifications is higher than the NN ones. Figure 5.2 shows that the peak results for RF models are achieved, with 400 trees as a variable parameter, while using the 80%-20% learning method. Also, the peak results for NN-3layers models are obtained, with 40 nodes for each layer as variable parameters, while using both learning methods (80%-20% and cross-validation).

Figure 5.3 depicts that benign samples are successfully separated with the RF model, and each malware category is showing distinct characteristics compared to other malware categories. However, a few malware samples are mixed up with benign

samples during RF classification. In Figure 5.3, the benign samples are adequately divided with the NN model, but a few malware samples are mislabeled.

To realize which classification method is working more efficiently in the particular problem, the receiver operating characteristic (ROC) curve can assist. The ROC curve is beneficial to compare the overall performance of classifiers in binary classification problems. With ROC, the analyzers can specify a reasonable threshold according to their needs to balance the FP rate and TP rate with great accuracy. The FP rate states that how often a classifier wrongly detects a *Negative* sample as *Positive* (False detection). On the other hand, the TP rate indicates how often a classifier correctly detects *Positive* labels as *Positive* (True detection). Accuracy (ACC) shows the percentage of all the correct predictions. In a malware classification problem, *Negative* can be allocated to *Malware* labels and *Positive* can be allocated to *Benign*. The ideal result includes lower FP and higher TP rates while maintaining high ACC.

Figure 5.4 presents the ROC curves for the trained binary RF and NN classifications based on the whole dataset of this research. As Figure 5.4 implies, the ROC curve of the RF model is working more effectively than the NN model since the RF curve is closer to the upper edge of the graph than the NN curve. In the android malware classification problem, the threshold should satisfy the highest TP rate and the lowest FP rate with a higher ACC. Since mislabeling malware is more dangerous, losing the TP rate (which means the classifier would label benign as malware wrongly) is more reasonable than increasing the FP rate (which means that the classifier would mislabel malware as benign). Hence, choosing a point closer to the left and upper edges in Figure 5.4 is the best threshold for the Android binary malware classifications.

The area under a ROC curve (AUC) indicates the accomplishment of the matching binary classifier in detecting samples. The more AUC is closer to 1, the better. If

AUC is under 0.5, it means the classifier is mislabeling all samples. As Figure 5.4 depicts, the RF ROC curve has a larger AUC than the NN ROC curve.

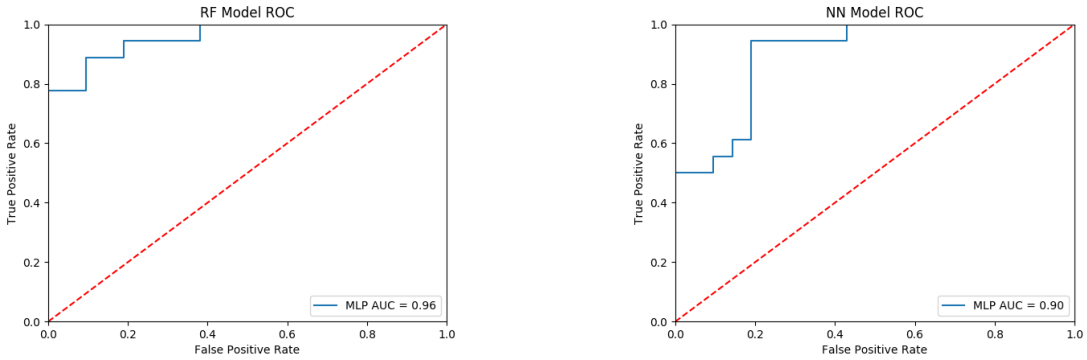


Figure 5.4: ROC curves regarding the trained binary RF and NN classifications

5.4 Concluding Remarks

This chapter provides the evaluation results regarding the proposed framework. In the chapter, details of the APK repository are presented. The chapter examines the framework with explained final selected samples through several learning classifiers and compares the results.

The next chapter concludes the thesis by providing the overall concepts of the proposed framework and the contributions, and it states the recommended future works.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

Android application development offers advanced programming characteristics to improvise capabilities for customers. The default setups for Android OS is determined to put the device in sleep mode after a few seconds of nonuse to prohibit battery draining during the day. However, Android OS considers background services for developers to proceed with continuous tasks even when the device is in sleep mode and avoid interrupting users while the screen is off.

These Android characteristics introduce sophisticated Apps with the ability to run tasks on the background without the user's awareness. These Apps keep running even when they are not the foreground App or when the device is in sleep mode. Advanced Malware Apps can misuse Android's background development options to accomplish malicious missions on the phone away from the user's sight.

The emersion of advanced malware has motivated us to propose a novel hierarchical three-layered malware analysis framework to investigate background malware APKs, with the assistance of dynamic automated reverse-engineering and semi-automated debugging methods. The Static Reversing Analyzer (SRA) layer divides APKs into

two groups, *SuspectedSamples* and *PureSamples*, according to specify background signatures in APKs' source files. The Dynamic Debugging Analyzer (DDA) focuses on monitoring the *SuspectedSamples* APKs with debugging techniques while capturing battery logs at different stages, including after installation, at a possible reached breakpoint, and before the uninstallation. The Dynamic Debugging Analyzer also observes the behavior of the *PureSamples* APKs in a simple capturing procedure. We also monitor Apps' network traffic through a remote HotSpot. Finally, Learning Classifier (LC) evaluates the extracted dynamic background features retrieved from formerly captured logs, and static characteristics retrieved from APK source files. Finally, the framework is evaluated with two classification models learned with two methods, including 80%Training-20%Testing and Cross-Validation. The evaluation results indicate that the classifications have succeeded in gaining 85% with the RF model and 77% with the NN model. These results represent the success rates of the framework in triggering particular hidden tasks with dynamic debugging and analyzing background malware for the first time in Android classification attempts.

6.2 Future Works

The research has some limitations that are overviewed below, alongside some future works as recommendations for proceeding the study:

- To attach a running App to a debugger, it must be a debuggable released APK version. However, the processes of decompiling the APK, adding the debuggable flag, and rebuilding the APK may corrupt the App. Currently, there are no alternative solutions for attaching a debugger to an executed App.

  Future Work: *Advanced Decompiler*, while a decompiler is more capable, it causes less data loss while reverting APK compilation sequences. For future

works, we would seek a more advanced decompiler than APKtool to reduce App corruption percentage in our analysis. Also, we will consider solutions for handling de-obfuscation and de-packing methods before decompiling Apps.

- Some previous studies have stated that advanced malware might detect an attached debugger and not reveal their intentions. In the real world, high-level malware can identify all analysis factors that researchers are currently using, including a connected phone, ADB, and emulators. At this point, there is no alternative malware analysis environment to address the mentioned issues.

  Future Work: *Sophisticated Malware*, we will investigate APKs to determine the percentage of Apps in our repository that are attempting to detect the analysis environment and avoid triggering intentions. We will separate the advanced malware and investigate, using a more complicated methodology.

- Reaching a UI breakpoint while debugging an App is similar to the search algorithm of finding a goal in a tree without knowing the exact branch. If we know the path to the target, it is easy to spot the goal. In this analysis, we utilize a semi-smart UI input testing method that provides no guarantee in finding a particular UI element since the method is not aware of the UI path.

  Future Work: *Smart UI Input Testing*, as future work, we will work on using a *Smart UI Input Testing* tool that is aware of the direction from the initial UI element to the target UI element, and automatically triggers all requirements to reach the target breakpoint. In this case, our analysis will be faster and more efficient.

- This study aggregates APKs from two resources for further analysis through the proposed framework due to the lack of time.

  Future Work: *Heterogeneous Resources*, in the future, we will collect more samples from a variety of APK repositories to investigate in a completed model.

- For the first time, we investigate background signatures. The debugging breakpoints are placed on these signatures in APKs' source files while executing on the phone. We extract a list of 8 more suspicious background WakeLock signatures of *GetSystemServices*. However, this list can always be improved.

  Future Work: *Comprehensive Background Signature List*, we will conduct more research on the list of background WakeLock lists and attempt to survey their purposes. We will create a more comprehensive list of risky background signatures while considering their intentions. The more signatures we collect, the more breakpoints we will set up. Hence, we will have additional chances of capturing logs and analyzing Apps.

Bibliography

[1] Android Analyzers, *Android adb shell recovery mounting*, Last accessed: on Jan 2020, https://android.stackexchange.com/questions/39623/android-adb-shell-recovery-mounting/112598.

[2] ———, *Android applications reversing 101*, Last accessed: on Jan 2020, https://www.evilsocket.net/2017/04/27/Android-Applications-Reversing-101/.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens, *Drebin: Effective and explainable detection of android malware in your pocket*, Ndss, vol. 14, 2014, pp. 23–26.

[4] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky, *Boxify: Full-fledged app sandboxing for stock android*, (2015).

[5] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li, *Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach*, computers & security **73** (2018), 326–344.

[6] EPI CIDRE, *Kharon dataset: Android malware under a microscope*, Learning from Authoritative Security Experiment Results (2016), 1.

[7] Android Developers, *Android architecture*, Last accessed: on Jan 2020, https://source.android.com/devices/architecture.

[8] ———, *Android background tasks*, Last accessed: on Jan 2020, https://developer.android.com/guide/background.

[9] ———, *Android context*, Last accessed: on Jan 2020, https://developer.android.com/reference/android/content/Context.

[10] ———, *Android intents*, Last accessed: on Jan 2020, https://developer.android.com/guide/components/intents-filters.

[11] ———, *Android permissions*, Last accessed: on Jan 2020, https://developer.android.com/guide/topics/permissions/overview.

[12] IDA Developers, *Ida help*, Last accessed: on Jan 2020, https://www.hex-rays.com/products/ida/support/idadoc/157.shtml.

[13] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani, *Characterization of encrypted and vpn traffic using time-related*, Proceedings of the 2nd international conference on information systems security and privacy (ICISSP), 2016, pp. 407–414.

[14] Chris Eagle, *The ida pro book*, no starch press, 2011.

[15] Stack Exchange, *Debugger*, Last accessed: on Jan 2020, https://reverseengineering.stackexchange.com/questions/tagged/debuggers.

[16] ———, *Decompiler*, Last accessed: on Jan 2020, https://reverseengineering.stackexchange.com/questions/tagged/decompiler.

[17] ———, *Disassembler*, Last accessed: on Jan 2020, https://reverseengineering.stackexchange.com/questions/tagged/disassemblers.

[18] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell, *Androdialysis: Analysis of android intent effectiveness in malware detection*, computers & security **65** (2017), 121–134.

[19] Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani, *Droidkin: Lightweight detection of android apps similarity*, International Conference on Security and Privacy in Communication Systems, Springer, 2014, pp. 436–453.

[20] Johannes Hoffmann, Stephan Neumann, and Thorsten Holz, *Mobile malware detection based on energy fingerprints—a dead end?*, International Workshop on Recent Advances in Intrusion Detection, Springer, 2013, pp. 348–368.

[21] Paul Irolla and Eric Filiol, *Glassbox: Dynamic analysis platform for malware android applications on real devices*, arXiv preprint arXiv:1609.04718 (2016).

[22] Jae-wook Jang, Hyunjae Kang, Jiyoung Woo, Aziz Mohaisen, and Huy Kang Kim, *Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. computers & security, 58, 125–138*, Google Scholar Google Scholar Digital Library Digital Library (2016).

[23] Jae-wook Jang and Huy Kang Kim, *Function-oriented mobile malware analysis as first aid*, vol. 2016, Hindawi, 2016.

[24] Jae-wook Jang, Jaesung Yun, Aziz Mohaisen, Jiyoung Woo, and Huy Kang Kim, *Detecting and classifying method based on similarity matching of android malware behavior with profile*, SpringerPlus **5** (2016), no. 1, 273.

[25] Hyunjae Kang, Jae-wook Jang, Aziz Mohaisen, and Huy Kang Kim, *Detecting and classifying android malware using static analysis along with creator information*, vol. 11, SAGE Publications Sage UK: London, England, 2015, p. 479174.

[26] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb, *Maldozer: Automatic framework for android malware detection using deep learning*, Digital Investigation **24** (2018), S48–S59.

[27] Kwanghwan Kim and Hojung Cha, *Wakescope: runtime wakelock anomaly management scheme for android platform*, Proceedings of the Eleventh ACM International Conference on Embedded Software, IEEE Press, 2013, p. 27.

[28] Arash Habibi Lashkari, Gerard Draper-Gil, Mohammad Saiful Islam Mamun, and Ali A Ghorbani, *Characterization of tor traffic using time based features*, ICISSP, 2017, pp. 253–262.

[29] Arash Habibi Lashkari, Andi Fitriah A Kadir, Hugo Gonzalez, Kenneth Fon Mbah, and Ali A Ghorbani, *Towards a network-based framework for android malware detection and characterization*, Proceeding of the 15th international conference on privacy, security and trust, 2017.

[30] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani, *Toward developing a systematic approach to generate benchmark android malware datasets and classification*, 2018 International Carnahan Conference on Security Technology (ICCST), IEEE, 2018, pp. 1–7.

[31] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen, *Droidbot: a lightweight ui-guided test input generator for android*, 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 23–26.

[32] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni, *Understanding and detecting wake lock misuses for android applications*, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 396–409.

[33] Aravind Machiry, Rohan Tahiliani, and Mayur Naik, *Dynodroid: An input generation system for android apps*, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 224–234.

[34] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo, *Machine learning aided android malware classification*, Computers & Electrical Engineering **61** (2017), 266–274.

[35] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero, *Avclass: A tool for massive malware labeling*, International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2016, pp. 230–253.

[36] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss, *"andromaly": a behavioral malware detection framework for android devices*, Journal of Intelligent Information Systems **38** (2012), no. 1, 161–190.

[37] Karina Sokolova, Charles Perez, and Marc Lemercier, *Android application classification and anomaly detection with graph-based permission patterns*, Decision Support Systems **93** (2017), 62–76.

[38] Guillermo Suarez-Tangil and Gianluca Stringhini, *Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned*, arXiv preprint arXiv:1801.08115 (2018).

[39] Laya Taheri, Andi Fitriah Abdul Kadir, and Arash Habibi Lashkari, *Extensible android malware detection and family classification using network-flows and api-calls*, 2019 International Carnahan Conference on Security Technology (ICCST), IEEE, 2019, pp. 1–8.

[40] Kabakus Abdullah Talha, Dogru Ibrahim Alper, and Cetin Aydin, *Apk auditor: Permission-based android malware detection system*, **13** (2015), 1–14.

[41] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro, *Copperdroid: Automatic reconstruction of android malware behaviors*, NDSS, 2015.

[42] Wikipedia Users, *Uml diagram*, Last accessed: on Jan 2020, `https://en.wikipedia.org/wiki/Class_diagram`.

[43] ———, *Component diagram*, Last accessed: on Nov 2019, `https://en.wikipedia.org/wiki/Component_diagram`.

[44] ———, *Sequence diagram*, Last accessed: on Nov 2019, `https://en.wikipedia.org/wiki/Sequence_diagram`.

[45] Ping Wang and Yu-Shih Wang, *Malware behavioural detection and vaccine development by using a support vector model classifier*, Journal of Computer and System Sciences **81** (2015), no. 6, 1012–1026.

[46] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou, *Deep ground truth analysis of current android malware*, International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2017, pp. 252–276.

[47] Michelle Y Wong and David Lie, *Intellidroid: A targeted input generator for the dynamic analysis of android malware*, NDSS, vol. 16, 2016, pp. 21–24.

[48] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu, *Malton: Towards on-device non-invasive mobile malware analysis for art*, 26th USENIX Security Symposium (USENIX Security 17), 2017.

[49] Li Zhang, Vrizlynn LL Thing, and Yao Cheng, *A scalable and extensible framework for android malware detection and family attribution*, Computers & Security **80** (2019), 120–133.

[50] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao, *Semantics-aware android malware classification using weighted contextual api dependency graphs*, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1105–1116.

[51] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou, *Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications*, Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2012, pp. 93–104.

[52] Yajin Zhou and Xuxian Jiang, *Dissecting android malware: Characterization and evolution*, Security and Privacy (SP), 2012 IEEE Symposium on, IEEE, 2012, pp. 95–109.

[53] Michał Łuszczuk, *Getsystem service*, Last accessed: on Jan 2020, `https://blog.propaneapps.com/android/getsystemservice-from-context-to-android`.

Appendix A

GNU Debugger (GDB)

GDB is a command-line based debugger for expert analyzers. To work with GDB on phones, GDB must be connected to a GDBServer, installed on the smartphone. Figure A.1 summarizes the connection structure between GDB and GDBserver installed on a phone.

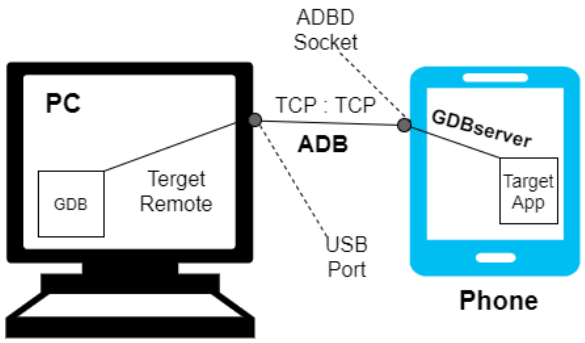


Figure A.1: Overall connection structure between GDB and GDBserver

A.1 GDB Installation Procedure

The following steps address the struggles of GDB installation.

A.1.1 Installing GDBserver on the phone

The below instructions guide us to locate GDBSserver of Android SDK and push it to the connected phone with enabled USB debugging port through ADB. GDBServer must be compatible with the Android device.

```
# find the GDBserver on the PC
:-$ cd /home/.../Android/Sdk/ndk-bundle/
:-$ find . -type f | grep gdbserver


# push GDBserver to the phone through ADB
:-$ adb push  ./prebuilt/android-arm64/gdbserver/gdbserver  /sdcard/
```

GDBserver must be relocated to /system/bin/ with executable privileges in the rooted ADB shell.

```
# in ADB shell with root access
:shell$ cp /sdcard/gdbserver  /system/bin/
:shell$ chmod 777  /system/bin/gdbserver
```

Before running GDBserver, all libraries of the device and emulator must be copied to the local PC for GDB to use later.

```
:-$ mkdir -p /home/.../dbgtmp/
:-$ adb pull /system/lib /home/.../dbgtmp/
:-$ ls -l /home/.../dbgtmp/lib/
```

Also, all native libraries of the target APK must be copied to the local PC.

```
:-$ mkdir tmp
:-$ unzip <TARGETAPK> -d  tmp
:-$ cp tmp/lib/<ARCH>/<NATIVELIBDBG>/home/.../dbgtmp/lib
```

A.1.2 Running GDB and GDBserver

The local GDB and GDBserver must be connected for debugging an App that is running on the phone in real-time.

1) Running GDB on the Phone:

- On GDBserver Shell: The following commands must be run on a rooted phone to start GDBServer.

:-$ adb shell
:shell$ su
:shell$ system/bin/gdbserver :5050 - -attach $(ps | grep [pkg name of the App] | awk 'print $2')

We describe the last command below in more details and Figure A.2 provides the PS command to retrieve App PID number.

:-$ adb shell system/bin/gdbserver :[port number] - -attach [App PID]

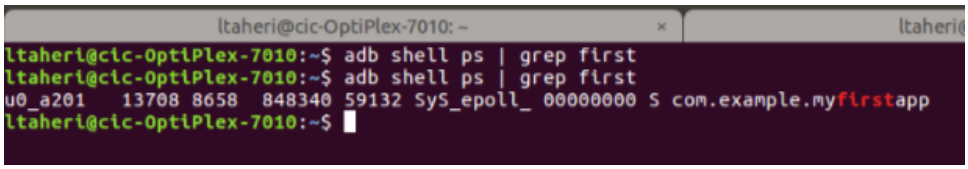


Figure A.2: PS command to retrieve App PID

Figure A.3 shows when GDBserver is connected to an App's PID, it suspends the App to receive a signal from GDB.

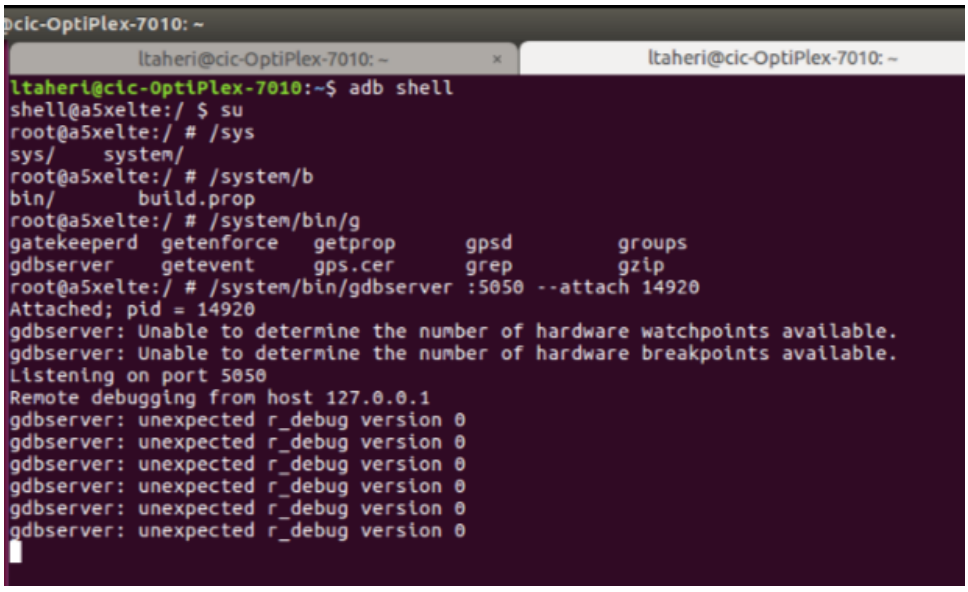


Figure A.3: A suspended App by GDBserver waiting for a signal of a remote GDB

2) Running GDB on the PC:

- On Local PC: The below command hooks the local GDB to the GDBServer through TCP ports.

:-$ adb forward tcp:5050 tcp:5050
:-$ /home/.../Android/Sdk/ndk-bundle/prebuilt/linux-x86_64/bin/gdb

- Inside GDB Shell: This command connects GDB shell to GDBServer shell.

(gdb) target remote :5050

As Figure A.4 presents, the following commands assist GDB to access to the copied native libraries of APK, which was mentioned in A.1.1.

(gdb) set solib-search-path /home/.../dbgtmp/lib
(gdb) info sharedlibrary

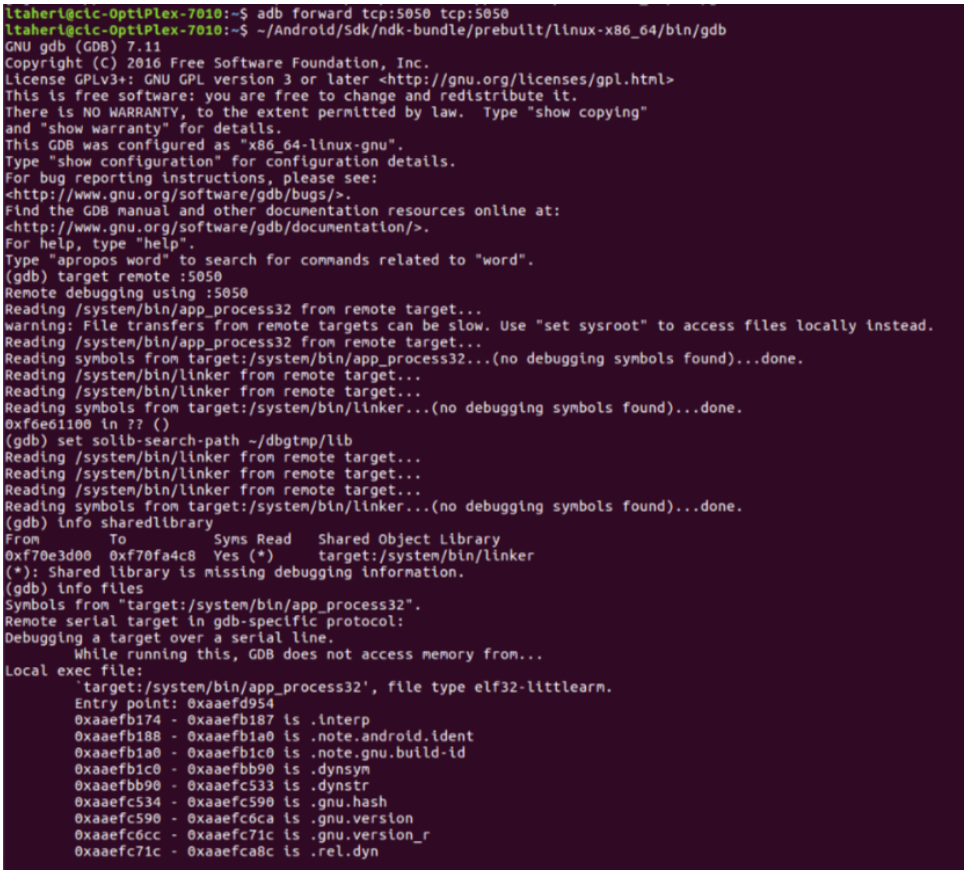


Figure A.4: Running GDB on PC

When GDB and GDBserver are connected, and all breakpoints are set, the debugging analysis will start. For example, Figure A.5 presents some GDB command that can be used in debugging analysis.

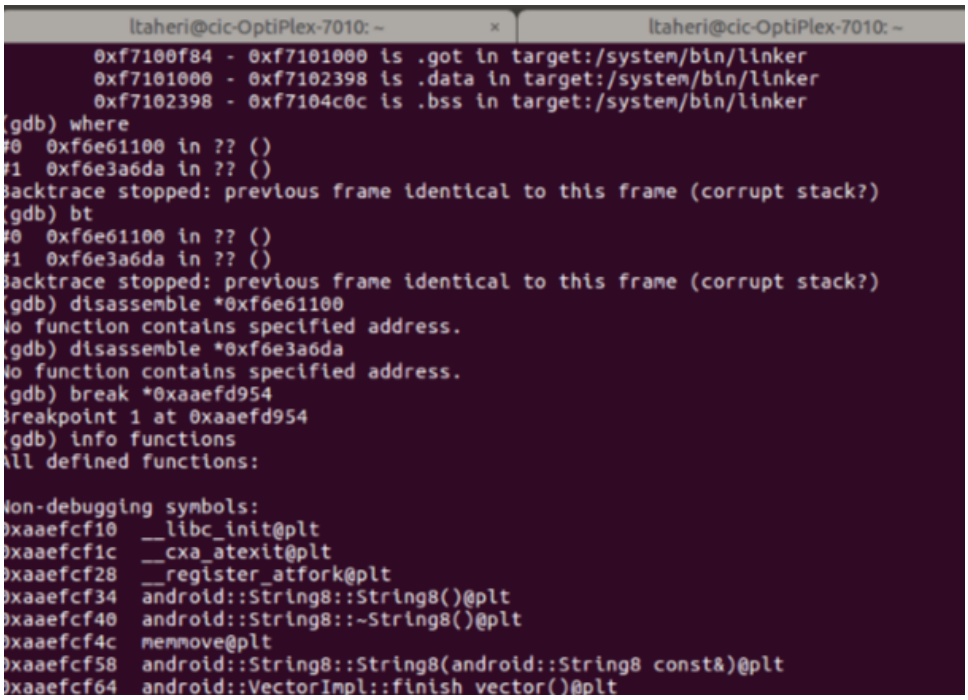


Figure A.5: Examples of commands of GDB

Appendix B

All WakeLock List Tables

Signature	**check-cast v0, <t: AlarmManager>**
APK Hash	42d735fac7ad7576922f0027c591622f
APK Category	Adware
APK Family	Youmi
Java Code	AlarmManager localAlarmManager = (AlarmManager)this.b.getSystemService("alarm");
Smali Code Format	const-string/jumbo v4, "alarm" invoke-virtual {v0, v4}, Landroid/content/Context;->getSystemService(Ljava/lang/String;)Ljava/lang/Object; move-result-object v0 check-cast v0, Landroid/App/AlarmManager;
Javap Code Format	aload_0 getfield // Field b:Landroid/content/Context; ldc_w // String alarm invokevirtual // Method android/content/Context.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/App/AlarmManager
IDA Code Format	iget-object v0, this, n_b const-string/jumbo v4, aAlarm_1 # "alarm" invoke-virtual {v0, v4}, <ref Context.getSystemService(ref) imp. @ _def_Context_getSystemService@LL_0> move-result-object v0 check-cast v0, <t: AlarmManager> invoke-virtual {v0, v1}, <void AlarmManager.cancel(ref) imp. @ _def_AlarmManager_cancel@VL> const/4 v4, 1 invoke-virtual {v0, v4, v2, v3, v1}, <void AlarmManager.set(int, long, ref) imp. @ _def_AlarmManager_set@VIJL>

Signature	**check-cast v0, <t: AudioManager>**
APK Hash	42d735fac7ad7576922f0027c591622f
APK Category	Adware
APK Family	Youmi
Java Code	paramContext = (AudioManager)paramContext.getSystemService("audio");
Smali Code Format	const-string/jumbo v0, "audio" invoke-virtual {p0, v0}, Landroid/content/Context;->getSystemService(Ljava/lang/String;)Ljava/lang/Object; move-result-object v0 check-cast v0, Landroid/media/AudioManager;
Javap Code Format	ldc // String audio invokevirtual // Method android/content/Context.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/media/AudioManager
IDA Code Format	const-string/jumbo v0, aAudio # "audio" invoke-virtual {p0, v0}, <ref Context.getSystemService(ref) imp. @ _def_Context_getSystemService@LL_0> move-result-object v0 check-cast v0, <t: AudioManager> invoke-virtual {v0}, <int AudioManager.getRingerMode() imp. @ _def_AudioManager_getRingerMode@I> move-result v4 if-nez v4, loc_2BE104

Signature	check-cast v0, <t: DownloadManager>
APK Hash	119658211908d17d4f78f36168477af1
APK Category	Scareware
APK Family	VirusShield
Java Code	Object localObject = (DownloadManager)paramContext.getSystemService("download");
Smali Code Format	const-string v0, "download" invoke-virtual {p1, v0}, Landroid/content/Context;->getSystemService(Ljava/lang/String;)Ljava/lang/Object; move-result-object v0 check-cast v0, Landroid/App/DownloadManager;
Javap Code Format	getfield // Field b:Landroid/content/Context; ldc // String download invokevirtual // Method android/content/Context.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/App/DownloadManager
IDA Code Format	const-string v0, aDownload # "download" invoke-virtual {p0, v0}, <ref Context.getSystemService(ref) imp. @ _def_Context_getSystemService@LL_0> move-result-object v0 check-cast v0, <t: DownloadManager>

Signature	const-string v3, aGcmLib # "GCM_LIB"
APK Hash	fd4b9f97d3534c009ccd95858032eb51
APK Category	Scareware
APK Family	FakIobOffer
Java Code	sWakeLock = ((PowerManager)paramContext.getSystemService("power")).newWakeLock(1, "GCM_LIB");

Smali Code Format	const-string v1, "power"
	invoke-virtual {p0, v1}, Landroid/content/Context;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;
	move-result-object v0
	check-cast v0, Landroid/os/PowerManager;
	const/4 v1, 0x1
	const-string v3, "GCM_LIB"
	invoke-virtual {v0, v1, v3}, Landroid/os/PowerManager;->newWakeLock(ILjava/lang/String;)Landroid/os/PowerManager$WakeLock;
	move-result-object v1
	sput-object v1, Lcom/google/android/gcm/GCMBaseIntentService;-¿sWakeLock:Landroid/os/PowerManager$WakeLock;

Javap Code Format	ldc // String power invokevirtual // Method android/content/Context.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/os/PowerManager iconst_1 ldc // String GCM_LIB invokevirtual // Method android/os/PowerManager.newWakeLock:(ILjava/lang/String;)Landroid/os/PowerManager$WakeLock; putstatic // Field sWakeLock:Landroid/os/PowerManager$WakeLock; ldc // String GCMBaseIntentService ldc // String Acquiring WakeLock invokestatic // Method android/util/Log.v:(Ljava/lang/String;Ljava/lang/String;)I pop getstatic // Field sWakeLock:Landroid/os/PowerManager$WakeLock; invokevirtual // Method android/os/PowerManager$WakeLock.acquire:()V
IDA Code Format	sget-object v1, GCMBaseIntentService_sWakeLock if-nez v1, loc_642AC const-string v1, aPower # "power" invoke-virtual {p0, v1}, <ref Context.getSystemService(ref) imp. @ _def_Context_getSystemService@LL> move-result-object v0 check-cast v0, <t: PowerManager> const/4 v1, 1 const-string v3, aGcmLib # "GCM_LIB" invoke-virtual {v0, v1, v3}, <ref PowerManager.newWakeLock(int, ref) imp. @ _def_PowerManager_newWakeLock@LIL> move-result-object v1 sput-object v1, GCMBaseIntentService_sWakeLock

Signature	check-cast v0, <t: LocationManager>
APK Hash	3289b7c5167d764081e6b8e379bd46c6
APK Category	Adware
APK Family	Shuane
Java Code	LocationManager localLocationManager = (LocationManager)paramContext.getSystemService("location");
Smali Code Format	const-string v0, "location" invoke-virtual {p0, v0}, Landroid/content/Context;->getSystemService(Ljava/lang/String;)Ljava/lang/Object; move-result-object v0 check-cast v0, Landroid/location/LocationManager;
Javap Code Format	ldc_w // String location invokevirtual // Method android/content/Context.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/location/LocationManager
IDA Code Format	const-string v0, aLocation_1 # "location" invoke-virtual {p0, v0}, <ref Context.getSystemService(ref) imp. @ _def_Context_getSystemService@LL> move-result-object v0 check-cast v0, <t: LocationManager> const-string v1, aAndroidPermiss_0 # "android.permission.ACCESS_FINE_LOCATION" invoke-static {p0, v1}, <boolean b.a(ref, ref) b_a@ZLL_0> move-result v1 if-eqz v1, loc_3EEE7A const-string v1, aGps # "gps" invoke-virtual {v0, v1}, <ref LocationManager.getLastKnownLocation(ref) imp. @ _def_LocationManager_getLastKnownLocation@LL> move-result-object v1

Signature	const-string v2, aPenetratedicti # "PenetrateDictionariesLock"
APK Hash	d9626d54c63f8ce4c32e9061b62009d0
APK Category	Scareware
APK Family	Penetho
Java Code	this.mWakeLock = ((PowerManager)paramActivity.getSystemService("power")).newWakeLock(268435482, "PenetrateDictionariesLock");

Smali Code Format	const-string v1, "power"
	invoke-virtual {p1, v1}, Landroid/App/Activity;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;
	move-result-object v0
	check-cast v0, Landroid/os/PowerManager;
	.local v0, "pm":Landroid/os/PowerManager;
	const v1, 0x1000001a
	const-string v2, "PenetrateDictionariesLock"
	invoke-virtual {v0, v1, v2}, Landroid/os/PowerManager;->newWakeLock(ILjava/lang/String;)Landroid/os/PowerManager$WakeLock;
	move-result-object v1

Javap Code Format	putfield // Field mActivity:Landroid/App/Activity; aload_0 aload_1 ldc // String power invokevirtual // Method android/App/Activity.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/os/PowerManager ldc // int 268435482 ldc // String PenetrateDictionariesLock invokevirtual // Method android/os/PowerManager.newWakeLock:(ILjava/lang/String;)Landroid/os/PowerManager$WakeLock; putfield // Field mWakeLock:Landroid/os/PowerManager$WakeLock; aload_0
IDA Code Format	const-string v1, aPower # "power" invoke-virtual {ctx, v1}, <ref Activity.getSystemService(ref) imp. @ _def_Activity_getSystemService@LL> move-result-object v0 check-cast v0, <t: PowerManager> const v1, 0x1000001A const-string v2, aPenetratedicti # "PenetrateDictionariesLock" invoke-virtual {pm, v1, v2}, <ref PowerManager.newWakeLock(int, ref) imp. @ _def_PowerManager_newWakeLock@LIL> move-result-object v1 iput-object v1, this, DownloadFileTask_mWakeLock

Signature	check-cast v0, <t: Vibrator>
APK Hash	14292932679d6930f521a21de4e8bffd
APK Category	Scareware
APK Family	AndroidDefender
Java Code	Vibrator localVibrator = (Vibrator)MainApp.a().getSystemService("vibrator");
Smali Code Format	const-string v6, "vibrator" invoke-virtual {v5, v6}, Landroid/App/Application;->getSystemService(Ljava/lang/String;)Ljava/lang/Object; move-result-object v5 check-cast v5, Landroid/os/Vibrator;
Javap Code Format	invokevirtual // Method getApplication:()Landroid/App/Application; ldc_w // String vibrator invokevirtual // Method android/App/Application.getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/os/Vibrator
IDA Code Format	move-result-object v0 const-string v1, aVibrator # "vibrator" invoke-virtual {v0, v1}, <ref MainApp.getSystemService(ref) imp. @ _def_MainApp_getSystemService@LL> move-result-object v0 check-cast v0, <t: Vibrator> const-string v1, aVibrateprogres # "vibrateProgress" iget v2, this, abo_a invoke-static {v1, v2}, <int zp.a(ref, int) zp_a@ILI> move-result v1

Signature	check-cast v0, <t: WindowManager>
APK Hash	37b993b5f59bbed7538265885429e4c4
APK Category	Adware
APK Family	Dowgin
Java Code	Object localObject1 = (WindowManager)paramContext.getSystemService("window");
Smali Code Format	const-string v0, "window" invoke-virtual {p0, v0}, Lcom/bringmore/giugogdrmsss/FlashLightService;->getSystemService(Ljava/lang/String;)Ljava/lang/Object; move-result-object v0 check-cast v0, Landroid/view/WindowManager;
Javap Code Format	ldc // String window invokevirtual // Method getSystemService:(Ljava/lang/String;)Ljava/lang/Object; checkcast // class android/view/WindowManager
IDA Code Format	move-result-object v0 iput-object v0, this, stru_9290 const-string v0, aWindow # "window" invoke-virtual {p0, v0}, <ref Context.getSystemService(ref) imp. @ _def_Context_getSystemService@LL> move-result-object v0 check-cast v0, <t: WindowManager> new-instance v1, <t: DisplayMetrics> invoke-direct {v1}, <void DisplayMetrics.<init>() imp. @ _def_DisplayMetrics__init_@V> invoke-interface {v0}, <ref WindowManager.getDefaultDisplay() imp. @ _def_WindowManager_getDefaultDisplay@L> move-result-object v0

Appendix C

Procedure of Rooting Android

To access the root privileges manually on an Android device such as Samsung Galaxy A5 or Nexus 5, the following steps are required [1]:

Note: Follow these steps on your cautions because they are susceptible, and a simple mistake can damage the device permanently.

- Step 1: Install Odin Windows software that is compatible with the Android device Model. For example, Odin V3.09 [2] is suitable for Galaxy A5.
- Step 2: Download the original Samsung firmware that is compatible with the device model. Check manufactured (S/N, CSC, PDA) phone information by dialing *#1234# code.
- Step 3: In the phone setting - developer option, enable USB debugging, and unlock OEM.
- Step 4: Open Odin3 as an administrator with the setups showed in Figure C.1 (check AP, Auto Reboot, F. Reset Time items) and click start. Wait for the device to reboot and the green "Pass!" button.

[1] https://android.stackexchange.com/questions/39623/android-adb-shell-recovery-mounting/112598

[2] https://odindownloader.com/download/odin3-v3-09

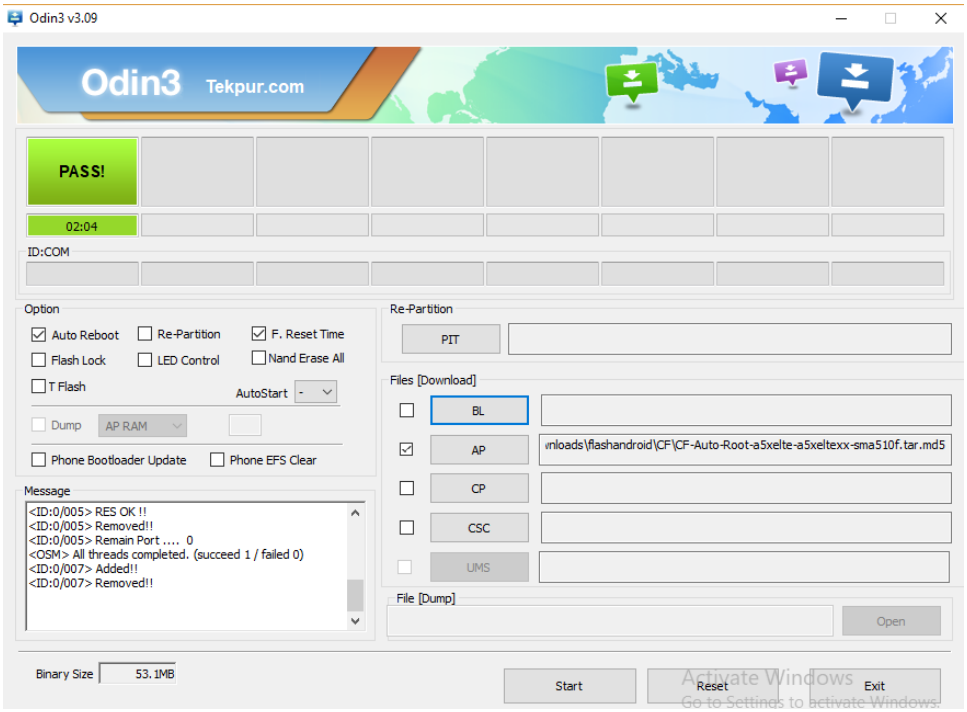


Figure C.1: A Successful booting with Odin3 to Achieve the root access on the Nexus 5 Android device

- Step 5: Check if the device is successfully rooted with RootChecker [3] App (see Figure C.2), or use the following command:

:-$ adb shell su

[3]https://play.google.com/store/apps/details?id=com.joeykrim.rootcheck&hl=en_CA

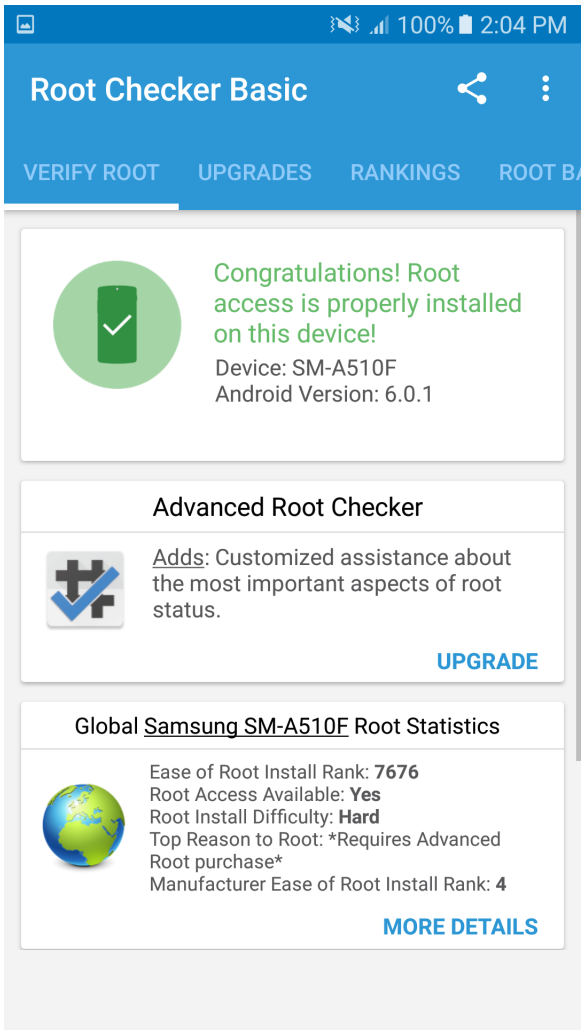


Figure C.2: The RootChecker App screenshot for testing the achieved root access on the device

Glossary

IDC language [12]	Is a C-liked language used in IDA Pro.
Disassembler [17]	Software to transform an executable code to its corresponding assembly language.
Debugger [15]	Software to interrupt the execution processes of a program and change its running states.
Decompiler [16]	Software to revert a compiled program to its relevant higher-level programming languages.

Vita

Candidate's full name:
Laya Taheri

University attended:

- M.Sc. in Computer Science in Cybersecurity field, under the supervision of professor Ali A. Ghorbani, University of New Brunswick (UNB), Fall 2017-Winter 2020
- B.Sc. degree in Computer Engineering, Shahid Bahonar University of Kerman (SBUK), Iran, Winter 2011-Summer 2016

Publications:

- Laya Taheri and Ali A. Ghorbani, "**Impact of Background WakeLock Features in Android Malware Classification**", Under the review of the 17th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2020, Lisboa, Portugal.
- Laya Taheri, Andi Fitriah Abdul Kadir, Arash Habibi Lashkari, "**Extensible Android Malware Detection and Family Classification Using Network-Flows and API-Calls**", In the proceedings of the IEEE (53rd) International Carnahan Conference on Security Technology (ICCST) 2019, Chennai, India.
- Arash Habibi Lashkari, Andi Fitriah A.Kadir, Laya Taheri, and Ali A. Ghorbani, "**Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification**", In the proceedings of the 52nd IEEE International Carnahan Conference on Security Technology (ICCST) 2018, Montreal, Quebec, Canada, 2018.

Conference Presentations:

- Laya Taheri, Arash Habibi Lashkari and Ali A. Ghorbani, "**Espionage Droid Sandbox (EDS)**", Poster presented at the ACM Canadian Celebration of Women in Computing (CAN-CWiC), Halifax, Nova Scotia, 2018.

- Arash Habibi Lashkari, Andi Fitriah A.Kadir, Laya Taheri, and Ali A. Ghorbani, "**Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification**", Poster presented at the 15th Annual Research Exposition of Faculty of Computer Science, UNB, 2018.