DETECTING PRIVACY LEAKS THROUGH EXISTING ANDROID FRAMEWORKS

Parul Khanna

A thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

in Information Systems Security at

Concordia University

Montréal, Québec, Canada

April 2017

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Parul Khanna**

Entitled: **Detecting Privacy Leaks Through Existing Android Frameworks**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Arash Mohammadi _____ Chair

Dr. Jeremy Clark _____ Examiner

Dr. Ashutosh Bagchi _____ External Examiner

Dr. Mohammad Mannan _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 2017 _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Detecting Privacy Leaks Through Existing Android Frameworks

Parul Khanna

The Android application ecosystem has thrived, with hundreds of thousands of applications (apps) available to users; however, not all of them are safe or privacy-friendly. Analyzing these many apps for malicious behaviors is an important but challenging area of research as malicious apps tend to use prevalent stealth techniques, e.g., encryption, code transformation, and other obfuscation approaches to bypass detection. Academic researchers and security companies have realized that the traditional signature-based and static analysis methods are inadequate to deal with this evolving threat. In recent years, a number of static and dynamic code analysis proposals for analyzing Android apps have been introduced in academia and in the commercial world. Moreover, as a single detection approach may be ineffective against advanced obfuscation techniques, multiple frameworks for privacy leakage detection have been shown to yield better results when used in conjunction.

In this dissertation, our contribution is two-fold. First, we organize 32 of the most recent and promising privacy-oriented proposals on Android apps analysis into two categories: static and dynamic analysis. For each category, we survey the state-of-the-art proposals and provide a high-level overview of the methodology they rely on to detect privacy-sensitive leakages and app behaviors. Second, we choose one popular proposal from each category to analyze and detect leakages in 5,000 Android apps. Our toolchain setup consists of IntelliDroid (static) to find and trigger sensitive API (Application Program Interface) calls in target apps and leverages

TaintDroid (dynamic) to detect leakages in these apps. We found that about 33% of the tested apps leak privacy-sensitive information over the network (e.g., IMEI, location, UDID), which is consistent with existing work. Furthermore, we highlight the efficiency of combining IntelliDroid and TaintDroid in comparison with Android Monkey and TaintDroid as used in most prior work. We report an overall increase in the frequency of leakage of identifiers. This increase may indicate that IntelliDroid is a better approach over Android Monkey.

Acknowledgments

First and foremost, I want to thank my supervisor, Dr. Mohammad Mannan of Concordia Institute for Information Systems Engineering (CIISE) for the time he spent discussing with me on ideas and contributions during my studies. Dr. Mannan has taught me to focus my thoughts and how to reason at an abstract level to clearly present ideas and express them. I am fortunate to have him as my supervisor (and mentor) who not only takes initiative and contributes to the research projects of his students but also cares about professional development and career advancement.

I want to thank my lab mates Lianying Zhao (Viau) and Xavier de Carné de Carnavalet for their continuous feedback during our numerous discussions and meetings. They have been very generous for their help during the course of my research. I would also like to thank my friends Deepak Kumar Aggarwal and Nada Khabouchi for supporting me during my stressful and difficult moments.

I am thankful to David Barrera and Furkan Alaca for sharing the CCSL observatory dataset.

I am also grateful for the financial support provided by Dr. Mannan, Concordia University, Mitacs-Accelerate and Irdeto Canada corporation for funding my studies.

I want to thank my brother Prince Khanna for providing me with unfailing support and my mother for continuous moral support throughout my years of study. They have made this endeavor even more worthwhile. Finally, I dedicate this thesis to the memory of my beloved father under whose careful protection I have been able

to enjoy my life. His teachings and blessings have always enabled me to accomplish milestones in my life successfully.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android is an operating system designed for mobile devices such as smartphones and tablets. It provides a lot of features for users but its openness empowers any developer to write applications and thus extend its feature set. The increasing market share of the Android platform is partly caused by a growing number of apps available on the Android market. App developers can easily upload their applications to the Android Market or other third-party markets with little if any, security vetting processes. Google makes use of a dynamic analysis tool known as Google Bouncer [31], an automated system that screens submissions to the Google Play Store. Even though Google Bouncer can detect some of the malicious applications, malicious applications have been successful in bypassing the vetting process [28]. Therefore, it is possible for users to download a malicious application from the Google Play Store.

The Android application framework comes with default security features aimed at restricting what applications can do. It features a fine-grained permission system allowing the user to review the permissions app requests and grant or deny access to resources. However, recent studies have found that permissions cannot ensure all of the security properties [40]. Consequently, there are security guarantees users cannot have from the system. In practice, many applications are doing malicious activities.

For example, in November 2016, researchers uncovered a family of Android-based malware namely Gooligan [11], that compromised more than 1 million Google accounts, hundreds of them were associated with enterprise users. According to a report by Forbes [21], the infection begins when a user downloads and installs a malware-infected app on a vulnerable Android device. Gooligan then downloads a rootkit from the Command and Control server that takes advantage of multiple Android exploits. In July 2016, another malware called HummingBird [29] was uncovered. The main purpose of the HummingBird malware was to trick users into clicking on mobile and web advertisements to generate advertising revenue for its parent company.

As these malicious apps become widespread and risks increase, Android's security is more and more studied. A number of static analysis [20, 69, 26, 3, 5, 4, 63] and dynamic [19, 55, 30, 32, 22] analysis frameworks have been proposed to detect privacy-sensitive behaviors in Android applications. For example, TaintDroid presented by Enck et al. [19] can detect privacy leaks by using a dynamic taint-tracking method. TaintDroid can provide useful results if it is provided with the direct inputs to trigger malicious behavior from the application. To this end, Wong et al. introduced IntelliDroid [63], a generic Android input generator that can be configured to produce inputs specific to a dynamic analysis tool. Gilbert et al. [22] tested a variety of categories of applications by generating random user events for 30 minutes. However, this can only achieve 40% or less code coverage in all cases. Hornyack et al. presented AppFence [30], a dynamic system implemented as modifications to the Android framework that prevents attacks against user privacy via data shadowing.

On the other hand, for static analysis of applications, one of the first approaches was Kirin [20], which recovers the set of permissions requested by applications with the goal of identifying potentially malicious behaviors. Other existing works include RiskRanker [4] and DroidRanger [69], which rely on symbolic execution and a set of

heuristics to detect unknown malicious applications. FlowDroid [5] and DroidSafe [26] propose precise static taint analyses to detect potentially malicious data flows. As a way of helping users to think about the permissions requested by various apps, and make informed decisions, Lin et al. [36, 48, 35] proposed a system called *PrivacyGrade*. In particular, PrivacyGrade uses static analysis and crowd-sourcing to capture user expectations of sensitive resources used by mobile applications. It later summarizes each app's privacy in the form of a grade ranging from A to D.

Summary of these proposals shows there is not a clear-cut solution that addresses every issue. Either static analysis or dynamic analysis, both approaches can be used separately or in conjunction, but each one has its own limitations. Malicious applications can fool static analysis frameworks by employing encryption and/or transformation techniques [68, 47]. Also, dynamic analysis frameworks can be evaded by anti-emulation techniques [39]. Since there is no robust proposal that addresses all the issues, it is very interesting to survey these proposals and compare them analytically against each other.

In existing app analysis proposals, many researchers rely on dynamic analysis to detect privacy-sensitive behaviors and often use Android emulators for the experiments. However, existing Android emulators possess a couple of limitations. For example, they cannot emulate IMEI, MAC and GPS sensors - components like these are likely to be used by malware applications. Also, malware apps are capable of detecting emulation and as a result can behave normally by not doing any kind of malicious activity [16].

It is interesting to replicate existing frameworks with real applications (on a real device) and evaluate a large number of apps for privacy leakages. Therefore, in this dissertation, we adapt popular static and dynamic analysis frameworks that rely on approaches for automatically extracting behavior of Android applications. To ensure

better coverage of functionality and permissions used by applications, we leverage IntelliDroid [63], a generic targeted input generator for Android applications. IntelliDroid obtains information of each app by decompiling the APKs into classes and determines trigger paths for all the functions in the APK. On the other hand, we use TaintDroid to detect privacy-sensitive behavior dynamically. TaintDroid alerts information leaks inside an Android app via dynamic taint tracking. It uses the concepts of taint sources, from which sensitive information (e.g., IMEI, text messages, contacts or GPS data) is obtained, and taint sinks, which are interfaces to the outside world (e.g., using data networks or sending SMSs) where tainted information is usually not expected to be sent. When tainted data reaches a taint sink, TaintDroid issues a warning to the user in the form of notification (more detailed information on TaintDroid is provided in Chapter 3).

In IntelliDroid's original work, the authors tested IntelliDroid and TaintDroid on malware samples from the Android Malware Genome Project [65]. The framework successfully detected privacy leaks in malware samples. However, in our analysis, we focus on using IntelliDroid and TaintDroid with real-world applications, both from the official Android market and third party markets.

In particular, this dissertation comprehensively surveys the state-of-the-art app behavioral analysis proposals and highlights the methodology they rely on to detect privacy-sensitive leakages. Also, we analyze 5000 real-world Android applications using two popular frameworks (static and dynamic) and characterize application behavior for security and privacy leakages. We found that more than one-third of the tested apps leak privacy-sensitive information over the network. We also report an overall increase in the frequency of leakage of identifiers. The following sections hightlight more details about our objectives, contributions, and outline for this dissertation.

1.1 Objectives

We structure the thesis around two objectives:

1. Analytically compare existing Android application analysis frameworks.

2. Test application behaviors under different frameworks.

   The two objectives have been validated through rigorous experiments as complete as possible.

1.2 Contributions

The primary contributions of our work are as follows:

1. **Survey**. We survey the state-of-the-art analysis platforms and provide a high-level overview of the methodology they rely on to detect privacy leakage and application behavior. In total, we characterize 32 comprehensive app analysis frameworks with a focus on securing the applications from privacy leaks and detecting malicious apps for sensitive data leakage.

2. **Evaluation**. We make an in-depth evaluation of 5000 real world Android applications to characterize application behavior for privacy violations such as attempts to retrieve user's sensitive information, send SMS, or access location of the device. Experiments include analysis with existing static and dynamic analysis frameworks, application classification based on leakage, and behavior characterization. We also conduct a cross-validation experiment to analyze the behavior of applications using different techniques.

3. **A Scalable Test Framework.** Given the enormous growth of the number of applications, it is very hard to analyze more and more applications in a

given period of time. In general, analyzing a large set of applications manually is very time consuming and error-prone process. To overcome this state of affairs, we write scripts and made changes in framework source to automate the analysis process. For example, we fully automate the static and dynamic analysis components of IntelliDroid for handling large volumes of applications and satisfy our setup requirements.

1.3 Outline

The rest of the document is organized in the following way:

1. In Chapter 2, we provide an overview of the background information about Android architecture along with its permission enforcement scheme.

2. In Chapter 3, we provide a taxonomy of existing static and dynamic analysis tools and frameworks.

3. In Chapter 4, we shed light on our threat model.

4. In Chapter 5, we provide the detailed explanation of our evaluation methodology.

5. In Chapter 6, we provide information about our analysis dataset & setup details. We also provide insights on why we use a real device instead of a simple Android emulator.

6. In Chapter 7, we provide experiment results and observations.

7. In Chapter 8, we discuss the limitations of our evaluation.

8. In Chapter 9, we provide conclusion and a brief discussion for this dissertation.

Chapter 2

Background

Before we discuss the details of our toolchain setup and methodology, it is important to understand how Android and its applications work. In this chapter, we provide a short introduction into the Android system architecture. In the end of this chapter, we also shed light on Android's security mechanism.

2.1 Android System Architecture

The Android operating system is built on the top of the Linux kernel and organized in a layered architecture consisting of four layers: (i) the Linux kernel, (ii) Android's native system libraries and Dalvik virtual machine runtime, (iii) Android's application frameworks, and (iv) a collection of installed applications. Figure 1 and the following sections briefly describe the basic blocks of an Android's system architecture.

2.1.1 Linux Kernel

The bottom most layer in the Android architecture is known as the *Linux kernel*. It helps in abstraction between the hardware of the device and contains all the key drivers for the Android based phones to work. It provides services such as memory

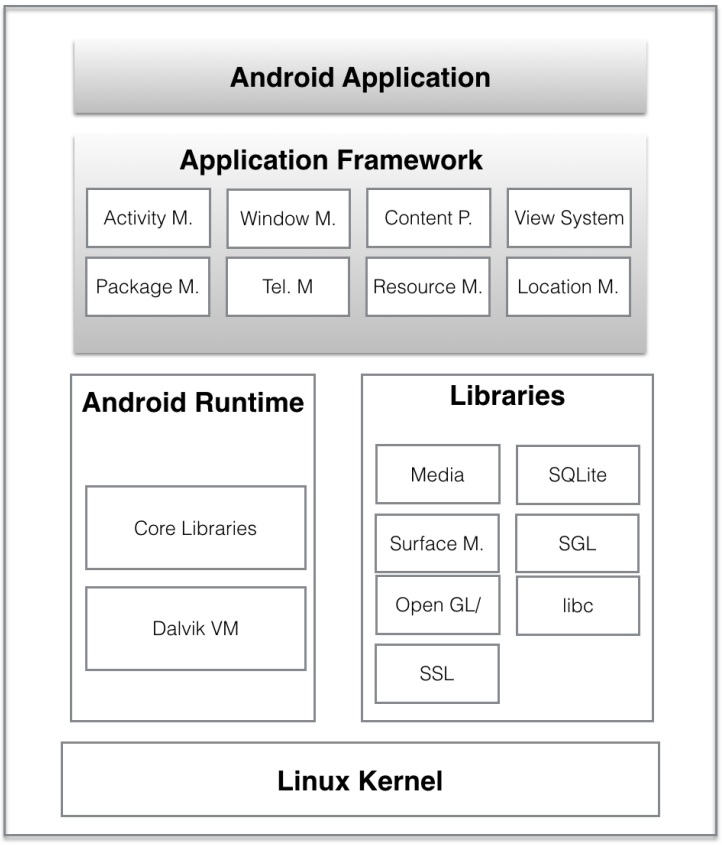

Figure 1: Android system architecture. TaintDroid is implemented inside the Dalvik VM and can inspect all the Java-based tasks (in grey).

and process management, access control, and a driver framework. Android uses a customized version of the Linux kernel with a few special additions. These include wake-locks, a memory management system that is more aggressive in preserving memory, the Binder IPC driver (to mediate interactions between apps), and other features that are important for a mobile embedded platform like Android.

2.1.2 Libraries

On the top of Linux kernel sits the set of *libraries*. The libraries consist of C and C++ code that compiles to the native binary format. The functionality in these libraries is exposed to applications from the third party developers through the Android framework.

2.1.3 Android Runtime

This is the third section of the architecture and sits next to the libraries. It has the key component named DVM (Dalvik Virtual Machine). It is a Java virtual machine specially designed for Android. DVM uses Linux features like memory management and multi-threading which are also used by the Java programming language. With the help of DVM, every Android application has its own process and instance. DVM performs the compilation and execution of Java code each time the application is launched.

The recent version of Android uses a different approach for the Runtime. Android introduced ART Runtime with the Android KitKat (Android 4.4) that applies Ahead-of-Time (AoT) compilation to convert Dalvik bytecode into native code. ART uses a different approach as compared to the DVM. ART compiles the APK into machine code during the process of installation of APK file. Compilation in ART is done during installation of the APK.

2.1.4 Application Framework

Application framework provides applications in the form of rich Java classes which are used by the developers to design their own app. Most components in this layer are implemented as applications and run as background processes on the device. Many components are responsible for managing basic phone functions like receiving phone calls, or text messages or monitoring power usage. It consists of the application manager, content provider, notification manager and location manager.

2.1.5 Applications

The top most layer in Android system architecture is the application layer and is responsible for the interaction between end users and the device. The general applications are installed in this layer only. Address book, browser, games are few examples of such applications installed in this layer.

2.2 Android Applications

Android application package (APK) files are used to distribute and install application software and middleware on Android operating system. APK files are the ZIP file formatted packages based on the JAR file format which has .apk extensions

To secure the applications, Android application sandbox helps to isolate the app data and code execution from the other apps. It provides an application framework with a common security functionality like cryptography, permissions, and IPC. As per the documentation of Android [24], it implements the principle of least privilege as each application has access to only the components it requires to do its work.

2.2.1 Application Components

We now outline a number of core application components that are used to build Android apps. For information on Android application fundamentals, we refer to the official documentation [24].

2.2.1.1 Activities

An *activity* represents the visual view of an app. Usually, an app consists of a list of Activities, instances of which are loaded every time the user is trying to interact with the app in the foreground. Activities together can be termed as the face of an

app. Activities, in turn, use Fragments, Views to render UI (user interface) related entities. A social networking app, for example, might have the possibility to start the music app's play activity to start playback of a received audio file. Also, Activities consists of various states:

1. An activity is present in *active* or *running* states if it is in the foreground.

2. An activity is in *paused* state if it has lost focus but is still visible for the end-user. Though the activity is paused it retains a copy of its state and other information from it is the active state.

3. An activity is in *stopped* state if it is no longer visible to end-user. The user can still retrieve the state of the app if the app is still running. Under the low memory conditions, Android system will often kill these no longer used apps to free the memory so that memory can be used by other active applications.

Storing the state information of an Activity can be facilitated using various methods available as part of the Activity API in Android.

2.2.1.2 Services

Services are components that run in the background to perform long-running operations such as playing music, handle network transactions, interacting content providers etc. It does not have any UI. The music application, for example, will have a music service that is responsible for playing music in the background while the user is in a different application. Services can be started by other components of the app such as an activity or a broadcast receiver. Moreover, service can run in the background indefinitely even if the application is destroyed.

2.2.1.3 Content Provider

Content providers make a specific set of the application's data available to other applications. They manage a shared set of application data. For example, contact information data is stored in a content provider so that other applications can query it whenever they require. A music player may use a content provider to store information about the current song being played, which could be further used by a social media application to update the user's 'current music playing' status.

2.2.1.4 Broadcast Receivers

Broadcast receivers respond to broadcast messages from other applications or from the system itself. These messages are sometimes called *events* or *intents*. For example, the SMS app broadcasting information about an SMS has being received and let other applications know about the ongoing event. Broadcast receivers do not have a user interface and are generally used to act as a gateway to other components. They might, for example, initiate a background service to perform some work based on a specific event.

2.2.1.5 Intents

Intents are asynchronous messages which allow application components to request functionality from other Android components. Intents allow users to interact with components from the same applications as well as with components contributed by other applications. For example, an activity can start an external activity for taking a picture.

2.3 APK File Architecture

The previous sections briefly outlined various components of Android system which are necessary for an app to run on Android. This section gives an overview of various components of an APK file. An APK (application package) file is the file format used to distribute applications in Android. APK packages comprise of the key component of an Android application which is the dex file created after compiling the bytecode of all the java source files in Android. Following are the details about other components present in an APK file.

1. **Android Manifest.** The *Android manifest* presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Every application comes with an *AndroidManifest.xml* file that informs the system about the app's components. The AndroidManifest.xml file contains information about application package, including components of the application such as activities, services, broadcast receivers, content providers etc. The Androidmanifest.xml also specifies application requirements such as special hardware requirements (e.g., accessing camera or GPS sensor), or the minimum API version necessary to run an app. To access protected components (e.g., location access, or access to sd card), an application needs to be granted permission. All necessary permissions must be defined in the app's AndroidManifest.xml. This way, during installation, the Android OS can prompt the user with an overview of used permissions after which a user explicitly has to grant the app access to use these components.

2. **lib.** This directory usually contains compiled the code of various supporting libraries which are referred usually in the application components. This directory, in turn, has different directories representing the processor base for which

the libraries are compiled for e.g., arm, x86, MIPS etc.

3. **res.** This directory contains the resources which are used in the application. An example of resources can be various images required in the UI layout of an Activity.

4. **assets.** This directory contains application assets. These can be accessed programmatically in Android using AssetManager.

5. **classes.dex.** The dex file generated after compiling the bytecodes of all the java source files in the applications.

6. **resources.arsc.** This is a precompiled binary of the contents of resource directory mentioned above.

7. **META-INF.** In order for the system to maintain a unique identity of authors of applications, apps are required to be signed before installation in Android. The contents of this directory contain the manifests and certificates of its digital signature.

2.4 Android Security Mechanisms

This section provides an overview of the Android security mechanisms. The focus of Android security is primarily about protecting user data, system resources, and app isolation. To achieve these goals, Android provides the following features;

2.4.1 Sandbox

Application sandbox is a means to isolate the applications from each other in the Android system by assigning a UID and a set of permissions.

When the application is installed on the device, it runs in its own sandbox and other applications cannot access or interfere. An application can only access its own files unless other applications explicitly assign the access permissions to this application. For example, if the applications are created by the same developers, the developers can make these applications share the same UID, then these applications will run in the same sandbox and share the resources in that sandbox.

2.4.2 Application Signing

Application signing is used to ensure the application security. It creates a certification between developers and their applications.

Before placing an application into its sandbox, the application signing creates a relationship between the UID and the application. The applications couldn't be run on the Android without signing. With the same UID, that is, running in the same sandbox, the applications can share the permissions and communicate with each other.

By using application signing, the application update process can be simplified. Since different versions of the same application have the same certificate, the package manager can verify this certificate. Then, the old version is replaced, the new version can have the permissions already granted to the old version. What's more, the application signing can also ensure that an application cannot communicate with another unless using the ICC. But if the author is the same, the author can use the same application signing to enable the direct communication among his/her applications.

2.4.3 Inter Component Communication

Android platform provides a secure ICC that is similar to IPC to the Unix system [34]. ICC is provided by the binder mechanism which is in the middleware layer of

Android. The binder is a remote procedure call that is a custom Linux driver. ICC is achieved by intents. An intent is a message that shows the target with some data optionally. It can be used in explicit communication if it identifies the name of the receiver, or used in the implicit communication that let the receiver see if it can access this intent or not.

2.4.4 Permission Model

An application is isolated when it's executed in the sandbox. When the applications want to access some sensitive features, such as camera, location, telephony, network. Android provides a permission model to achieve this goal.

Permissions mechanism is used to make some restrictions when the applications want to access the sensitive APIs of the operating system.

As discussed in Section 2.3, an application developer can declare all the permissions required in the Androidmanifest.xml file. Before the application is installed on the device, the system will ask the users if they grant the permissions to this application. If the users agree to grant all requested permissions to the application, the installation continues, otherwise, the installation cancels. Unlike iOS, the user cannot choose which permissions they want to grant and which permissions they want to deny. Moreover, the application can get the permissions through the application signing.

The permissions have four levels, normal permissions, dangerous permissions, signature permissions and signature/system permissions. Normal permissions can be granted automatically; dangerous permissions are inferred to those granted by the users; signature permissions are granted within the same sandbox; signature or system permissions are granted to pre-installed applications or the applications installed by the root.

Chapter 3

Android App Analysis Frameworks

Since 2010, there has been a steep increase in research about the Android application behavior analysis. This could be explained by the fact that the Android is popular, open-source, which eases analysis and modification of the OS, and that millions of applications are available for analysis. A number of existing static and dynamic analysis proposals focus on analyzing Android applications for privacy leaks. In this chapter, we organize 32 of the most recent and promising privacy-oriented proposals on Android app analysis into two categories: static and dynamic analysis.

3.1 Characteristics

For our survey, we group common characteristics of the target frameworks and tools into a set of sub-categories: Type of framework, methodology, and deployment. We use these attributes to compare the frameworks in Tables 1 and 2.

3.1.1 Type of Framework

The type of a framework describes the goal of the research done. We further categorize the following different types of proposals:

1. **Analysis.** Frameworks that helps analyzing Android applications for privacy-sensitive leaks, application internal mechanism (e.g., IntelliDroid and TaintDroid both belong to this category).

2. **Detection.** Frameworks that help in detecting and reporting privacy leaks or malicious behavior in Android applications (e.g., TaintDroid).

3.1.2 Methodology

Since the rise of these static and dynamic analysis frameworks and tools, different approaches are used to analyze and detect privacy leaks in the applications. In our view, the most common approaches used by existing proposals are explained below:

1. **Dynamic.** In this category, we list those frameworks that use dynamic analysis techniques to analyze Android applications. The proposals in this category record the application behavior during the runtime.

2. **System Calls.** Frameworks that capture Android system calls made by an application.

3. **Method Tracing.** Frameworks (dynamic analysis) that keep track of specific method invocations.

4. **Static.** Frameworks that use static analysis techniques on target applications (e.g., IntelliDroid uses static analysis to generate API inputs for the target application).

5. **Decompilation.** Frameworks that apply decompilation techniques on the target APKs.

3.1.3 Deployment

In this category, we discuss the various deployment methods used by the researchers for the implementation of the frameworks:

1. **Android apps** Frameworks that analyze regular Android applications.
2. **APK** Frameworks that provides an Android application (APK) for deployment.
3. **OS modification** Frameworks that are deployed on Android OS. In this case, the Android OS needs to be patched with framework files. For example, TaintDroid and IntelliDroid require modification of Android OS.
4. **WebApp/Script** Proposals that are available in the form of the web application or scripts/package.

3.2 Taxonomy of Approaches for Analyzing Apps

In this section, we characterize the state-of-the-art reverse engineering tools and frameworks listed in Tables 1 and 2. These tools and frameworks have reported the pervasiveness of privacy disclosures in Android apps. We categorize our current analysis approaches into two types: static and dynamic analysis.

3.2.1 Static Analysis

This class describes tools and frameworks that are used to investigate Android applications using static analysis. In static analysis frameworks, the analysis is conducted by disassembling the APK packages and analyzing the decompiled code. The control flow of an app is determined by events and callbacks for system-event handling. This section also includes information on decompilers and disassemblers that are used by

popular static analysis tools. Following are the more details on popular static analysis tools and frameworks listed in Table 1:

1. **Tools.** The packaging model of Android apps requires the entire code to be shipped into a one single APK file. The APK is then decompiled for processing and analysis.

   Following are the common reverse engineering tools that help research analysts for static analysis of applications by implementing a Dalvik bytecode decompiler or disassembler.

   (a) **APKinspector** [53] is a GUI static analysis tool to analyze the Android applications. The goal of this framework is to help analysts and reverse engineers to visualize compiled Android packages and their corresponding *dex* code. APKinspector provides both analysis functions and graphic features for the users to gain deep insight into the malicious apps. In particular, it provides detailed CFGs, call graphs, static instrumentation and APK information for analysis.

   (b) **Apktool** [61] is a very popular tool for reverse engineering closed binary Android apps. It decodes resources to nearly the original form and rebuilds them after making some modifications.

   (c) **DeDexer** [42] disassembles the *.dex* class files into *Jasmin-like* syntax and creates an individual file for every class conserving the package directory structure for easy reading and manipulation.

   (d) **Dare** [12] framework re-targets Android applications in *.dex* or *.APK* format to the traditional *.class* files. Dare cannot reassemble the disassembled intermediate class files like *apktool*.

Tool/ Framework	**Analysis**	**Detection**	**Taint tracking**	**Decompilation**	**WebApp/Script**	**Availability**
IntelliDroid 2016 [63]	✓	✓		✓	✓	✓
DroidSafe 2015 [26]	✓	✓	✓	✓	✓	✓
Dex2Jar 2015 [43]	✓			✓	✓	✓
FlowDroid 2014 [5]	✓	✓	✓	✓	✓	✓
DroidAPIMiner 2013 [3]	✓		✓			
WHYPER 2013 [44]		✓				
DroidRanger 2012 [69]		✓			✓	
RiskRanker 2012 [27]		✓			✓	
Dare 2012 [12]	✓			✓	✓	✓
Androguard 2012 [14]	✓	✓		✓	✓	✓
APKinspector 2012 [53]	✓			✓	✓	✓
DroidChecker 2011 [10]		✓	✓		✓	
apktool 2010 [61]	✓			✓	✓	✓
DeDexer 2009 [42]	✓			✓	✓	✓

Table 1: Comparison between state-of-the-art static analysis tools and frameworks.

(e) **Dex2Jar** [43] is a tool for converting Android's *dex* formatted files to Java byte-code. Given an Android APK, the tool can convert it directly into a .jar file and vice versa.

2. **IntelliDroid**

Category: Analysis and detection

Type: Static and targeted API analysis (output can be used for a dynamic analysis tool that monitors the execution of an Android application)

IntelliDroid [63] is a static analysis framework for automatically generating inputs for targeted applications, specific to a dynamic analysis tool.

The inputs generated by IntelliDroid can be effectively used to trigger the malicious behavior in the target applications. For generating inputs, IntelliDroid can be provided with a list of sensitive targeted APIs and it will automatically detect the occurrences of targeted APIs in the application and will generate inputs to trigger them. IntelliDroid's static component is also capable of determining the order in which input has to be provided.

One of the challenging issue in detecting privacy leaks is that not all the applications reveal their malicious behavior when they are installed or even run on the device. Instead, the malicious behavior can be triggered based on different conditions. For instance, a group of malware can stay hidden until an API is triggered by an event (activating its malicious functionality). Some events are independent of user interactions with the application (i.e. network connection), yet some others are based on user input. However, testing mobile application is not a simple task due to a variety of inputs and heterogeneity of the technologies.

Solving this problem, IntelliDroid obtains information of each app by decompiling the APKs into *classes* and a*Androidmanifest.xml* file. It further scans

the entire decompiled classes and determines trigger paths for all the functions in the APK. IntelliDroid is more promising than popular fuzzing tool Android Monkey [25] in terms of input generation and static analysis as it uses an entry-point discovery mechanism to identify paths for targeted APIs.

Also, it is known that Android applications use activity driven graphical user interface heavily. Therefore, simply running the application for some time may leave many application's functionalities un-executed and is difficult to figure out if the application is a threat to the user or the device. There are different execution paths in an application and only a small number is covered by merely starting or running the application. Since dynamic analysis checks the executing code's behavior, to provide better, if not full, coverage the static analysis tool can provide user inputs so that more paths can be covered.

As a comprehensive static analysis solution, IntelliDroid develops a call graph (partial) from the discovered entry points. This call graph is further used to look for all the calls to functions and constructors of Android callback listeners and to add listener methods to the list of entry points. The framework uses information about dynamic analysis in conjunction with static analysis of the APK file provided. It generates a set of paths that are specific to dynamic analysis framework and triggers malicious behavior to be detected.

For our evaluation methodology, we adopt the original model of IntelliDroid's design and working. Therefore, our evaluation methodology and IntelliDroid's original methodology is the same. We explain the methodology and the design of IntelliDroid (including the changes we made in the original framework) in Chapter 5.

3. **DroidSafe**

Category: Analysis and detection

Type: Information flow static analysis, Taint analysis

DroidSafe [26] is a static application analysis tool designed to analyze malicious information flows in Android source code and APK files. DroidSafe tracks information flows from sources (Android API calls that inject sensitive information) to sinks (Android API calls that may leak information) for the target Android applications. It uses a combination of static and dynamic analysis to report the flow of information (tainted or not). This combination is enabled by accurate analysis *stubs*, a technique that enables the effective analysis of code.

4. **FlowDroid**

   Category: Analysis and detection

   Type: Information flow static analysis, Taint analysis

   FlowDroid [5] provides a highly precise static taint analysis that is fully object-sensitive, flow-sensitive, and context-sensitive. FlowDroid effectively analyzes and determine connections from source to sink for a targeted application. The focus of FlowDroid is to detect information leakage in the Android applications. FlowDroid covers various types of taint analyses, which include context, flow, field, object-sensitive, and lifecycle aware taint analysis. The code source of FlowDroid is publicly available.

5. **WHYPER**

   Category: Analysis

   Type: Application behavior analysis, Risk assessment

   WHYPER [44] framework uses Natural Language Processing (NLP) techniques to determine at he need for a given permission in an application's description.

The goal of WHYPER is to examine permissions for an app, whether the app description provides any indication for the requirement of the permission.

6. **DroidAPIMiner**

   Category: Analysis

   Type: Malware analysis

   DroidAPIMiner [3] uses a robust and efficient approach for describing Android malware that relies on the API, package, and parameter level information. It follows a generic data mining approach that aims to build a classifier for Android apps. To predict if an app is benign or malicious, the classifiers rely on the semantic information within the bytecode of the applications ranging from critical API calls, package level information and some dangerous parameters invoked.

7. **DroidRanger**

   Category: Detection

   Type: Static analysis using heuristics-based filtering scheme

   DroidRanger [69] recognizes suspicious behaviors from all possible malicious applications and detects the Android features that may be misused. The framework extracts fundamental properties associated with each app (e.g., the requested permissions and author information) and organizes them along with the app itself in a central database for efficient indexing and lookup. Additionally, DroidRanger also uses heuristics-based detection engine to uncover malware that has not been reported before.

8. **RiskRanker**

   Category: Analysis

Type: Application behavior analysis, Risk assessment

RiskRanker [27] performs signature-based analysis for the detection of known exploits on Android applications. It statically classifies applications into multiple security risk categories. Then, it prioritizes potential risks from the target untrusted apps and narrow downs the search space to a manageable size. RiskRanker aims at detection, permission analysis and data-flow analysis in Android malware. Applications using a combination of dynamic code loading and native code execution are labeled as high-risk apps by RiskRanker.

9. **Androguard**

   Category: Analysis and detection

   Type: Reverse engineering, Malware analysis

   Androguard [14] is a popular static analysis tool based on python and can run on Linux/Windows/OSX, provided python is installed in the system. It can effectively disassemble and decompile Dalvik Bytecode back to Java source code. Given two APK files, it can also compute a similarity value to detect repackaged apps or known malware. It also has modules that can parse and fetch information from the app's XML files. Due to its flexibility, it is used by some other (dynamic) analysis frameworks that need to perform some form of static analysis. The important features of Androguard are: finding app code similarities, a risk indicator for apps, and managing a database of signatures for malicious apps.

10. **DroidChecker**

    Category: Analysis and detection

    Type: Taint tracking, Control flow tracking

DroidChecker [10] is an automated framework to detect capability leaks in Android applications. It targets applications that use at least one permission while containing unprotected components that are publicly visible. DroidChecker uses inter-procedural control flow graph analysis (CFG) and static taint checking to detect exploitable data paths in an Android application. It also aims at discovering privilege escalation attacks and only analyzes exported interfaces and APIs from the applications that are classified as dangerous.

3.2.2 Dynamic Analysis

Dynamic analysis frameworks rely on the run-time behavior of the apps to classify them as malicious. Under the umbrella of this concept, privacy leakage detection is done by following a runtime data flow tracing model: finding feasible traces from predefined source APIs (the ones that read and leak private data) to sink APIs (the ones that send private data out of the device). Dynamic analysis is performed while the app is being executed on real devices or emulated environments. Following is the summary of dynamic analysis frameworks listed in Table 2.

1. **CopperDroid**

   Category: Analysis and Detection

   Type: Behavioral analysis

   CopperDroid [57] is a dynamic analysis framework that captures OS level event sequences and high-level Android specific behaviors in Android applications. CopperDroid leverages QEMU (an open source machine emulator and virtualizer) to automatically conduct black-box dynamic analysis on Android applications. The VM-based system makes use of dynamic system call analysis in order to determine Android behaviors. CopperDroid also has the ability to determine

Framework	Analysis	Detection	SystemCalls	Method tracing	Taint tracking	Static	Android apps	OS modification	WebApp/Script	Availability
CopperDroid 2015 [57]	✓	✓	✓			✓			✓	✓
TaintDroid 2014, 2010 [19]	✓	✓			✓			✓		✓
Andrubis 2014 [37]	✓	✓		✓	✓	✓	✓	✓	✓	✓
Mobile-SandBox 2013 [54]	✓	✓				✓				
SandDroid 2013 [58]	✓	✓		✓	✓	✓		✓	✓	✓
AppsPlayGround 2013 [46]	✓		✓	✓	✓	✓		✓	✓	✓
DroidScope 2012 [66]	✓		✓	✓	✓	✓			✓	✓
SmartDroid 2012 [67]	✓		✓	✓	✓	✓		✓	✓	
Google Bouncer 2012 [31]	✓	✓								
Aurasium 2012 [64]		✓	✓							✓
I-ARM-Droid 2012 [13]		✓							✓	
QUIRE 2012 [17]		✓						✓		
Droidbox 2011 [15]	✓			✓	✓	✓		✓	✓	✓
Andromaly 2011 [51]		✓					✓			✓
AppFence 2011 [30]		✓						✓		✓
CrowDroid 2011 [9]		✓					✓			
XManDroid 2011 [8]		✓						✓		
AASandBox 2010 [7]	✓	✓	✓			✓		✓	✓	

Table 2: Comparison between state-of-the-art dynamic analysis frameworks.

whether a malware was initiated using Java, JNI, or native code.

2. **TaintDroid**

   Category: Analysis

   Type: Information-flow tracking, dynamic taint tracking, and analysis

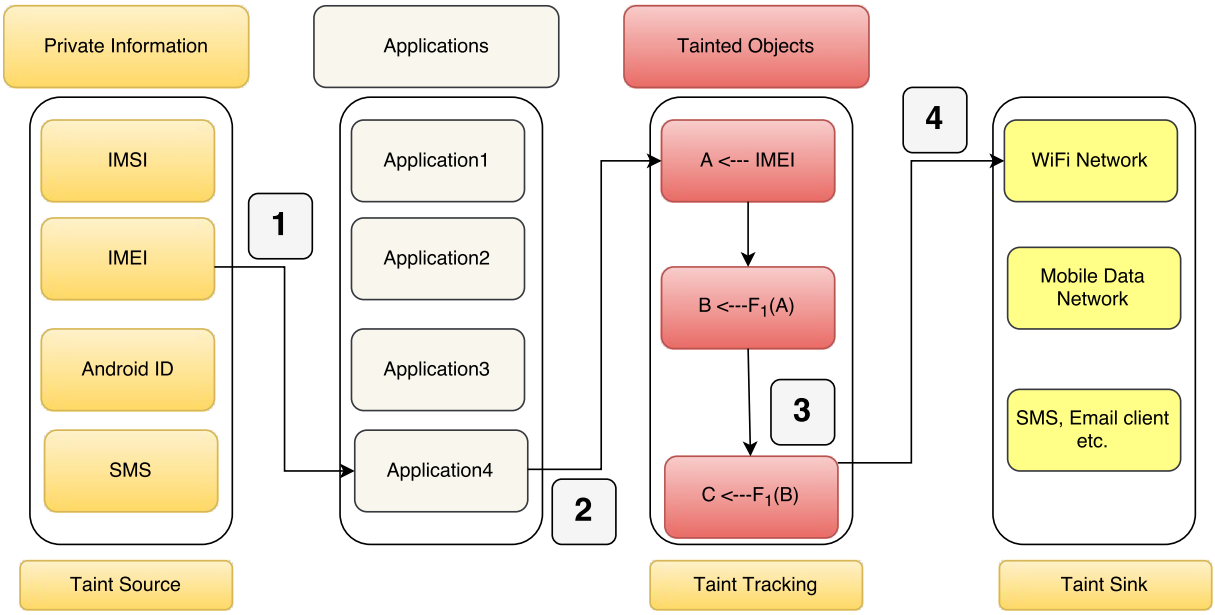

Figure 2: The simplified architecture of TaintDroid taint analysis.

TaintDroid [19] is one of the first dynamic analysis engines introduced for Android apps. It performs taint tracking to precisely analyze how private data is obtained and released at runtime. In achieving this, it adopts an efficient way to handle taint storage. It also defines taint propagation rules on Dalvik instructions across API calls. As TaintDroid handles taint analysis of Dalvik instructions across API calls at runtime, it is resistant to Java reflection and code encryption. In addition, TaintDroid can be loaded into real devices, allowing for real-time monitoring of actual hardware and sensors. These advantages have made TaintDroid be used widely in Android app behavior analysis.

Figure 2 gives a more clear explanation of TainDroid taint analysis. The steps followed in Figure 2 are explained below:

(a) Application4 accesses private data (IMEI).

(b) IMEI is saved in variable 'A'.

(c) Variable 'A' evolves and propagates through variable 'B' and variable 'C'.

(d) Application4 attempts to leak the value stored in variable 'C' to the attacker and a taint log notification is generated.

As TaintDroid is implemented as an extension to the Dalvik virtual machine, it can oversee all the activities which run above it (recall Figure 1). The framework uses the concepts of taint sources, from which sensitive information (e.g., IMEI, address book, location data or private device identifiers) is obtained, and taint sinks, which are interfaces to the outside world (e.g., using data networks or sending SMSs) where tainted information is usually not expected to be sent. When tainted data reaches a taint sink, TaintDroid issues a warning to the user.

TaintDroid is an easy to use with a static analysis tool and can be easily integrated with any supporting static analysis framework. For instance, IntelliDroid using TaintDroid can trigger and detect sensitive- privacy leaks. In original work, authors show that IntelliDroid's event chain detection and device-framework interface input injection enabled it to effectively generate inputs that trigger targeted APIs in a corpus of malware.

3. **Andrubis**

Category: Analysis and Detection

Type: Static and dynamic analysis

Andrubis [37] performs static as well as dynamic analyses and allows uploading

APK files using a web interface.[1] Andrubis [37] leverages various techniques like VMI and monitors events in the Dalvik VM, as well as native code for system calls through QEMU VMI. Andrubis has an extensive feature set, including support for the native code loading. A major limitation of Andrubis is, it still runs on Android version 2.3.4 which is too outdated for current Android applications. Andrubis leverages TaintDroid to track sensitive information across application borders in the Android system.

4. **Mobile-SandBox**

   Category: Analysis

   Type: Static and dynamic analysis

   Mobile-Sandbox [54] is a static and dynamic analyzer for Android applications with the purpose to support malware analysts to detect malicious behavior. In the static analysis, it parses the application's *AndroidManifest.xml* file and decompiles the application. It also performs dynamic analysis on target application in order to log all performed actions including those stemming from native API calls. Mobile-Sandbox also allows regular users to submit apps for analysis via a web application. It uses a combination of static and dynamic analysis techniques.

5. **SandDroid**

   Category: Analysis and detection

   Type: Static and dynamic analysis

   SandDroid [58] is an online Android app analysis tool. SandDroid performs detailed static and dynamic analysis on the target applications and presents the results graphically to the user. It leverages Androguard [14] and DroidBox

[1] `http://anubis.iseclab.org`

[15] to track sensitive information across application borders in the Android system. Besides its comprehensive analysis approach, it also provides security rating to the applications based on the results of the analysis.

6. **AppsPlayGround**

   Category: Analysis

   Type: Dynamic analysis

   AppsPlayground [46] is a framework that automates the analysis of Android applications and monitors taint propagation (using TaintDroid), specific API calls and system calls. Kernel level monitoring is also implemented in the framework to defend against known exploits. AppsPlayground focuses on detecting evasion and using automated exploration techniques for increased coverage of the code of the application and therefore to trigger the malicious behavior of an application. Its main contribution is a heuristic-based smart black-box execution approach to explore the app's GUI. The testing approach used in AppsPlayGround is similar to Android monkey exerciser to explore possible application's GUI points.

7. **DroidScope**

   Category: Analysis

   Type: Static and dynamic analysis

   DroidScope [66] is an analysis tool that utilizes a VM-based system in order to detect malware. Droidscope uses Virtual Machine Introspection to reconstruct Dalvik and native code instruction traces. Specifically, DroidScope reconstructs both the kernel and system-level semantics in order to facilitate malware analysis. Furthermore, DroidScope utilizes three tiers of APIs to emulate an Android device. These three tiers include the hardware, Android OS, and Dalvik VM.

Results indicate that DroidScope was effective in assessing malware samples with low overhead. One of the major limitations of Droidscope is it is bound to emulator-use only.

We have already discussed in Chapter 1 that malware applications can detect the use of an emulator and may decide not to start malicious activities in this scenario [16].

8. **SmartDroid**

   Category: Analysis and detection

   Type: Static and dynamic analysis

   The SmartDroid[2] framework statically extracts the function call graph and activity call graph of Android applications. It then dynamically traverses these graphs to find elements that trigger conditions to expose an app's sensitive behavior.

9. **Google Bouncer**

   Category: Analysis and detection

   Type: Possibly dynamic (no official confirmation)

   To detect the malicious behaviors of apps, Google introduced Google Bouncer [31], a malware scanning service to detect malicious in-store apps. Google Bouncer performs a set of analyses on new applications, applications already in GooglePlay, and developer accounts for malicious behavior. It also analyzes new developer accounts to help prevent malicious and repeat-offending developers from coming back. As Google Bouncer is a proprietary project, therefore, no information about its methodology is available.

[2] `http://sanddroid.xjtu.edu.cn`

10. **Aurasium**

    Category: Detection

    Type: Repackaging APKs, Privacy policy enforcement

    Aurasium [64] automatically repackages arbitrary applications and closely watches the behavior of security and privacy intrusions such as attempts to retrieve a user's sensitive information by target applications. To attach sandboxing code, Aurasium exploits Android's unique application architecture of mixed Java and native code execution and introduces libc interposition code. Because of this, Aurasium is capable of mediating almost all types of interactions between the application and the Android OS. Also, Aurasium bypasses the need to root the device when modification of the Android OS is required. The behavior of the application can be modified or the flow of the information can be followed. Additionally, Aurasium has the ability to detect and prevent cases of privilege escalation attacks.

11. **I-ARM-Droid**

    Category: Detection

    Type: Reference monitor, Rewriting framework

    I-ARM-Droid is an inline reference monitor-based approach to enforce security policies in Android OS. In I-ARM-Droid, the framework user first identifies a set of security sensitive API methods and then specifies proposed security policies, which can be further tailored to a set of target applications. Then the framework automatically rewrites the Dalvik bytecode in the application, where it interposes on all the invocations of these API methods to implement the desired security policies.

The approach used by I-ARM-Droid does not allow the instrumentation of applications on the phone so far, however, it supports to instrument calls to any Java method and covers reflective Java calls.

12. **QUIRE**

    Category: Detection

    Type: Tracking call chain

    QUIRE [17] is a framework proposed to detect and protect 'confused deputy attacks' which are a kind of privilege escalation attack exploiting the vulnerability of the security model of Android. It is available in the form of a group of extensions to the Android operating system that enables applications to propagate call chain context to downstream callers and to authenticate the origin of data that they receive indirectly. The set of Android extensions provided by QUIRE allows apps to defend themselves against confused deputy attacks on their public interfaces and enable mutually untrusting apps to verify the authenticity of incoming requests with the Android. QUIRE is a backward compatible OS extension to the Android operating system that allows existing Android applications to co-exist with applications that make use of QUIREs services.

13. **Droidbox**

    Category: Taint Analysis

    Type: Dynamic analysis

    DroidBox [15] uses TaintDroid to detect privacy leaks by modifying Android's core libraries. It also comes with a Dalvik VM patch to monitor the Android APIs and report file system and network activity, the use of cryptographic operations and cell phone usage such as sending SMS and making phone calls.

In the newer version release of DroidBox, it utilizes bytecode rewriting instead of modifying the core Android libraries.

DroidBox is openly available and has been used as a base system by in several other dynamic analysis platforms including Andrubis [37], Mobile-Sandbox [54], and SandDroid [58].

14. **Andromaly**

    Category: Detection

    Type: Malware detection

    Andromaly [51] is a behavior-based detection framework for Android based devices. It is a host-based intrusion detection system and can continuously monitor various resources and classify malicious applications. It exploits machine learning algorithms to detect intrusion on Android OS. Andromaly considers the occurrences of higher level events and use them to detect intrusion but has been tested only on proof-of-concept malware.

15. **AppFence**

    Category: Detection

    Type: Malware detection

    AppFence [30] is a lightweight extension of the Android to enforce information flow policies at runtime. AppFence keeps track of the propagation of private information by leveraging TaintDroid. When the privacy leakage is detected, AppFence either blocks the leakage at the sink or shuffle the information from the source (e.g., by using fake contact information). Keeping usability into consideration, the authors proposed multiple sets of access control rules and also conducted empirical studies to gather the best policies in practice for deployment with AppFence.

16. **CrowDroid Category**: Detection

    Type: Malware detection

    Crowdroid [9] is a lightweight client application that monitors system calls invoked by a target mobile application, preprocesses the calls, and sends them to cloud where a clustering technique helps determine whether the application is malicious. CrowDroid uses *strace*, a debugging utility for Linux and some other Unix-like systems, to monitor every system call and the signals it receives. It identifies malicious behavior and detects malware utilizing popular *K-means* algorithm on the server side.

17. **XmanDroid**

    Category: Detection

    Type: Heuristic based analysis, Detection of covert channel attacks

    XmanDroid [8] is a device-centric and policy-driven runtime monitoring system that regulates communications between different applications on Android platform. It allows real-time detecting and blocking privilege escalation attacks by using ICC. XManDroid monitors the communications of the applications, then compares them with a set of pre-defined security policies. It also claims to stop collusion attacks that communicate via channels other than Android's standard IPC mechanism. Evaluation results show that the performance overhead imposed by XmanDroid is below human perception and have very little performance effect on overal working of Android.

18. **AASandbox**

    Category: Analysis and Detection

    Type: Static analysis, dynamic analysis, and API monitoring

AASandbox [7] was the first system (presented in October 2010) combining static and dynamic analysis for the Android platform. It implements a system call monitoring approach using a loadable kernel module. Furthermore, it uses the resulting system call footprint to discover possibly malicious applications. The framework allows identification of software reliability flaws and to trigger malware without requiring source-code. The framework makes use of dynamic approach by collecting run-time behavior analysis and also the I/O system calls generated by the applications. Unfortunately, AASandbox is not maintained anymore.

3.3 Discussion

The goal of this survey was to introduce the existing static and dynamic analysis proposals for analyzing Android apps. We characterized several recent proposals into different categories (e.g., analysis, risk assessment, detection, static analysis tools, etcetera). Considering the way that author's implemented the proposed solution, Table 1 and 2 shows the comparison results for all the covered works in this survey. We have discussed very high-level details specific to each proposal in the previous section. However, we discuss a few limitations common limitation(s) of the both categories as follows.

Static analysis. Most static analysis frameworks cannot handle obfuscation techniques such as code encryption and dynamic code-loading [68, 47]. More critically, while Android apps are generally developed in Java, compiled to Java bytecode, and only then converted to Dalvik bytecode. Some frameworks perform analysis only on DEX bytecode, limiting its effectiveness in practice.

For tools like Android Monkey and other automated UI exercisers, a major problem is that it is difficult to guarantee that all malicious behaviors can be triggered

during testing. Just measuring the code/path coverage of the app might not be sufficient as malicious apps can easily hide malicious behaviors deep inside the program. One option to solve UI exercisers problems is to use a static analysis tool that accesses the application sensitive APIs problematically.

Dynamic analysis. The way of implementation for all the surveyed proposals in both categories is different.

For dynamic analysis, the frameworks can be implemented on the application level or framework and Linux kernel level. As shown in Table 2, most proposals modify the framework and Android OS. The need to modify the Android OS comes from this fact that dynamic solutions are based on app activities. Therefore, the only way to monitor the applications activities such as system calls is modifying the kernel or the framework. The main problem with rooting the device and making changes in the Android OS for an implementation is that in fact, they may make the device unsafe and vulnerable to attacks. Also, many of the dynamic analysis proposals for identifying privacy-sensitive behavior suffer from false positives, which can effect the detection results.

Another thing to consider is, a majority of dynamic analysis proposals do not assume the use of native code in Android apps. Native code must also be considered in developing new detection and analysis algorithms due to its wide presence in real-world apps. In our survey, Mobile-Sandbox [54] is the only proposal that is able to track native code API calls.

Chapter 4

Threat Model

This dissertation considers a threat model in which the frameworks we deploy in our toolchain can detect applications aiming to gain and abuse sensitive resources of Android stealthily. More specifically, this work assumes an application can abuse Android resources and permissions to access sensitive data. In general, we consider a privacy leak to be any transfer of user's personal data (e.g., contacts, location) or any information that helps to identify the device uniquely.

1. It is challenging to draw a clear line between the malicious activity and a normal operation of mobile applications. A large number of malware samples harvest the private data stored in the mobile device and send the data to remote servers with malicious intent. This form of information leak attack could have more impact on users when the financial credential is targeted. However, to make our goals more evident, this dissertation defines the information as private data leaks without user's consent. Our work does not distinguish the difference between personal data being used by an app for user-expected application functionality; nor do we attempt to differentiate between safe and malicious leaks. Another key point is, it is very difficult to learn the intentions of the developer during

the development of the application.

2. Our work focuses on Android applications leaking private sensitive information within the scope of the Android security model for Android 4.3 or lower. We are not concerned with vulnerabilities or bugs in Android OS code, the SDK, or the Dalvik VM which runs applications. The trusted computing base for the frameworks we leverage for our analysis is the Android framework, Linux kernel, and the Dalvik virtual machine.

3. Some applications that work on Dalvik does not work on ART. While we were analyzing the apps, we found that many applications were crashing (the ones we downloaded from Google Play Store). The applications were crashing because the app developers always tend to target newer Android versions to use latest features. Unfortunately, the newer features are written for applications according to the ART system, which TaintDroid framework does not support. Therefore, TaintDroid is not able to analyze many new applications. We exclude such apps from our analysis.

4. Android apps can load code at runtime to dynamically extend their functionality. However, this technique comes with severe security implications. While dynamic code loading is popular for legitimate reasons, such as loading external add-on code, shared library code from frameworks, or dynamically updating code during beta and/or A/B testing, it is especially interesting for malware. In our toolchain, apps are analyzed only once by IntelliDroid, malicious apps can download and load their malicious payload later at runtime to evade detection in dynamic analysis. As the dynamically loaded code is downloaded at runtime, our work does not aim to detect inputs for the dynamically loaded code.

Chapter 5

Methodology

In this chapter, first, we discuss various building blocks of the information flow tracking framework used in our analysis. Later, we discuss types of sensitive information that can be detected by the TaintDroid framework. In the end, we discuss the methodology of our cross-validation approach.

We demonstrate toolchain work flow in Figure 3 for a single application. As depicted in this figure, our toolchain contains two major phases: static analysis and dynamic analysis. In the first phase, we leverage IntelliDroid for pre-processing of the APK files. In the second phase, we rely on the TaintDroid framework for dynamic analysis.

To conduct our experiment on regular Android applications, the runtime information inside the app should be fully monitored while it is being executed. Hence, the crucial function in our approach is to track and record the specific privacy sensitive activities that may take place in the application. In order to validate our setup and scripts, we repeated the analysis done by original authors of IntelliDroid on malware samples provided by Android Genome Project [65] and got similar results as the original study. Our detailed methodology for the analysis is given below:

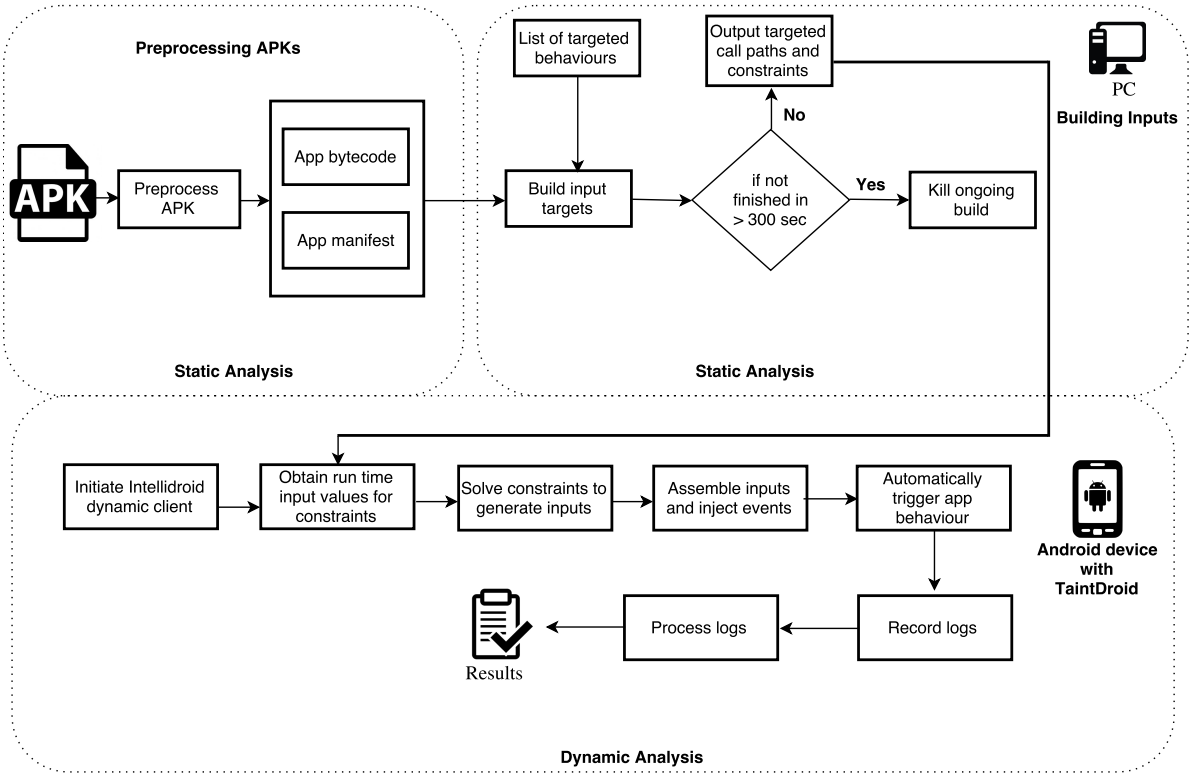

Figure 3: The simplified architecture of our IntelliDroid and TaintDroid toolchain analysis workflow for a single APK.

5.1 Static Analysis

In this phase, we adapt IntelliDroid's original methodology to process and analyze the APK files. To scale-up the process, we introduce few modifications and scripts in the work-flow of the toolchain. More details of the steps followed in this phase are described in the following sections:

5.1.1 Preparing the Dataset of Target APKs

Our dataset includes a variety of applications from different categories and sources. We discuss detailed information about our dataset in Section 6. We follow the steps below to install a batch of APKs on our Android device

1. We categorize all the applications in our dataset and sorted them into their

respective categories (e.g., Entertainment, Social, Business, Transportation).

2. We write a script to create flashable zip packages for all the sorted APKs. On average, each batch of zip package had nearly 100 APK files from each category.

3. Then, we upload the *TWRP recovery image* [2] on our Android device. TWRP is an open-source software custom recovery image for Android-based devices. TWRP allows the installation of custom ROMs, kernels, and add-ons (e.g. Gapps, custom zip packages).

4. Finally, we flash the target APK zip packages on our Android device using TWRP recovery options.

5.1.2 Pre-processing APKs

We process our target APKs using pre-processing scripts provided by the IntelliDroid framework. In this phase, IntelliDroid derives the path constraints for each call path made in application code. In this stage, the applications are unpacked and converted to Java bytecode using decompilation tools. The preprocessing stage makes use of *APKParser* [18] and *Dare* [12] frameworks to decompile and converted apps to Java bytecode.

5.1.3 Building Input Constraints

After the APKs files are translated to java bytecode, the generated files are passed to IntelliDroid's static component. The static component makes use of WALA [59] static analysis libraries. In general, WALA is used for static analysis, call graph construction and inter-procedural data-flow analysis of the provided code.

To perform the static analysis on the APK code, *IntelliDroidAppAnalysis* component identifies invocations of targeted API and find target call paths from the handlers

that lead to the targeted API in the application. A list of targeted APIs is passed to *IntelliDroidAppAnalysis* component for processing. For example, in our analysis, we use *taintdroidtargets.txt* file provided by the IntelliDroid's original work for building the targeted API inputs. This list contains multiple Source and Sink APIs that are tagged has privacy sensitive by the original authors (More information on our Target APIs is given in Section 5.4).

For generating the input constraints, IntelliDroid develops a call graph to get an accurate representation of execution flows between different methods in the application. The call graphs are mainly used to search for the targeted APIs.

Considering time taken for generating input constraints, we observe that a few constraint building processes were taking significant amount of time and system memory to generate the inputs. As a result, these processes occupied all the system resources leading to the system crash. As per our observation, a simple application takes around 5 minutes to generate the inputs (average calculated for 100 apps). We had written a script to automatically build targets for a set of APKs using the commands provided by the IntelliDroid framework to execute in a specified period of time (i.e., 300 seconds). If the time limit is not satisfied, our script will kill the ongoing process and will initiate the next build in the list. Once the path generation is completed, in the final step, these paths along with information for dynamic analysis are then injected to the core *device framework* of Android for further analysis.

5.2 Dynamic Analysis

In this phase, we initiate IntelliDroid's dynamic client script from our host machine. The dynamic client host program is responsible for connecting to the Android device client program by using Android Debug Bridge (*adb*) [23]. The dynamic client helps in injecting and triggering the input paths into Android's core device framework.

The core of dynamic client for IntelliDroid uses the z3 constraint solver [38] to trigger path inputs. As shown in Figure 3, the Android device runs a system service called *IntelliDroidService*, which further connects to the host machine to receive inputs from the IntelliDroid client.

5.2.1 Trigger Paths

In this phase, input triggers with sensitive APIs are sent to the Android device. When the trigger instructions are being sent to the device, the input execution activates the trigger-based behavior, which enables us to observe the trigger-based behavior in a controlled environment. We modified IntelliDroid's dynamic client script to automatically trigger n inputs generated in the static analysis phase for each application.

5.3 Log Collection

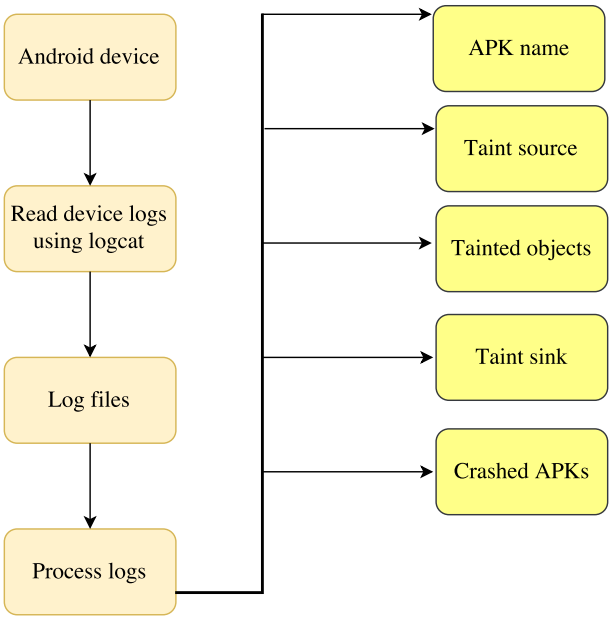

Figure 4: Log collection process subsystem.

Android helps in providing a mechanism to collect and view the system debug

output. The logs are collected in a circular fashion serially from various mobile applications and different portions of the systems as well. During the dynamic analysis phase, we use the *logcat* command to capture and filter the logs that are collected from the various applications.

The process of identifying the logs specific to TaintDroid is very straight forward as TaintDroid automatically labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and IPC messages. When tainted data are transmitted over the network, or otherwise leave the system, TaintDroid logs the data's labels, the application responsible for transmitting the data, and the data's destination.

Figure 4 shows our log collection subsystem. At first, when the APK file is executed in the device, logcat actively collects the logs of applications. Later, our script processes the logs with the details like APK package name, timestamps, crashed applications, and tainted data information.

5.4 Taint Sources and Sinks

In this section, we discuss in detail how we identify sensitive functionalities that could leak user's data. Table 3 enumerates the resources that TaintDroid's authors consider as privacy sensitive. As per original work, if there is a flow of privacy sensitive data from an information source through a sink, such a flow is referred as privacy leak. Following is the detailed explanation of sensitive information sources listed in Table 3.

1. **Sources.** API calls that return private device information are considered as information sources (e.g., location, IMEI, address book, etc). Our setup tracks 7 types of private information tagged as privacy sensitive by TaintDroid;

Taint	Category	Function
IMEI	User Identification	Obtain IMEI of the device
AppID	Application Identification	Reference to the Application Identifier
Android ID	User Identification	Obtain 16 digits Android ID
UDID	User Identification	Obtain/Create unique device identifier
Location	Accessing Resources	Obtain Location of device
Contacts	Accessing Resources	Accessing AddressBook of device
System Status	Accessing Resources	Accessing state of running applications
SMS	Accessing Resources	Reading SMS from device

Table 3: Target sensitive information sources and their functionality.

(a) **Location Information.** Location information represents the current location of the Android user. Malicious apps can track device location with automated location tracking, geo-fencing, and activity recognition [60]. With location data, an app can predict a user's potential actions, recommend actions, or perform actions in the background without user interaction. Before an application can receive any location data, the developer must request location permissions. There are two location permissions that can be used to access location, *ACCESS_FINE_LOCATION* and *ACCESS_COARSE_LOCATION*.

(b) **Phone Identifiers.** Applications can send phone identifiers to remote network servers. TaintDroid tracks 6 different phone identifiers: phone number, IMEI (device ID), IMSI (subscriber ID), ICC-ID (SIM card serial number), UDID (unique device identifier), Android ID(16 digit device identifier), and AppID (Application package identifier):

i. **IMEI.** Every smartphone device has a unique 15 digit code, called an IMEI (International Mobile Equipment Identity) number. IMEI of a user device can be accessed by applications without requiring explicit user authorization.

ii. **AppID.** Every Android app has a unique application ID that looks

like a Java package name, such as *com.example.myapp*. This ID uniquely identifies the app on the device and in Google Play Store. If a developer wants to upload a new version of his app, he must keep the same application ID as the original APK, otherwise, Play Store treats the APK package as a completely different application.

iii. **AndroidID and UDID.** AndroidID is an another unique identifier used for tracking the devices. In general, AndroidID and UDID are used by ad networks and advertisers to track the user's devices for targeted advertisements. Typically, apps would pass this ID to ad networks, which would store it and use it to track users as they interacted with various apps. Thus, it can be unsafe if device ID is sent to unauthorized users of malicious attackers together with other user information. The Android ID could only be reset by wiping the entire device. Notably, many developers leverage IMEI number and device ID as UDID [49, 45]. Android apps can read this identifier from the device if *READ_PHONE_STATE* permission is granted.

In the release of Android 4.4, Google introduced a new identifier called the *AdvertisingID*. This new ID is subject to a new user setting can be used to opt-out of *behavioral advertising*. According to Google, when a user activates this Opt-Out setting, the Advertising ID will only be used for contextual advertising, reporting security and fraud detection. To enhance privacy, the new advertising identifier is not connected to personally-identifiable information or associated with any persistent device identifier (e.g., IMEI, MAC address, etc.).

(c) **Contacts.** Contact information provides the app access to the address book of the Android device. An application can access contact information

directly by calling ContentResolver methods or by sending intents to a contacts app. The *READ_CONTACT* permission is required by an app to access the address book.

(d) **CurrentApps.** Applications in Android can check the status of currently running applications in the OS. For this they have to call Activity Manager by using *getRunningAppProcesses* or *getRunningTasks* in their app.

2. **Sinks.** The tainted data identified by dynamic platform leaves the device at a taint sink. The functionality that can help data exit the device are considered as sinks by TaintDroid.

(a) **Network/File.** Relevant API calls in the Webview, Outputstream, DataOutputStream, and HttpURLConnection objects are considered as sinks.

(b) **SMS.** The *sendTextMessage()* function in object *SmsManager* is also considered as a sink. SMS information represents the SMS storage of the device. Malicious applications can export the SMS data from the device to a Microsoft Excel file or remote SQL database. To fetch SMSs from the device, the *READ_SMS* permission is required. To send SMSs from the device, the *WRITE_SMS* permission is required.

5.5 Cross Validation

To compare the results of our original approach with other existing methods, we tested 100 randomly selected applications from our dataset in different environments. For this, task, we scripted to load the applications automatically into the Nexus 4 phone, start the application, and capture the logs generated by the dynamic analysis framework.

In the first case, we manually executed randomly selected applications with only TaintDroid running on the device and recorded the application activity. In the second case, we use a combination of Android Monkey [25] and TaintDroid. Android Monkey is used to randomly exercise the user interface (UI) of the application. Monkey is a program running on Android that feeds the application with pseudo-random streams of user events such as clicks and touches, as well as a number of system-level events. While the triggered interaction sequences include any number of clicks, touches, and gestures, the Android Monkey specifically tries to hit buttons. As some use cases might require repeatable analysis runs without any random behavior introduced by the Monkey, we can also provide a fixed seed in order to always trigger the same interaction sequences. We follow the following steps to analyze applications with Monkey:

1. The installation of a given APK is done with the help of *adb*. Here the adb copies the APK into the device and then runs the *PackageManager*, which is an essential part of Android and installs it into the system. This means that the APK will be unzipped and copied to the specified directories.

2. After installation, the application is imported into the Android system and can be launched with the help of the Android Monkey.

3. As per our strategy, Monkey has the role of a simulator of human interaction on the to examined application.

During the runtime of Monkey, there are exactly 1000 generated events with 1000 ms delay in between. For example, after the application installation is completed the Android Monkey will be started via *adb shell monkey -p $ACTIVITY -vv –throttle 1000 1000*. This tells the Monkey to start the activity associated with the application and generate 1000 random user events which will be used to simulate normal user

behavior. Initially, we chose more than 1500 random events to simulate normal user behavior and tested it on few applications. However, we noticed that applications started crashing after a certain number of events (e.g., 1000), therefore we choose 1000 events to simulate the stable user behavior and prevent applications from crashing.

In addition to analyzing the logs automatically, we manually inspected the log files generated under the proposed testing scenarios and compared the results of both approaches with our original evaluation results.

Although the number of entries in log files is typically large enough for manual inspection, this approach is obviously not scalable and the system can be improved by developing further techniques.

Chapter 6

Dataset and Setup

In this chapter, we describe our analysis dataset and the setup we used for our analysis. We collected free APKs available on the Play Store (a Google-proprietary service which allows browsing and downloading of applications that were published by different developers) and other sources as explained below. A total of 5,000 APK files were collected. Following are the details of our dataset.

1. **Android Apps from CCSL** We collected applications from Carleton University's App Observatory [1] (managed by Carleton Computer Security Lab). We analyze around 4500 applications from this dataset source. The dataset has previously been used in existing research [6].

2. **PrivacyGrade.org** This website characterizes permission leakage in popular free Android apps available on Google Play Store applications [35]. Grades are assigned using a privacy model that designed by the developers of the website. Grades range from A to D, where *A* stands for least privacy offending application and *D* stands for most offending applications. This privacy model measures the gap between people's expectations of an app's behavior and the app's actual behavior. For example, most people don't expect applications

like *Calculator* to use location data, but surprisingly, many of them actually do. This kind of surprise is represented in the privacy model as a penalty to an app's overall privacy grade. In contrast, most people do expect apps like Google Maps to use location data. This lack of surprise is represented in PrivacyGrade's privacy model as a small or no penalty. We analyze around 500 applications from this dataset source rated in C and D category of PrivacyGrade.

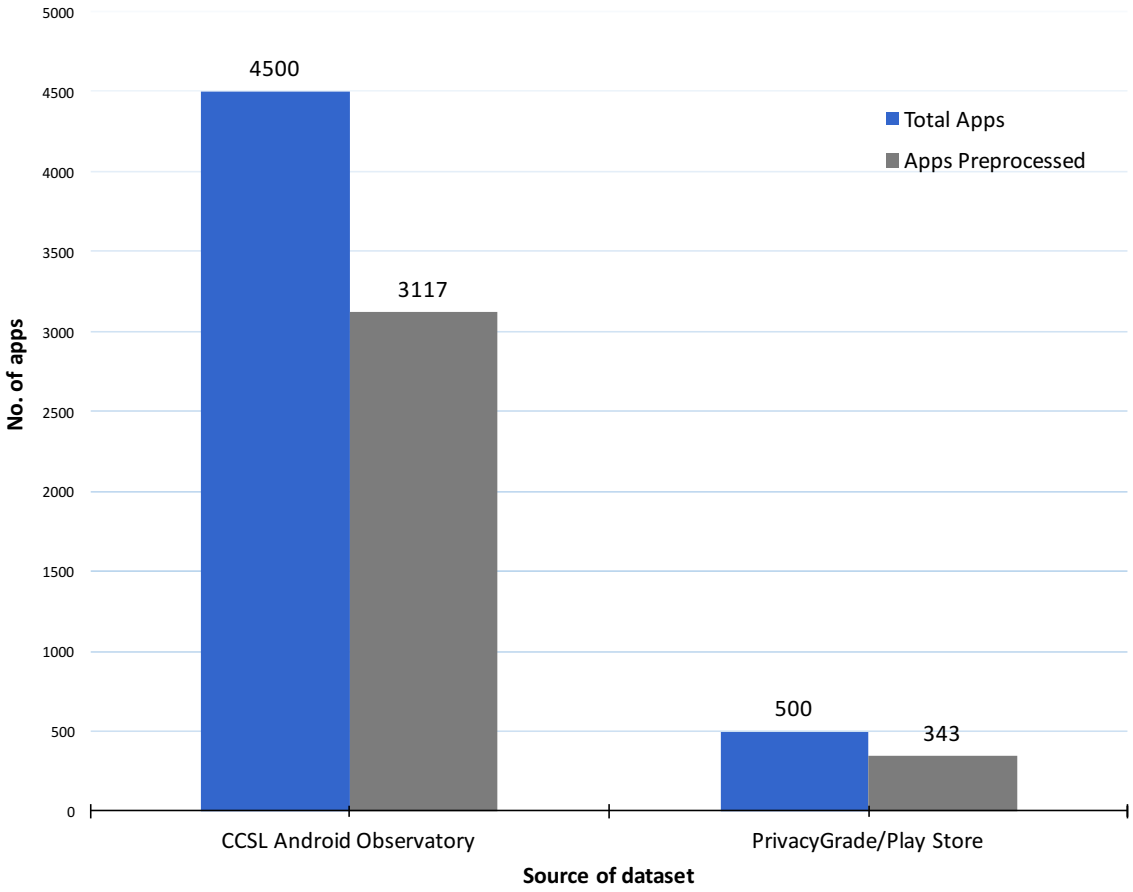

Figure 5: Dataset.

Pre-processed APKs. The target APKs were pre-processed by IntelliDroid's static analysis component. In this phase, IntelliDroid derives the path constraints for each targeted API in application code. In some cases, IntelliDroid was not able

to built API input constraints within a reasonable amount of time. In this case, we simply set a timeout of 300 seconds. As seen in Figure 5, around 70% of total applications were able to go through static analysis phase, rest 30% of them failed to compile in specified amount of time.

As per our final pre-processing results, despite the technical difficulty in certain cases, our overall approach offers great practical value for the analysis of trigger-based behavior in Android applications.

6.1 Setup

In existing dynamic analysis work [9, 19, 55, 30, 32, 22, 63], researchers rely on dynamic analysis to extract privacy sensitive behaviors and often use Android emulators for the experiments. However, Android emulators have some limitations.

Currently, Android emulators cannot emulate following components that are likely to be used by malicious applications.

1. No support for IMEI number (00000 is returned, IMEI information leaks accounts for the maximum share of all the data leaks in our results).
2. No support for MAC Address.
3. No support for real world GPS testing.
4. No support for Camera.
5. No support for Sensors (Acceleration sensor, Gravity sensor, and Gyroscope)
6. No support for determining network connected state (This check is present in most of the applications)

Also, a number of companion chips like WLAN, Bluetooth, GPS, Radio are present on the actual device and interact with the CPU in ways that are not predictable and hard to simulate on the emulator. Moreover, malicious apps may detect emulation [16, 56, 52] and as a result, it does not execute the payload to evade the detection.

To overcome this state of affairs, we use an LG Nexus 4 device with Quadcore 1.5 GHz CPU, 2 GB memory, and 8 GB internal storage for our experiment. The test device runs a custom built Google Android firmware, i.e., Jelly Bean 4.3 with Linux kernel version 3.4.39.

For the dynamic analysis setup, we download and compile the latest TaintDroid targeting aosp-arm-eng, which is based on Android 4.3 released in July 2013 (android-4.3-r1). On the other hand, for IntelliDroid dynamic analysis client, we download *IntelliDroid dynamicclient* patches from IntelliDroid's Github [62] and patched them to the Android base source code.

Chapter 7

Results

In this chapter, we give a summary of detected privacy leakages during dynamic analysis. Figure 7 summarizes the information sources most commonly leaked to the network by the applications.

Leakage specific to uniquely identifying the device is significantly more than apps accessing resources: overall, 33.09% of total apps leak information over the network. Most applications leak device specific identifiers, such as the IIMEI, IMSI and the Android ID. The case of IMEI information leaks accounts for the maximum share of all the data leaks. We find that around 76% of applications were leaking IMEI

Name	**Category**	**Leaked Data**	**Fetching Method**	**No. of Downloads**
Brightest Flashlight	Productivity	IMEI, Location	HTTP	50M-100M
Camera 360*	Photography	IMEI, Location	HTTP	450M-500M
GoBattery	Tools	Location	HTTP	10M-20M
GoLocker	Personalization	IMEI, Location	HTTP	100M-150M
GoSMS Pro*	Communication	Address book, IMEI	HTTP	100M-150M
Fruit Ninja*	Games (Arcade)	IMEI	HTTP	100M-500M
My Talking Tom	Games (Casual)	IMEI	HTTPS	100M-500M
AngryBirds*	Games (Arcade)	IMEI	HTTP	500M-1000M
Trial Xtreme*	Games (Racing)	IMEI	HTTPS	50M-100M
Speedtest	Tools	IMEI, Location	HTTP	10M-50M

Table 4: Information leaks detected by IntelliDroid and TaintDroid for the most popular applications (as of March 2017); Applications marked with (*) have multiple versions.

number of the device. Interestingly, some applications transmit device's IMEI details up to 16 times to several servers during the runtime.

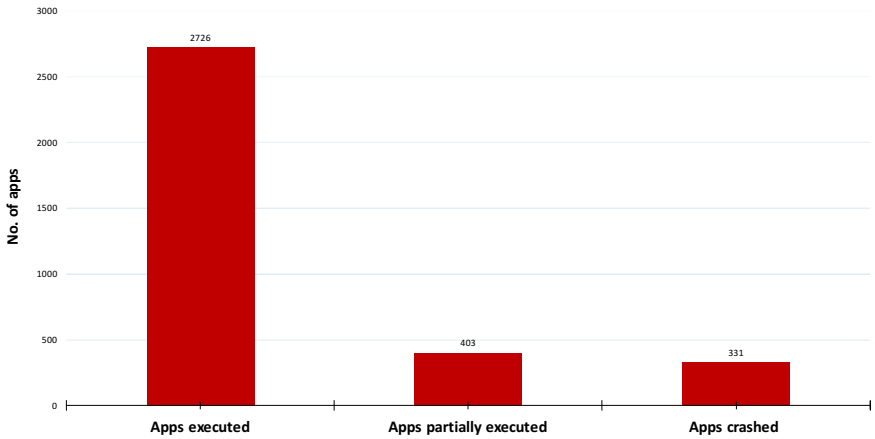

Figure 6: Statistics of applications

An existing report [41] has identified that leakage of names and phone numbers from the user's address book is more common amongst malware than it is amongst benign apps. Instead, regular applications mainly leak the location, an information source less commonly leaked by malware applications. In 30% of applications, user's location coordinates were leaked out. Android has two categories of location data: coarse and fine. Coarse location data uses triangulation from the cellular network towers and nearby wireless networks to approximate a device's location, whereas fine location data uses the GPS module on the device itself. We do not differentiate between coarse and fine location data as we believe any leakage of location information to be important.

For instance, popular fitness applications continuously track the distance of the running course and allow the phone user to calculate the calories. This kind of apps reads device location continuously no matter if the application is running in the background. We also observed similar kind of pattern of location tracking in navigation and transportation applications. Among the leaked data types, the location and GPS location seem similar but actually, have a delicate difference in getting the location

information. There could be several ways to find the location of a phone. Applications can fetch the location information of a device through the network instead of the GPS. In particular, when the user is inside a building or surrounded by high-rise buildings, known as a GPS Dead Zone, the location should be resolved by the network. In our experiment, the location and GPS location are both tainted whenever the location data leaks happen. However, most privacy leaks occur at the early time of app's execution and discontinue without explicit inputs.

Among all the identifiers, Android ID is the one with least risk, as it can be changed at any time [33]. Around 48% of applications leak Android ID. Other identifiers like UDID i.e., 45% of apps are leaking can be used for long-term tracking. Developers manage to assign the application any persistent device identifier marked as UDID. For example, many developers assign UDID identifiers value of IMEI, IMSI, etc. In general, the identifiers are permanently associated with either the device or the SIM card.

Concerning accessing information on the device, only 3 (0.11%) applications tried to access SMS and address book during the dynamic analysis phase. Due to very small number of applications leaking address book and SMS details, we do not list them in Figure 7 (same for the apps that access system services). Also, there is is a possibility that these applications were accessing address book content just for the functionality of the app.

We also characterize information leaks for the most popular applications (with download's in millions) from our dataset. Table 4 summarizes our characterization results for the most popular apps. In this table, we also crawl the number of downloads to highlight the number of affected users.

Figure 8 shows statistics of apps (in %)that leak atleast one identifier from each category. This categorization is based on a total of 1500 apps. A large number

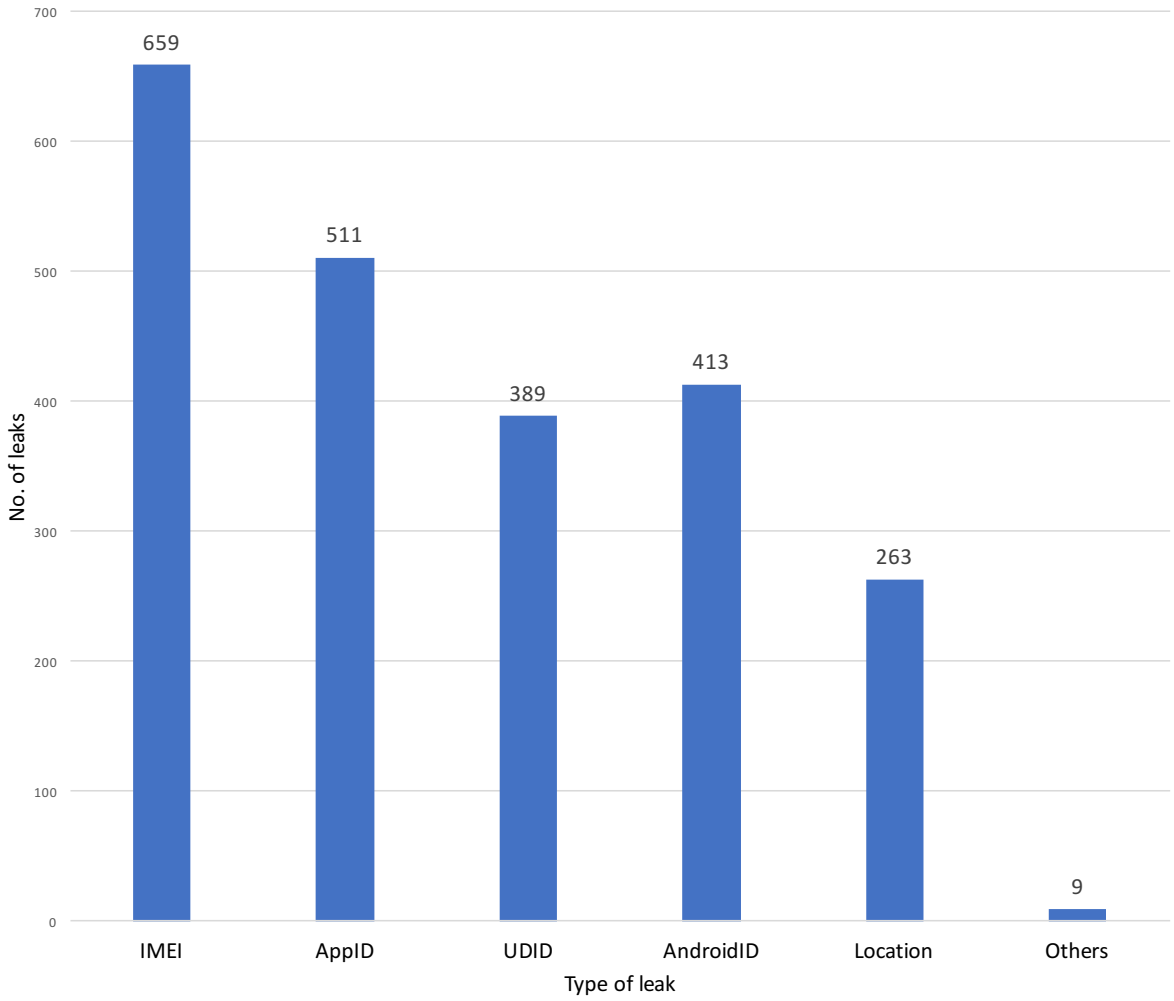

Figure 7: Distribution of identifiers leaking.

of the APKs in our target dataset were not categorized. Also, tracking category of each uncategorized APK manually is a very time-consuming task.Therefore, our categorization results are based on a subset of our dataset (i.e., for 1500 apps).

Categorical classification shows that transportation and weather applications account for the maximum share of location data leaks. It is notable that most of the applications in these 2 categories highly rely on permissions and functionality (e.g., GPS navigation apps, location and device tracking for weather updates etcetera). We are uncertain if these identifiers and sensitive data are used for malicious purposes. The problem is that many apps may have a good reason to use them because several

different things can be covered by one permission and there's no good place to see exactly what they all mean. It is possible that sometimes an app that is tracking different identifiers and location information is saving the settings in the cloud to tie them to a user. Applications like Google Maps, Waze, Community GPS navigation, ROUTE 66 etcetera heavily rely on features like location tracking to uniquely identify the device. A rough location fetched through a Wi-Fi Access Point database works well enough for tracking approximate location but sometimes applications need to get precise.

Utility applications like *Swift Wifi*, *DolDolLauncher*, *GoLocker*, *GOweatherForecast*, *GoSMS*, *Camera 360* and *My talking tom* use several permissions to grab mobile analytics data. As seen in Figure 7, many applications have full access to the phone state, IMEI, IMSI/phone number and location of the device. Also, the frequency of connections made with the remote server is very high. After doing a manual check, on every *GO* app, including *GoLocker* and *GoLauncher*, we found that they all have the same design and pattern.

Similarly, popular games *Fruit Ninja*, *Drag Racing* access full phone state every minute. Kids reading book *NoraEGame* access device information up to 60 times. On the other hand, popular utility app *Brightest flashlight* application for LED torch fetches the device precise location and sinks out the location information to a remote server. It is very clear from these applications that accessing location and unique details of the device has nothing related to the functionality of the application.

It is also possible that ad libraries associated with the applications may use fetched information for targeted advertisement. An ad library linked with application application may correlate a user's ad traffic across each application they've installed containing their ad library. This is because every ad provider consistently transmits the same UDID field (hashed or unhashed UDID value) regardless of the application

in which it is included. For example, if one ad library in an application sends Android ID or IMEI, every application on a user's device containing the same ad library will transmit the same value, allowing these companies to correlate the information provided by all of the user's applications that use the exactly same ad library.

Additionally, a further concern involves a network sniffer that may track users private information. There are cases in which many applications transmit device identifying information in clear text.

As seen in Figure 7, many applications transmit location information to the developers. For example, we found a Calculator application capturing both IMEI and location of the device to the developer directly. This shows that many users are not aware of the permissions used by the applications. There are chances that the developer may use this location for various malicious purposes but it's also not obvious, why an app needs the permission.

We found that, in around 10 seconds, IMEI and AppID information are likely to be leaked no matter if input triggers have started or not. On the other hand, the remaining of identifiers and location leaks seem to take place when all the inputs are triggered normally. We also found that most other data leaks occur while the apps under test are triggered functions by IntelliDroid dynamic client framework.

For our analysis, we also collected a miscellaneous dataset of 200 applications from Carleton's App Observatory dataset. These applications were rated as the safest applications with no leakages by F-Droid. In our analysis, we found only 2% of the applications leaking device information to the remote server. The low detection rate validates that most applications were safe and were leaking no sensitive information off the device.

As already discussed in Section 4, TaintDroid only supports Android applications targeting version 4.3 or lower. After the release of Android 4.4 (KitKat), Android

got a major architectural update as, Google decided to introduce another way of executing apps on top of the Android operating system i.e., ART. Android 4.3 uses Dex bytecode, which is completely different from ART. Our results show that the applications developed for Dalvik work normally when running with ART. Whereas, applications developed for ART fail to execute on Dalvik after initialization.

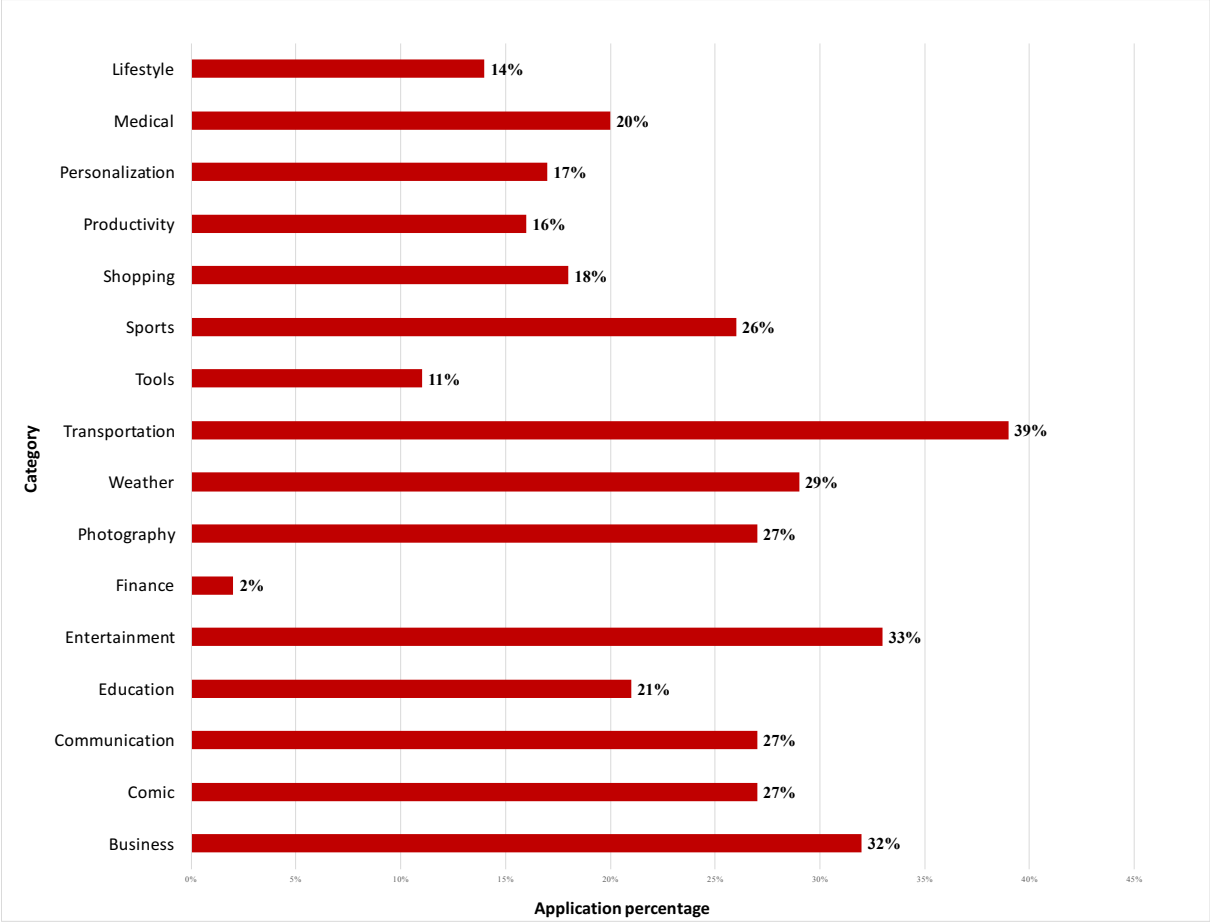

Figure 8: Statistics of apps that leak atleast one identifier from each category. This categorization is based on a total of 1500 apps.

7.1 Cross Validation

We perform three different experiments for cross validation of our results. All three analysis environments are enumerated below;

1. IntelliDroid and TaintDroid

2. TaintDroid and Android Monkey

3. TaintDroid only

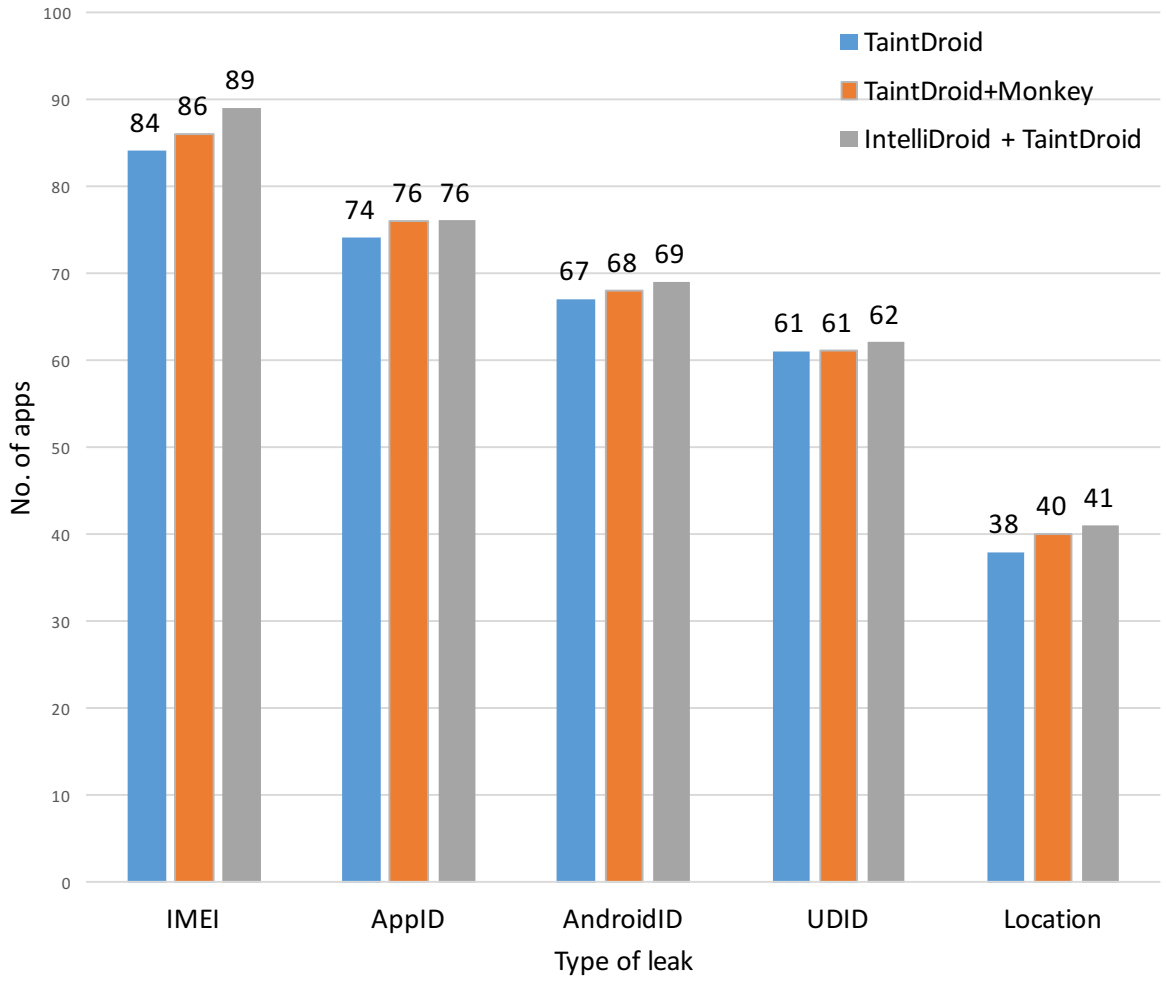

Figure 9: Cross validation evaluation results.

As described in Chapter 5, we use static analysis to generate inputs for the target app. We then trigger each input automatically from the host machine and calculate the differences in the leaks during the phase of dynamic analysis.

Based on the summary of the logs collected for the evaluation, we found that some applications leaks are activated based on events that are independent of the user interaction while some others are only activated on the activity triggered by Android Monkey and IntelliDroid.

In a few cases, IntelliDroid input triggers for an application to leak details of IMEI (or ICCID) leak while with only TaintDroid there is no leak. This is because, with IntelliDroid, the different execution paths of the application are parsed in advance (during static analysis) and later triggered programmatically.

Overall results for comparative analysis are shown in Figure 9. In total, IntelliDroid and TaintDroid were able to detect around 89 applications leaking IMEI details to a remote server. Whereas, in the case of app executed under TaintDroid enviorment only, TaintDroid failed to detect 5 applications leaking the same identifier. This result states that in the scenario of manual testing, only a small number of functions are covered by interacting or running the application. Hence, there are fewer leaks as compared to automatic input trigger methods.

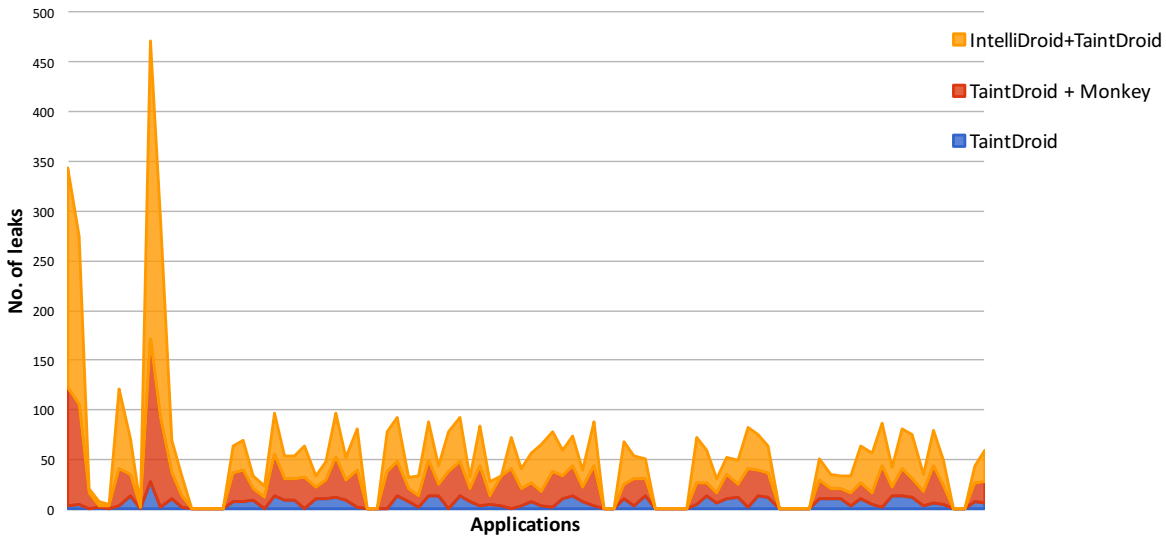

Figure 10: Comparision between frequency of the user data leakage using three different methodologies.

The major observation we made in the comparison experiment is the difference

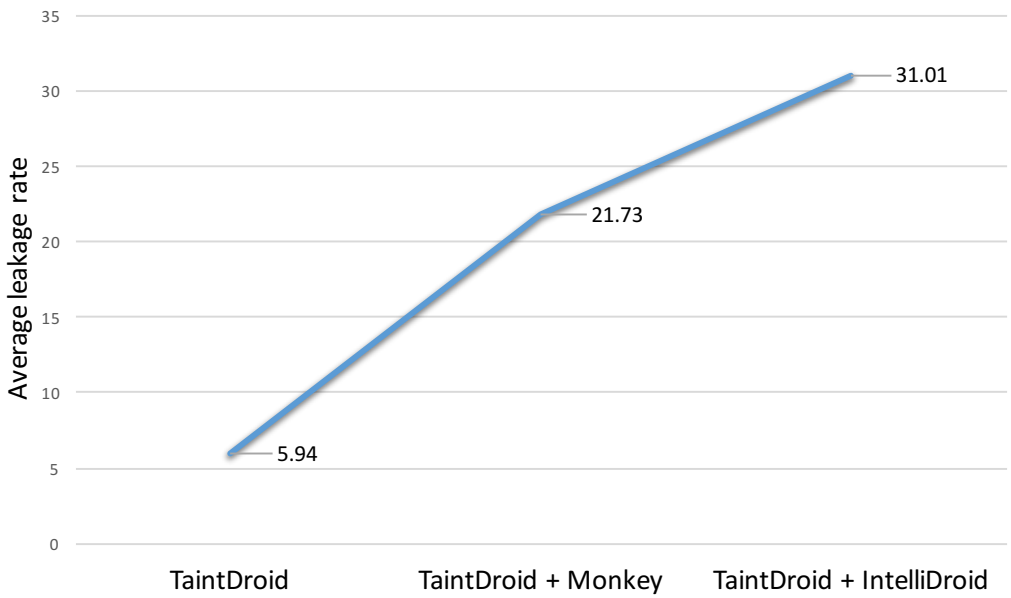

Figure 11: Average rate of frequency leakage using three different methodologies.

in the number of requests being sent to the remote servers in all three cases. Figure 10 shows the comparison of the rate of frequency leakage for a subset of applications we analyzed. Similarly, Figure 11 characterizes total average of the frequency of identifiers, leaked by all the applications in our dataset.

For all the applications, specifically for the TaintDroid only environment, we recorded little count of information leaks in our logs (in terms of frequency of leakages). However, in the case of Android Monkey and IntelliDroid, we detect that the applications attempted to establish a connection to a remote server multiple times (the highest for IntelliDroid). For example, *Camera 360*, a photo editing application established a connection to a remote server more than 200 times/min. Whereas, only 3 times when running under the scenario of simple TaintDroid and 120 times, in the case of Android Monkey. Similarly, we found that content sharing application *Bump* also leaks the device identifiers and location multiple times with a remote server.

In our results, IntelliDroid was able to detect 41 application leaking location details. However, in the case of only TaintDroid, we were able to detect 38 applications

leaking location information and missed 3 applications leaking location information to a remote server. Android Monkey also missed 1 location leak as compared to IntelliDroid.

The Monkey program tends to generate bugs with the instrumentation. 15% of the total applications crashed while we were using Monkey on them. On the other hand, Android Monkey and IntelliDroid do not use the same approach to trigger paths. Still, they have approximately the same number of detection in case of tracking AppID.

7.2 Analysis Time

Our static analysis phase costs around 416 hours to analyze all 5,000 apps. The analysis time can be further reduced by distributing the analysis workload to multiple machines.

Our dynamic taint analysis costs 3 to 5 minutes to verify a certain path reported by static analysis, depending on the search space and the complexity of the app. 70% of total apps were executed in the dynamic analysis phase, rest 30% of them failed in pre-processing (as discussed in Chapter 6). The dynamic taint analysis costs us around 250 hours to analyze approx 5000 apps.

Chapter 8

Limitations

1. **Implicit Flows.** A fundamental limitation of dynamic taint tracking is the inability to track implicit information flows via control flow. TaintDroid shares this limitation.

2. **OS Compatibility.** The dynamic analysis framework TaintDroid was originally designed for the virtual-machine-based system (i.e., Dalvik VM), and implemented on Android version 2.1 to Android 4.3. The core functioning of the framework utilizes the internal memory of Dalvik VM for taint storage and propagation. However, to enhance the performance and battery life of Android devices, Google pushed changes to the master branch of AOSP that remove the Dalvik virtual machine and replace it with ART that uses ahead-of-time (AOT) compilation runtime system. The newer ART runtime is an Ahead-of-Time compiler that processes application instructions before they are needed. Standing for Android Runtime, ART was introduced in Android 5.0 and replaced Dalvik as the platform default. Therefore, we cannot use TaintDroid for the newly-designed runtime and can only be supported up to Android 4.3.

3. **Application Compatibility.** Most Android apps should just work without

any changes under ART. While we were analyzing the apps, we found that many applications were crashing (the one's we downloaded from Google Play Store). The applications in our test suite were crashing because of the app developers always tend to target newer Android versions so as to use latest features. Unfortunately, the newer features are written for applications according to ART system, that TaintDroid framework does not support. Therefore, TaintDroid is not able to analyze many new applications.

4. **Native Code Taint tracking.** It is known that TaintDroid tracks taint calls for Dalvik bytecode only. Native code taint tracking would likely require dynamic binary instrumentation or VM introspection. TaintDroid currently does not use such methods for native code taint tracing; these methods result in a typical slowdown of 10x to 30x [50] for the code and hence are not very attractive from the performance perspective.

Chapter 9

Conclusion and Discussion

Along with the increasing prevalence of Android smartphones, the number of Android apps including malware apps is increasing. In spite of deployed Android security mechanisms, many apps take advantage of the default Android security weaknesses to misuse the granted resources. The increasing capabilities of the Android devices open new doors for attackers to exploit different types of links, sensors, services related to user's device. Thereby, researchers have proposed many tools and frameworks to control the outreach of vulnerabilities in Android devices.

In this dissertation, at first, we surveyed 32 most promising proposals for two major app analysis categories (i.e., static and dynamic analysis). From our survey, it is evident that both static and dynamic analysis techniques have its own limitations. Depending on the main objective of each proposal, the way of implementation for proposed works is different. The proposed works are primarily behavior-based and their main contribution is tracking the apps privacy-sensitive behavior and also to restrict them from doing any kind of malicious activities. We highlighted and characterized several important features of

state-of-the-art app analysis proposals and introduced their methodology. We also compared them considering their type, features, functionality and known limitations.

For the evaluation, we conclude that using static analysis tools in conjunction with dynamic analysis tools provide clearer results than rest of the methods. We found that about 33% of the tested apps leak privacy-sensitive information over the network (e.g., IMEI, location, UDID), which is consistent with existing work. The case of IMEI information leaks accounts for the maximum share of all the data leaks. Furthermore, we also report an overall increase in the frequency of leakage of identifiers (with IntelliDroid and TaintDroid).

Many dynamic analysis proposals rely on testing application behavior on Android emulators [7, 9, 67, 66, 46, 37]. Due to several limitations of emulators as discussed in Chapter 6, we wanted to test applications on a real device and find out the difference. On the basis of our results, we conclude that the using a real device is more promising as compared to the emulator. It provides more realistic values of device identifiers (e.g., IMEI, ICCID, UDID), which is not possible in case of emulators).

Considering the limitations of automatic exercisers (like Android Monkey) or manual analysis, it is clear that the approach used by IntelliDroid is better than Android Monkey or other UI exercisers. The technical problem with Android Monkey and similar tools is they cannot guarantee that all malicious behaviors can be triggered during the testing of apps. As an example, consider a malicious application with special characteristics. In this case, the application will fetch user's personal information after 2 hours of app launching (*time bomb*) or only when it is attached to a PC (*logical evasion*). In this case, if we will use Android Monkey exerciser to trigger this behavior, it is expected to fail. On the other

hand, there is a chance that IntelliDroid may trigger the behavior (provided that target API is known). It is notable that some events are independent of user interactions with the application (i.e. the existence of the network, etc), yet some others are based on user input.

However, testing mobile application is a non-trivial effort due to a variety of inputs and heterogeneity of the technologies. Therefore, it is difficult for UI based exercisers to have full coverage.

With this snapshot of the overall app analysis frameworks landscape, we thus hope that the security community can better explore various potential opportunities to further design robust and promising solutions to detect privacy leakages in Android applications and devices, including addressing the remaining challenges.

Bibliography

[1] Android Observatory. `https://androidobservatory.org/`.

[2] TWRP recovery framework. `http://twrp.me/`.

[3] Y. Aafer, W. Du, and H. Yin. DroidAPIminer: Mining API-level features for robust malware detection in Android. In *International Conference on Security and Privacy in Communication Systems (SECURECOMM'13)*, Sydney, Australia, 2013.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Network and Distributed System Security Symposium (NDSS'14)*, San Diego, USA, 2014.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM Sigplan Notices (ICFP'16)*, New York, NY, USA, 2014.

[6] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *ACM Conference on Computer and Communications Security (CCS'10)*, Chicago, IL, USA, 2010.

[7] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An Android application sandbox system for suspicious software detection. In *5th international conference on Malicious and unwanted software (MALWARE'10)*, Nancy, France, 2010.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new Android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (CCS'11)*, Chicago, IL, USA, 2011.

[10] P. P. Chan, L. C. Hui, and S.-M. Yiu. Droidchecker: analyzing android applications for capability leak. In *ACM workshop on Security and privacy in smartphones and mobile devices (WiSec'12)*, Tucson, Arizona, USA, 2012.

[11] CheckPoint. Gooligan Checker. `https://gooligan.checkpoint.com`.

[12] P. M. Damien Octeau and S. Jha. DARE: Dalvik retargeting. `http://siis.cse.psu.edu/dare/`.

[13] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In *Mobile Security Technologies (MoST'12)*, San Francisco, CA, USA, 2012.

[14] A. Desnos. Androgaurd. `https://github.com/androguard/androguard`.

[15] A. Desnos and P. Lantz. Social media apps are tracking your location in shocking detail. `https://code.google.com/p/droidbox`.

[16] H. Dharmdasani. Android.HeHe: Malware now disconnects phone calls. (January. 21, 2014). `https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html`.

[17] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium (USENIX'11)*, San Francisco, CA, USA., 2011.

[18] L. Dong. APKparser checker. `https://github.com/clearthesky/apk-parser`.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *ACM Transactions on Computer Systems (TOCS'14)*, New York, NY, USA, 2014.

[20] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. `http://www.enck.org/pubs/NAS-TR-0094-2008.pdf`.

[21] A. Fox-Brester. Android Gooligan hackers just scored the biggest ever theft of google accounts. (November. 30, 2016). `http://www.forbes.com/sites/thomasbrewster/2016/11/30/gooligan-android-malware-1m-google-account-breaches-check-point-finds/#569191e4470d`.

[22] P. Gilbert, L. P. Chun, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services (MobiSys'11)*, Bethesda, MD, USA, 2011.

[23] Google. adb. `https://developer.android.com/studio/command-line/adb.html`.

[24] Google. Android official documentation: Introduction to Android. (March. 13, 2017). https://developer.android.com/guide/index.html.

[25] Google. UI/Application exerciser Android Monkey. https://developer.android.com/studio/test/monkey.html.

[26] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *Network and Distributed System Security Symposium (NDSS'15)*, San Diego, CA, USA, 2015.

[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys'12)*, Ambleside, United Kingdom, 2012.

[28] A. Greenberg. Researchers say they snuck malware app past Google's Bouncer Android market scanner. (May. 23, 2012). https://www.forbes.com/sites/andygreenberg/2012/05/23/researchers-say-they-snuck-malware-app-past-googles-bouncer-android-market-scanner/#105adee07816.

[29] L. Hautala. How to tell if your Android phone has the HummingBad malware. (July. 7, 2016). https://www.cnet.com/how-to/hummingbad-how-to-tell-if-your-android-phone-has-a-bad-case-of-malware/.

[30] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security (CCS'11)*, Chicago, IL , USA, 2011.

[31] O. Hou. A look at Google Bouncer. (July. 20, 2012). http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/.

[32] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (ICSE'11)*, Waikiki, Honolulu, USA, 2011.

[33] HugeStreet. How to change IMEI, device ID of any Android device. http://www.hugestreet.info/2015/08/Free-Android-Device-ID-and-IMEI-number-changer.html.

[34] IBM. Speaking UNIX: Interprocess communication with shared memory. https://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/.

[35] Jialiu, S. Lin, S. Amini, K. Luan, M. Ku, B. Villena, and R. Ramachandran. PrivacyGrade: Grading the privacy of smartphone apps. http://www.privacygrade.org.

[36] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (Ubicomp'12)*, Pittsburgh, PA, USA.

[37] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS'14)*, Wroclaw, Poland, 2014.

[38] Microsoft. The Z3 theorem prover. `https://github.com/Z3Prover/z3`.

[39] J. Oberheide and C. Miller. Dissecting the Android bouncer. `http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/`.

[40] T. Ogasawara. Android apps now ask for over 200 kinds of permissions: Pew research. (November 12, 2015). `https://www.extremetech.com/mobile/217914-android-apps-now-ask-for-over-200-kinds-of-permissions-pew-research`.

[41] P. Olson. First-known targeted malware attack on Android phones steals contacts and text messages. `https://www.forbes.com/sites/parmyolson/2013/03/26/first-known-targeted-malware-attack-on-android-phones-steals-contacts-and-text-messages/#44397525a2d1`.

[42] G. Paller. Dedexer. `http://dedexer.sourceforge.net`.

[43] B. Pan. dex2jar. `https://github.com/pxb1988/dex2jar`.

[44] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium (USENIX'13)*, Washington, D.C, USA, 2013.

[45] C. Parsons. Privacy tech-know blog: Uniquely you: The identifiers on our phones that are used to track us. (December. 8, 2016). `http://developer.samsung.com/technical-doc/view.do?v=T000000103`.

[46] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy (CODASPY'13)*, San Antonio, TX, USA, 2013.

[47] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIACCS'13)*, Hangzhou, China, 2013.

[48] J. Sadeh and J. I. Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium on Usable Privacy and Security (SOUPS'2014)*, Ottawa, CA, 2014.

[49] Samsung. How to retrieve the device unique ID from Android device. (Jul. 8, 2011). `http://developer.samsung.com/technical-doc/view.do?v=T000000103`.

[50] M. L. Scott. *Programming language pragmatics*. `https://www.cs.rochester.edu/~scott/pragmatics/`.

[51] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for Android devices. *Journal of Intelligent Information Systems (JIIS'12)*, 2012.

[52] G. Sims. Obad was the nastiest piece of android malware discovered in 2013. (December. 16, 2013). `http://www.androidauthority.com/obad-nastiest-piece-android-malware-discovered-2013-324830/`.

[53] R. Smith. APKinspector. `https://github.com/honeynet/apkinspector`.

[54] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into Android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, New York, NY, USA, 2013.

[55] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for Android runtime. In *ACM Conference on Computer and Communications Security (CCS'16)*, Vienna, Austria, 2016.

[56] Symantec. Android pincer. (January. 21, 2014). `https://www.symantec.com/security_response/writeup.jsp?docid=2013-052307-3530-99&tabid=2`.

[57] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Network and Distributed System Security Symposium (NDSS'15)*, San Diego, CA, USA, 2015.

[58] B. R. Team et al. Sanddroid: An apk analysis sandbox. xi'an jiaotong university. `http://sanddroid.xjtu.edu.cn`.

[59] W. T.J. T.J. Watson libraries for analysis. `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[60] N. Vladimirova. New Android Hummer trojan may become the worst mobile malware in the history. (June. 30, 2016). `http://virusguides.com/new-android-hummer-trojan-may-become-worst-mobile-malware-history/`.

[61] R. Wisniewski. apktool. `https://ibotpeaches.github.io/Apktool/`.

[62] M. Wong. Intellidroid. `https://github.com/miwong/IntelliDroid`.

[63] M. Y. Wong and D. Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Network and Distributed System Security Symposium (NDSS'16)*, Chicago, IL, USA, 2016.

[64] R. Xu, H. Saïdi, and R. J. Anderson. Aurasium: practical policy enforcement for Android applications. In *USENIX Security Symposium (USENIX'12)*, Bellevue, WA, USA, 2012.

[65] Z. Yajin and J. Xuxian. Android Malware Genome Project. `http://www.malgenomeproject.org`.

[66] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium (USENIX'12)*, Bellevue, WA, USA, 2012.

[67] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in Android applications. In *ACM Workshop on Security and Privacy in Smartphones and Mobile (SPSM'12)*, Raleigh, IL, USA, 2012.

[68] M. Zheng, P. P. Lee, and J. C. Lui. ADAM: an automatic and extensible platform to stress test Android anti-virus systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*, Heraklion, Crete, Greece, 2012.

[69] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In *Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA, USA, 2012.