Techniques and Tools for Analyzing and Understanding Android Applications

By

LIANG XU

B.S. (Wuhan University) 2005
M.S. (Wuhan University) 2007

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Shyhtsun Felix Wu

Hao Chen

Committee in Charge

2013

Techniques and Tools for Analyzing and Understanding Android Applications

Dedication

To my parents, for their eternal love and support.

Contents

List of Figures

List of Tables

Abstract

With the rise of mobile technologies in recent years, smart mobile devices, such as smartphones and tablet computers, have become an integral part of people's daily lives. Like traditional cell phones, smartphones can be used for phone calls and text messaging. Driven by recent innovations, current smartphones have also become media players, GPS systems, digital cameras, portable gaming consoles, and Web browsers. The capabilities and functionalities of smart mobile devices have been significantly expanded by the large number of third-party applications. As we witness a phenomenal growth in the number and variety of mobile applications, their usage has become increasingly prevalent, yet we lack a systematic understanding of this emerging, important software ecosystem.

This dissertation presents novel, general techniques and tools for analyzing and understanding mobile applications. It focuses on analyzing applications on the Android operating system, which is currently the most popular mobile platform. This dissertation presents the first large-scale, systematic study of third-party Android applications. By investigating unique characteristics of Android applications, it uncovers and provides insights into common errors and pitfalls during application development and provides actionable recommendations where applicable. This dissertation also presents Heimdall, a static analysis framework that automatically tracks the flow of sensitive information in Android applications. Heimdall has been used to uncover potentially unsafe flows of sensitive information on a large scale, providing a comprehensive understanding of privacy-sensitive information usage in Android applications. The findings presented in this dissertation benefit the whole Android community, from platform and tool developers to application developers and end-users.

Acknowledgments

This dissertation would not have been possible without the assistance from many people. I would like to express my gratitude and appreciation to my family, friends, and colleagues for their support.

First and foremost, I am deeply grateful to my doctoral advisor Zhendong Su for his guidance and support. He has always been a patient, supportive, and inspiring mentor, and I am influenced by his passion, drive, and dedication to research. He has taught me how to do good research, from problem formulation to critical thinking and paper presentation. He has always set high standards of excellence and prompted me to pursue top-notch research. I am indebted to him for everything he has done for me.

I would like to also thank my colleagues and fellow students: Mehrdad Afshari, Earl Barr, Mark Gabel, Zhongxian Gu, Lingxiao Jiang, Taeho Kwon, Vu Minh Le, Andreas Sæbjørnsen, Thanh Vo, and Gary Wassermann. They have been invaluable resources for both research and everyday life. I appreciate their useful feedback and constructive comments on research ideas and paper drafts, which help greatly improve my work.

Outside of UC Davis, I would like to thank Anders Møller and Mathias Schwarz for their collaborative research efforts. Our combined interests, expertise, and efforts have led to the creation of Heimdall, which is an important piece of this dissertation. It has been a real pleasure working with them.

Last but not least, special thanks to my partner both in life and in research, Fangqi, for sharing the trying times as graduate students and for her constant love and support. My most heartfelt thanks also go to my parents for their unconditional love, encouragement, and support during my years in graduate school.

Chapter 1

Introduction

Smart mobile devices, including smartphones and tablet computers (tablets), have led to a new, popular digital platform for our daily activities. Their portability and rich functionalities have attracted a large number of users. A recent study by comScore reports that there are over 130 million smartphone users in the U.S. and the number has been growing rapidly [34]. Globally, smartphones are already outselling PCs [27]. Approximately two thirds of the consumers choose smartphones when they purchase a new mobile phone [75], and it was estimated that there would be one billion smartphone users worldwide by 2012 [35].

An appealing feature of smart mobile devices is their ability to run a wide selection of applications (apps) covering various categories, from education and business to gaming and entertainment. Mobile application usage is soaring [74] along with the success of digital application distribution platforms. As of May 2013, Apple's App Store hosts over 850K iOS applications, passing 50 billion downloads since its launch in 2008 [20]. Growing at an even faster rate, the Google Play Store hosts more than 700K Android applications [85], and it is predicated to hit one million applications by June 2013 [70, 86]. Recent report indicates that people on average spend more time using mobile applications than browsing the Web, and mobile applications will soon challenge broadcast television as the dominant channel for media consumption [63]. The term "app" was voted "Word of the Year" by the American Dialect Society

in 2010 [89], and smartphones are sometimes referred to as "app-phones" for their distinctive capabilities of running third-party applications [87].

Google's Android has become the most popular mobile platform [28, 34, 51, 60], and it is still growing at a rapid pace. As of May 2013, over 900 million Android devices have been activated worldwide, and the Google Play Store passed 48 billion application downloads, with more than 2.5 billion downloads in a single month [79]. This tremendous growth has created and shaped a large ecosystem of Android applications. However, we do not yet have much in-depth understanding of the unique characteristics of this important, emerging software ecosystem.

This problem — the general lack of understanding in mobile applications — is the focus of this dissertation. It presents techniques, frameworks, and tools for automatically analyzing third-party Android applications and aims to provide a systematic understanding of such mobile applications.

This dissertation is organized into five chapters. Chapter 2 provides background information on the Android platform, and each of the following two chapters forms a distinct set of contributions:

Chapter 3 In contrast to traditional applications, mobile applications have appeared relatively recently with a number of characteristics unique to the mobile paradigm; hence they present a new domain with additional challenges for analysis techniques. Chapter 3 presents the first large-scale, systematic study of third-party Android applications. Focusing on their unique aspects, we have extensively examined the manifest files, permission usage, and application bytecode. We have uncovered a large number of common errors and pitfalls, and provided insights from the coding and development perspective. Our main findings are summarized below:

- The majority of applications contain errors or suffer from structural issues in their manifest files. Specifically, elements and attributes are misspelled, misplaced or misused, which may lead to bugs, unexpected behavior or even security vulnerabilities.

- Application developers often request incorrect permissions (*e.g.*, misspelled, non-existent, deprecated, or protected ones). Such errors can be attributed to developer confusion or lack of clear, detailed API documentation. Incorrect permission usage may lead to bugs or even runtime crashes.

- We also identify and quantify a number of characteristics of application code. For example, we reveal the most frequently used Java and Android-specific library APIs. We also found that applications commonly use Java reflection and often contain obfuscated code.

The study also sheds light on the root cause of such errors, discusses their implications, and provides actionable recommendations where applicable. Understanding Android applications on such a large scale provides practical benefits for improving this large, new software ecosystem.

Chapter 4 As mobile devices collect and compile an increasing amount of private or sensitive information about users, security and privacy issues have become an important concern. Although the Android system uses coarse-grained permissions to warn about sensitive information access by applications, users usually have little knowledge of or control over how their privacy-sensitive data is used. There has been extensive research on the information leakage problem, and nearly all prior work only considers the exfiltration of data through the network. While the proposed tools are effective, they are limited in scope and little is known as to where and how sensitive information flows in general. This motivated us to build Heimdall, a static analysis framework that automatically tracks sensitive information flow in Android applications.

As a new way of approaching the problem, we consider both malicious applications (with intentional leaks) and vulnerable applications (with inadvertent leaks) and monitor multiple taint sinks. Based on static taint analysis, Heimdall tracks the flow of sensitive information and raises an alarm when such information flows to unsafe sinks that can be accessed without the guarding permissions used to retrieve the information. To analyze real-world applications, Heimdall performs analysis on bytecode.

Using Heimdall, we have analyzed a large number of applications and uncovered four common types of unsafe sinks: log output, network connection, public files, and SMS messages. The tool is practical — it found pervasive flows of phone state and location data, and frequent flows to log and network sinks. Our extensive evaluation also shows that Heimdall is precise and scalable for real-world deployment. We believe that the work also provides a much deeper and more complete understanding of sensitive information usage in third-party Android applications.

Finally, Chapter 5 concludes and outlines a few interesting directions for future work in this area.

Chapter 2

Background

This chapter provides background on the Android platform necessary to understand the remaining chapters.

Android is an open-source Linux-based operating system designed for mobile devices, including smartphones and tablets. It has recently been adopted for other embedded devices, such as smart TVs and media streaming players. Android applications are typically written in Java. They are first compiled to standard Java bytecode, which is then converted to the customized Dalvik bytecode format, and packed into *Application Package (APK)* files.

2.1 APK File Format

Applications are distributed and installed on the Android platform as APK files. Each application is compiled and packed into a single APK file that contains Dalvik bytecode, resources (*e.g.*, images and language translations) and a manifest file. An APK file is essentially a compressed ZIP archive. Figure 2.1 shows the structure of an APK file, which typically contains the following:

- `assets`: the directory containing raw resource assets that are not compiled.
- `lib`: the directory for native library binaries. Although Android applications are written mostly in Java, some of their functionalities may be implemented in native code [12].

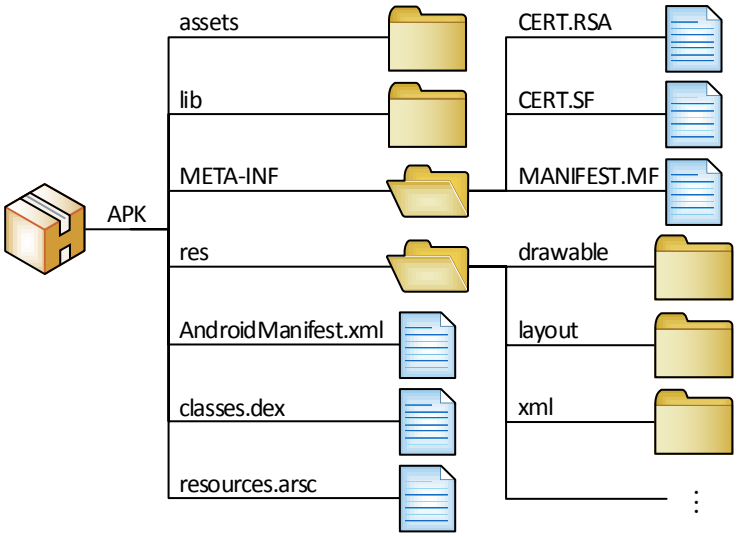


Figure 2.1: APK file structure.

This directory is optional as most Android applications do not contain native code.

- `META-INF`: the directory holding meta-data and certificate information.
- `res`: the directory for resources not compiled into `resources.arsc` (described below). This directory contains a few subdirectories, such as `drawable`, `layout`, and `xml`.
- `AndroidManifest.xml`: the manifest file containing the essential information about an Android application.
- `classes.dex`: classes compiled in the Dalvik Executable (DEX) format. This file contains all the application code in the form of Dalvik bytecode and can be executed in the Dalvik VM [3].
- `resources.arsc`: pre-compiled resources. This file contains a dump of the resource table, which keeps track of the resources and their associated IDs and packages.

The manifest file is unique to the Android platform, and we describe it in more detail below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="edu.ucdavis.cs.android" android:versionCode="1" android:versionName="1.0">
  <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="16" />
  <permission android:name="edu.ucdavis.cs.android.PERMISSION" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-feature android:name="android.hardware.wifi" />
  <application android:icon="@drawable/icon" android:label="@string/appname">
    <activity android:name=".MainActivity" android:label="@string/main">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Figure 2.2: An example of Android manifest.

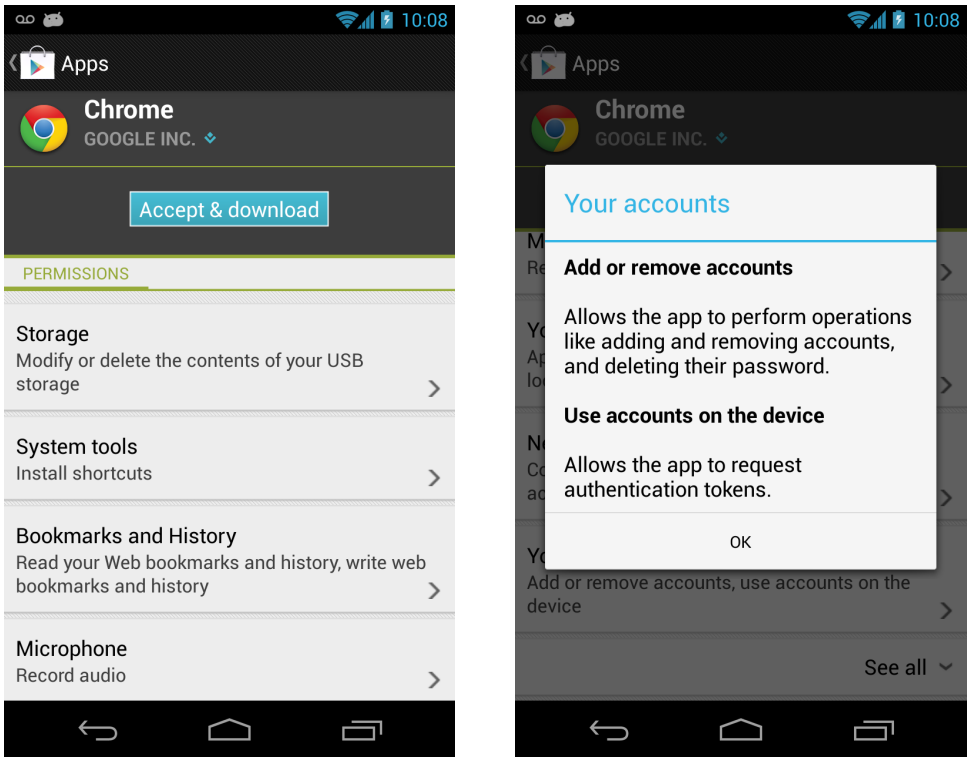
2.2 Android Manifest

The Android manifest file provides important information to the system about various aspects of an application [1]. A simple manifest file is given in Figure 2.2.

The manifest file defines the package name of the application, which serves as a unique identifier. It also describes the components of the application. Each application must declare in its manifest file the permissions required to access protected APIs and resources, and may optionally define its own permissions that other applications can use to interact with its components. The manifest also contains compatibility requirements of the application, such as the minimum API level (*i.e.* platform version), supported screen features (*e.g.* size and density), and hardware/software features (*e.g.* GPS and OpenGL version). Next, we describe Android's permission system and application components.

2.3 The Permission System

The Android system leverages Linux user-based protection and sandboxes applications at the OS level. Each application is assigned a unique UID and runs in a separate process. The



(a) List of requested permissions (b) Descriptions of requested permissions

Figure 2.3: Screenshots of requested permissions at install time in Google Play.

Android application sandbox isolates data and code execution on a per-application basis, so that buggy or malicious applications cannot adversely impact other applications or the system. As a notable exception, the framework also supports the *sharedUserId* feature, which allows multiple applications to run in the same process, provided that they are signed by the same certificate.

By default, Android applications are unprivileged and cannot perform sensitive operations or access sensitive data, such as users' address books and device identifiers. The Android platform provides an extensive set of APIs that are guarded by *permissions* for granting accesses to sensitive data and operations [13]. Each permission regulates the access to some specific sensitive data or privileged operations. An application must explicitly request permissions that it needs in its manifest. When installing an application, a user is presented with all the permissions that are requested by the application (Figure 2.3a) along with brief descriptions of the requested permissions (Figure 2.3b). To install the application, the user must explicitly accept all the

requested permissions. It is not allowed to grant a subset of the requested permissions. Once the application is installed, it cannot request any additional permissions, nor can the user revoke any already granted permissions, unless the application is updated or reinstalled. Since applications using the `sharedUserId` feature run in the same sandbox, they share each other's permissions.

Android 4.2, released in October 2012, defines 200 permissions, of which 130 have been documented [11]. Each permission is assigned a *protection level* indicating the level of potential risks associated with the permission. The platform defines these protection levels[1]:

- *Normal permissions* govern APIs that pose minimal risks to other applications, the system, or users. For instance, `VIBRATE` and `FLASHLIGHT` allow applications to control certain aspects of the hardware, which may be annoying but should not cause harm to users or the system. Such permissions are automatically granted by the system when applications are installed.

- *Dangerous permissions* regulate APIs that may be privacy-intrusive or impact the system adversely. For instance, placing phone calls and sending text messages are high-risk operations that may cost money, and require the `CALL_PHONE` and `SEND_SMS` permissions respectively. Such permissions will only be granted upon user's explicit approval.

- *Signature/signatureOrSystem permissions* protect the most sensitive APIs and resources. For instance, `INSTALL_PACKAGES` and `DELETE_PACKAGES` are required to install and uninstall applications. *Signature* permissions are only granted to applications signed with the device manufacturer's certificate. Similar to *signature* permissions, *signatureOrSystem* permissions are also protected, but it can be granted to applications in the Android system image. These permissions are intended for internal use by the system or system applications, and are normally not granted to third-party applications.

Each application may also define its own permissions to protect itself from unauthorized

[1]Two additional flags are introduced in Android 4.1: permissions with the *system* flag are only granted to applications installed on the system image; permissions with the *development* flag can also be granted to development applications.

access, but third-party applications are advised not to define permissions using the *signature* or *signatureOrSystem* levels.

2.4 Application Components

The Android framework defines four types of components [2]:

- *Activity* provides a screen that users can interact with in order to perform certain actions. An application usually consists of multiple activities, and each activity may start another activity to perform different actions.
- *Service* runs in the background without any user interface. It is often used to perform long-running tasks or work for remote processes.
- *Content provider* manages shared application data and provides standard interfaces to transfer data among applications.
- *Broadcast receiver* listens and responds to broadcast announcements from other applications and the system.

An application may have multiple components of each type. Most application components, such as activities, services, and broadcast receivers, can be activated using intents. An *intent* is a message that holds abstract descriptions of operations to be performed. An intent provides the runtime binding among components, and can be used for both intra- and inter- application communications [8]. There are two types of intents: *explicit intents* and *implicit intents*. Explicit intents designate target components by name and thus can be used to start a specific component. They are mostly used for intra-application messaging. Implicit intents, on the other hand, do not specify any designated components. When an implicit intent is received, the system tries to resolve it and decide which components can carry out the desired task. They are often used for inter-application communications.

To inform the system that it is only interested in certain implicit intent messages, a component can use an *intent filter*. When a component is declared in the manifest, intent filters can affect its visibility, either public or private. Public components can be accessed by other applications, while private ones are only accessible from within the same application. Activities, services and broadcast receivers are private by default, but the presence of intent filters may make them public. Content providers do not support intent filters and are public by default[2]. Applications can also limit access to its public components by imposing permission requirements.

2.5 Dalvik Executable

Although Android applications are written in Java, they are compiled and converted to Dalvik executables [4] that run in the Dalvik VM. The compact Dalvik executable (DEX) file format is designed specifically for systems with resource constraints. Unlike the stack-based Java VM, Dalvik VM is register-based. It is optimized for low memory configurations and designed to allow multiple VM instances to run efficiently. For reverse-engineering and analysis purposes, various tools have been developed to convert Dalvik bytecode back to Java bytecode [17, 81], or even to Java source code [77].

[2]Content providers are no longer exported by default if an application targets Android 4.2 or higher.

Chapter 3

A Systematic Study of Errors and Pitfalls in Android Applications

Following the widespread adoption of smart mobile devices, recent years have witnessed an explosion of mobile applications. However, we have yet to understand the unique characteristics of this important, emerging software ecosystem. This chapter presents the first large-scale study of third-party applications on Android, currently the most popular mobile platform. To this end, we have gathered and studied over 120K unique applications from the Google Play Store. We have systematically examined the manifest files, permission usage and bytecode of these applications, focusing on uncovering and understanding common errors and pitfalls. Among various findings, we have observed that 1) 50,785 applications contain potentially vulnerable components, and 862 applications use an advanced security feature (`sharedUserId`) that can mislead users and lead to security threats; 2) more than 14,600 applications use incorrect permissions, such as misspelled, non-existent, protected or deprecated ones, likely resulting in bugs or even crashes; and 3) more than 70% applications use Java reflection, and over 30% applications contain obfuscated code, making automatic analysis challenging. Many of the errors and pitfalls are also present in the most popular applications. The most notable example is an error in a top 100 application that results in a serious, easily exploitable vulnerability.

This chapter presents our detailed findings and discusses their implications. We believe that our study benefits the whole Android community, from improving platform and tool support to building better applications and raising user awareness.

3.1 Introduction

Recent innovations in mobile technology has given rise to a whole new class of software — applications created specifically for smart mobile devices. To gain insights into this emerging and vibrant software ecosystem, this chapter presents the first large-scale, systematic study of third-party Android applications. We have gathered a large corpus of 122,570 unique applications from the Google Play Store. With this wide range of Android applications, we aim to systematically investigate their characteristics by examining the manifest files, permission usage and application bytecode. While recent studies focus only on specific aspects of Android applications, we extensively analyze their unique characteristics from the coding and development perspective. To our knowledge, this is the largest, most comprehensive study of Android applications from Google's Android application market. We seek to uncover and understand common errors and pitfalls, and provide actionable recommendations where applicable. Our findings can benefit the whole Android community, from platform and tool developers to application developers.

Platform Developers Platform developers may gain insights into the usage of their platform. Our results may help Google prioritize development efforts, enhance the security of the Android platform, improve frequently-used APIs, and provide better documentation. For instance, our study has revealed that some applications use an advanced security feature called `sharedUserId`. When these applications are installed from Google Play, they may be implicitly granted additional permissions not presented to users. We believe that it is important for the platform to warn users of such security concerns and provide accurate permission usage information to help users make more informed decisions.

Tool Developers A comprehensive understanding of applications also enables tool developers to build better tool support, such as testing, analysis, and development environments, to assist application developers. Specifically, our analysis of the manifest files has uncovered many errors resulting from typos, which can be easily avoided with a spell checker. We have also observed heavy use of Java reflection and obfuscation in application bytecode, which indicates that it is important to address such challenges when developing analysis tools for Android applications.

Application Developers Our findings provide application developers useful knowledge to improve their applications. By studying the uncovered common programming errors and pitfalls, application developers can avoid similar mistakes. For example, we have found that many applications misuse permissions, such as non-existent or protected ones, due to scarce documentation. Our findings can educate developers and help them avoid such errors during development.

While we examine Android applications on a large scale and present qualitative and quantitative results, we also break down the statistics to show the importance and relevance of our findings on popular applications. In addition, we demonstrate the impact of reported errors by manually analyzing and testing applications from Google Play. Specifically, we have found that several errors lead to security vulnerabilities in top applications that have tens of millions of downloads. As a notable example, we have detected and confirmed a serious, easily exploitable vulnerability in the popular *Handcent SMS* application that allows attackers to retrieve users' private messages without any permission (Section 3.2.4).

Understanding Android applications on a scale as large as ours is difficult, and we have encountered a number of technical challenges during our study. For instance, we have detected various structural and semantic issues in the manifest files. Since there is no central bug database for all Android applications, we have manually identified and highlighted a few important and prevalent classes of flaws by studying the official Android documentation. However, the documentation lacks details, is inconsistent or outdated, a primary reason for many of the errors. We have devoted much time exploring online resources, such as message boards and mailing

lists, when trying to understand the root causes of the common errors. Occasionally, we had to examine the Android framework source code to understand implications of the detected flaws. In the process, we have also developed a number of automated tools to aid our study and analysis.

The remainder of this chapter is structured as follows. Section 3.2 describes our methodology, outlines the design of this study, and presents our detailed findings. We reveal common errors and pitfalls, and provide quantitative information to shed light on the prevalence of the detected flaws. We also investigate root causes of these problems, discuss their implications, and present recommendations for improving the platform and applications. Finally, we survey related work in Section 3.3 and conclude in Section 3.4.

3.2 Study Design and Results

This section explains our data collection process, describes our methodology, and presents our detailed study results.

3.2.1 Application Collection

A prerequisite for our study is a quality collection of real-world Android applications. To this end, we have gathered a large number of third-party Android applications. To build this corpus, we wrote a customized crawler that automatically downloads top and new applications in all categories from the Google Play Store. We collected 122,570 unique applications (identified by their package names) over a period of three months. Table 3.1 shows a summary of our application collection. “Top Apps” refers to those applications considered popular by Google, while “New Apps” refers to those just released or updated. Since the list of popular applications do not change as frequently as new applications, we have a larger number of new applications in our corpus.

Although we have only included free applications in our collection, we believe that they are representative of all applications (including paid ones). First, many applications adopt

Type	Category	Top Apps	New Apps
Application	Books & Reference	660	5,235
	Business	720	5,146
	Comics	960	2,202
	Communication	723	2,518
	Education	691	4,667
	Entertainment	687	11,025
	Finance	661	2,212
	Health & Fitness	662	2,038
	Libraries & Demo	678	1,484
	Lifestyle	715	5,368
	Media & Video	681	3,482
	Medical	671	827
	Music & Audio	770	8,744
	News & Magazines	647	3,811
	Personalization	678	5,462
	Photography	869	2,537
	Productivity	672	2,303
	Shopping	693	1,773
	Social	700	3,283
	Sports	722	3,885
	Tools	656	5,341
	Transportation	729	1,104
	Travel & Local	698	2,987
	Weather	662	481
Game	Arcade & Action	710	3,544
	Brain & Puzzle	745	4,162
	Cards & Casino	691	997
	Casual	710	2,988
	Racing	780	617
	Sports Games	761	945

Table 3.1: Applications retrieved from Google Play.

the freemium model for monetization, which means they are free to use but require in-app purchases to unlock premium features or contents. Second, we observe that a number of paid applications only serve as license keys while all the functionalities are already included in their free versions. In both cases, the free applications can also be viewed as paid ones and do not differ significantly in terms of quality. Popularity wise, we observe that free applications, especially the top applications, have much larger user bases. All these suggest that our collection

of applications should be representative, and we believe that our findings should be applicable in general.

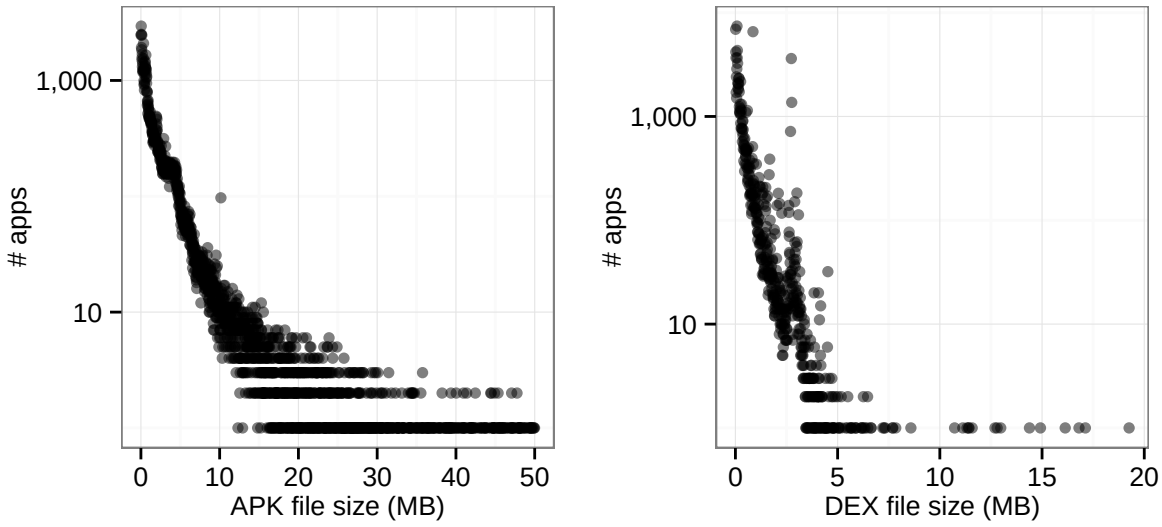
3.2.2 Methodology

Our goal is to understand Android applications in a systematic way. We have designed our study to focus on unique aspects of Android applications: manifest, permissions, and code.

Manifest The manifest file reveals important properties of an application. It also controls how an application behaves and interacts with the system and other applications. First, we extract the `AndroidManifest.xml` file from each application archive, *i.e.*, each APK file. The extracted XML file is in a binary format, so we decode it to its original form using existing tools [14, 15] and feed it to the manifest parser we developed. The manifest file, which is well documented [1], may contain a number of different elements organized in a well-defined structure. We examine the use of various elements as well as the structures to identify common errors.

Permissions The permission system is a unique and important feature of the Android platform. Android applications request permissions and define their own permissions in manifest files. We analyze how applications use permissions and discuss our detailed findings. We also uncover permission errors and pitfalls that can be attributed to developer confusion and lack of proper documentation.

Code The core of an Android application lies in its Dalvik executable. There exist tools that convert Dalvik bytecode to Java bytecode [17, 78, 81], to assembly-like code [14, 80], or even to Java source code [77]. In this study, we take a DEX file, convert it to Java bytecode, and then feed it to an efficient analyzer we built using the Soot [92] library. Our analyzer extracts APIs and other interesting code features. We examine the use of Java and Android library APIs, and then report and discuss our findings.



(a) APK file size distribution

(b) DEX file size distribution

Figure 3.1: Size distribution of Android applications.

Since the manifest and permission system are important and unique to the Android platform, our study focuses more on these aspects. Following the methodology described above, we present our results in the following sections. We begin with some general descriptive statistics on the applications studied. We then present our detailed findings on manifest files, permission usage, and application code, highlighting the common errors and pitfalls. We also discuss possible implications of our findings and provide actionable recommendations where applicable.

3.2.3 General Statistics

The 122,570 applications are published by 37,322 developers, with a maximum of 4,052 applications under a single developer, *ReverbNation Artists*, which is an application builder platform that helps musicians and artists create and publish their own mobile applications. Each developer has published around three applications on average.

Figure 3.1a shows the distribution of application package (APK) size. We can see that the largest applications are 50MB in size. This is because Google imposes a limit of 50MB on the size of APK files published on its Play Store [50]. Figure 3.1b shows the size distribution of

Type of Error	Top 100 Apps			Top 1,000 Apps			All Apps		
	#errors	#apps	#devs	#errors	#apps	#devs	#errors	#apps	#devs
Element misuse	8	8	6	144	77	65	15,514	5,108	2,441
Illegal element	2	2	2	14	13	13	1,136	911	409
Illegal attribute	81	28	26	593	193	170	21,136	11,004	5,271
Wrong parent element	9	9	8	47	44	41	5,960	4,310	2,420

Table 3.2: Common errors in AndroidManifest.xml.

DEX files, which hold application code. The largest dex file is around 20MB. The rest of an application's space is mostly taken by resources in its APK file.

3.2.4 Manifest

Manifest files serve a key role in Android applications. We have extracted structural rules from the official Android documentation and checked the manifest of each application for common errors, security concerns, and pitfalls.

Developer Errors

The structure of the manifest file is well defined and documented [1]. This file is intended primarily for internal use by the Android framework, and in some cases by Google Play for filtering purposes [7]. Therefore, it should follow the standard structural rules. Developers should not add their own elements or attributes in this file, as they will not be understood by the framework. We have found that a large number of applications contain errors or do not strictly follow the documented rules. Table 3.2 summarizes the common developer errors we found in the manifest files. For each type of error, we list the number of instances of the error, the number of applications having the error, and the number of developers involved. We also break down the statistics to show the relevance of reported errors in top applications. It is interesting to observe that the ratio of affected applications rises with their popularity. We speculate that this is because top applications tend to be more complex, which makes them more error-prone.

Element Misuse The Android documentation describes how elements should be used. For instance, each manifest file must contain *exactly* one `manifest` element and one `application` element. While no application misses any of these two elements, we have found that two applications contain duplicate `manifest` elements and 26 applications contain duplicate `application` elements. A more frequently misused element is `uses-sdk`, which defines the minimum API level (*i.e.*, `minSdkVersion`) required for the application to run, as well as the target API level (*i.e.*, `targetSdkVersion`) for which the application is intended. We found this element missing in 68 applications. When this happens, the system assumes a default value of 1 for the minimum API level, which means that the application is compatible with all Android versions. This leads to runtime crashes on an earlier version of the platform if the application is incompatible with that version. The values of both API levels should be integers[1], but we found string values, *e.g.*, "1.6" and "2.2", in two applications. We also found that a number of applications include multiple `uses-sdk` elements in their manifest files, and one of them has as many as four `uses-sdk` elements. In these cases, the SDK tools will not merge the attributes from different `uses-sdk` elements, which may lead to undefined behavior. We discovered through testing that the Play Store uses the lowest defined API level as the minimum API level for filtering, while the Android framework uses the last defined target API level on devices. Among the 1,829 applications with multiple `uses-sdk` elements, 435 of them define conflicting API levels while 655 split up the attributes across multiple elements. The latter case is likely due to developer confusion and has been mitigated by improved documentation.

Another frequently misused element is `intent-filter`. Intent filters are commonly used to advertise to the system the capabilities of application components. If an `intent-filter` element is found inside the component declaration, it filters unwanted implicit intents so that only those that pass through the filter are delivered to the component. The `intent-filter` element usually contains a few sub-elements, such as `action`, `category`, and `data`, to specify the filtering conditions. According to the documentation, at least one `action` must be present.

[1]Provisional API levels use strings, but applications with provisional API levels can only run in the emulator and cannot be published on Google Play.

Element	#typos	#apps	Example Typos
uses-permission	207	189	USES-PERMISSION, user-permission, uses_permission
uses-feature	183	93	users-feature, used-feature, use-feature
supports-screens	118	118	support-screens, supports-screen, supported-screens
receiver	111	111	Receiver, reciever, recevier
intent-filter	31	21	intent-filer, intent-fileter, itent-filter

Table 3.3: Most commonly misspelled element names.

Otherwise, no intent would be able to pass through the filter. We found 12,485 instances of intent filters without actions in 3,286 applications. These errors are likely due to the developers' lack of a clear understanding of intents and intent filters.

Illegal Element The manifest file should only contain supported elements, but we found 1,136 instances of illegal elements across 911 applications. After inspecting the illegal elements, we discovered that 725 of them are typos, and 215 of the typos are case-sensitivity errors. While all the element names should be spelled in lowercase, it is not stated clearly in the documentation. Careless developers may spell element names in uppercase or mixed-case, and such errors can be difficult to notice. The most commonly misspelled elements are listed in Table 3.3. Besides the typos, 411 illegal elements remain. We noticed that the most commonly used illegal element is `original-package`, which turns out to be an undocumented element. Some of the system applications, such as Launcher and Camera, use this element to handle package name changes across system updates. This element should not be used in third-party applications.

Illegal Attribute The manifest documentation defines a set of possible attributes for each supported element. We found 21,136 instances of illegal attributes in 11,004 applications. We divide these attributes into four categories: *misplaced*, *missing prefix*, *undocumented*, and *unknown*. Misplaced attributes refer to those unsupported by the enclosing element but are valid inside other elements. For instance, the `android:debuggable` attribute is only valid inside the `application` element, so it is misplaced if found within the `manifest` element. A total of 11,935 attributes belong to this category. Except some attributes of the `manifest`

Category	Attribute	#errors
Misplaced	android:screenOrientation	2,070
	android:debuggable	1,857
	android:launchMode	1,744
	android:configChanges	1,485
	android:required	1,008
Missing Prefix	xlargeScreens	983
	smallScreens	262
	anyDensity	259
	largeScreens	237
	normalScreens	211
Undocumented	android:allowClearUserData	941
	android:allowBackup	447
	android:testOnly	157
	android:finishOnCloseSystemDialogs	132
	android:restoreNeedsApplication	38
Unknown	class	886
	a:name	475
	android:layout_width	195
	android:layout_height	193
	ns0:name	149

Table 3.4: Most common illegal attributes.

element, all attribute names begin with the "`android:`" namespace prefix. We found that 3,222 attributes miss this prefix and would otherwise be valid. In addition, we examined the framework source code and found that some attributes are supported but undocumented or only partially documented, which means that they are missing on the manifest documentation page but are briefly referenced elsewhere. Some of these attributes are reserved for system applications and should not be used in third-party applications. They account for 1,750 illegal attributes. Finally, the remaining 4,229 are unknown attributes. Some of them may have been defined by developers, and some are typos. In either case, they are not supported by any of the legal elements. The most common illegal attributes are listed in Table 3.4.

Wrong Parent Element Some manifest elements may contain sub-elements as data. A common error is to place an element inside a wrong parent element. We found 5,960 instances

Misplaced Element	Wrong Parent	Correct Parent	#errors
uses-library	activity	application	845
action	activity	intent-filter	706
supports-screens	application	manifest	581
category	activity	intent-filter	389
meta-data	manifest	activity, etc.	381

Table 3.5: Common misplaced elements.

of such errors in 4,310 applications. Table 3.5 shows the most commonly misplaced elements, where they are misplaced, and where they should belong. We can see that developers tend to misplace sub-elements that contain global information about the application inside `manifest` and `application` elements.

Security Concerns

This section highlights two security concerns that we have observed in our study. We start with some general statistics of application components and continue to explain potentially vulnerable components. We also describe in detail the issues of a security feature called `sharedUserId` and discuss possible solutions.

The manifest file contains component declarations for applications, which is used by the system to locate and launch components when necessary. While most applications contain one or more components, we found that 250 applications in our corpus declare no components at all. These applications may not offer any functionalities on their own, but could act as add-ons or plug-ins that work with other applications. A common use case is resource packs (*e.g.*, language packs and theme packs) which contain only resources that can be loaded and used in other applications. Table 3.6 shows the number of applications containing each type of components. We also list the average number and the maximum number of components that each application uses. Note that we eliminate duplicate components when calculating the statistics. We can see that top applications generally contain more components. The statistics show that activity is the most frequently used type of components. Over 99% of the applications have at least one

Component		Top 100 Apps				Top 1,000 Apps				All Apps			
		#apps	Avg.	Max.	Std.	#apps	Avg.	Max.	Std.	#apps	Avg.	Max.	Std.
All	activity	100	23.12	110	21.16	993	18.18	242	21.87	121,870	11.55	779	15.08
	service	73	2.44	17	3.12	584	1.44	26	2.28	45,217	0.57	40	1.00
	provider	45	1.06	12	1.86	264	0.46	12	1.05	6,749	0.08	19	0.40
	receiver	77	3.11	23	4.01	642	2.09	47	3.31	45,940	0.90	47	1.60
Exported	activity	100	4.52	29	5.36	991	3.81	98	6.89	121,572	1.94	444	4.55
	service	24	0.42	4	0.86	208	0.37	25	1.29	13,610	0.15	25	0.51
	provider	41	0.84	12	1.57	221	0.37	12	0.94	5,700	0.06	18	0.36
	receiver	74	2.69	19	3.27	626	1.87	46	3.04	37,938	0.52	46	1.07
Unprotected	activity	100	4.30	29	4.86	991	3.74	98	6.80	121,561	1.94	444	4.55
	service	23	0.37	4	0.78	148	0.30	25	1.25	11,381	0.11	25	0.45
	provider	36	0.62	4	1.01	196	0.31	8	0.78	5,473	0.06	18	0.35
	receiver	73	2.41	19	3.04	612	1.70	46	2.93	34,738	0.43	46	0.96
Potentially Vulnerable	activity	68	3.13	28	4.72	505	2.67	97	6.70	32,931	0.93	443	4.50
	service	14	0.17	2	0.45	108	0.19	25	1.01	10,326	0.10	25	0.40
	provider	34	0.55	4	0.93	189	0.30	8	0.77	5,005	0.06	18	0.34
	receiver	49	1.17	13	2.03	323	0.62	37	1.68	15,502	0.15	37	0.49

Table 3.6: Application component statistics.

activity, and applications have more than 11 activities on average. Content provider is the least popular type of components only found in 5.5% of the applications.

We observe that application components are often exported and unprotected. In general, application components can be either private or public (*i.e.*, exported). Private components are intended for internal use within the same application, while exported components can be launched by other applications. Table 3.6 shows the number of applications with exported components. We can see that over 99% of the applications have at least one public activity. When a component is exported, the developer may protect it with a permission such that only entities with that permission can access the component. The third section of Table 3.6 shows the number of applications containing exported components that are not guarded by permissions. We can see that the majority of the exported components are unprotected.

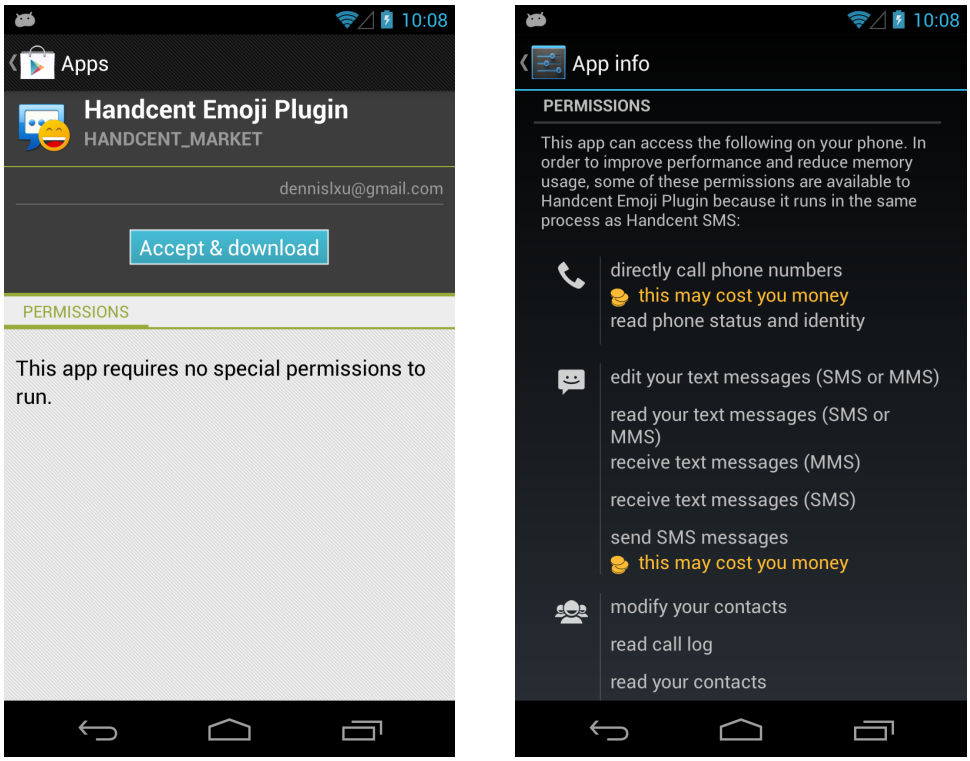
Now we take a closer look at unprotected components. In some cases, developers may intentionally make a component publicly accessible. For instance, most applications have an exported main activity that serves as an entry to the application. However, developers may also inadvertently make components public. As described in Section 2.4, most components are private by default, but they can be implicitly made public by including an intent filter. Developers unaware of this implication may accidentally export a component and expose it to security

risks [33]. In general, we consider a component *potentially vulnerable* if it is implicitly exported and unprotected, with a few exceptions for activity and receiver components. For activities, we exclude applications' main activities, as they are intended to be public and unprotected. We also exclude two types of broadcast receivers: those that only receive protected system broadcast messages and those that correspond to App Widget declarations. In both cases, the receivers typically use intent filters to subscribe to specific broadcast messages sent by the system or the App Widget host (*i.e.*, the launcher). The statistics for potentially vulnerable components are presented in the last section of Table 3.6. We found a total of 151,753 potentially vulnerable components in 50,785 applications. We also observe that top applications have a higher number of potentially vulnerable components, likely because they are more complex and contain more components in general.

To understand the security impact on real-world applications, we manually analyzed a few potentially vulnerable components in the top applications. We have identified and confirmed a severe security vulnerability in the popular *Handcent SMS* application, which has 10 to 50 million downloads in Google Play. Among various features, it advertises additional security options that allow users to password protect and hide messages in a privacy box. However, our analysis reveals that the contents of this privacy box are exposed by a public unprotected content provider. We believe that the developer intended to make this component private by including the `exported` attribute and setting it to `false` in the component declaration. Unfortunately, this attribute is not recognized by the platform as it is missing the namespace prefix (*i.e.*, `android:exported`), which we discussed in the previous section. This vulnerability poses serious threats to user privacy as it exposes the information that users want to protect the most. To make matters even worse, it is easily exploitable and we crafted a proof-of-concept application that retrieves contents of the privacy box without a single permission. Similarly, we have also detected and manually confirmed vulnerabilities in public components from the *WeChat* and *Sina Weibo* applications. Both services are very popular with over 300 million users and their Android clients are also among the top applications. We have notified the developers of these applications, but as of May 2013, the vulnerabilities still exist in their latest releases.

From our analysis of the manifest files, we observed that some applications use the `sharedUserId` feature, which may cause security threats to users. The permission mechanism is enforced at the process level, and each application runs in its own process by default. However, the Android platform supports the `sharedUserId` feature to allow multiple applications to share the same permissions by running in the same process. To take advantage of this feature, each of these applications needs to specify the same user ID in its manifest file and they must also be signed with the same signature to retain security. For instance, if two applications from the same developer use this `sharedUserId` feature, any permission granted to one application is also automatically granted to the other. However, most users do not know this feature or understand its security implications. Worse even, when installing applications that use `sharedUserId`, the Google Play Store does not provide any visual indication and fails to inform the user of potential risks. Figure 3.2 illustrates this security concern using the *Handcent Emoji Plugin* application. Figure 3.2a shows the screen presented to the user when he tries to install the application from the Google Play Store, which indicates that it does not require any permissions. Since this application uses `sharedUserId`, it is implicitly granted permissions requested by another application, *Handcent SMS* in this case, which is already installed on the device. However, if the user opens the application information page in system settings, all the granted permissions are listed (Figure 3.2b). This constitutes a security concern because users are usually more cautious with granting permissions before installing an application and less likely to check permissions after the application has been installed.

We found that 862 applications use `sharedUserId`. To better understand how this feature is used, we inspected all 862 identified applications. Among them, 297 applications do not share the same user ID with any other application in our corpus, and the most commonly shared user ID is found in 56 applications. In most cases, the applications having the same user ID are published by the same developer. We notice that a common use case for this feature is for add-on and plug-in applications that work with a main application. In this case, the add-on applications typically require few or no permissions but would be granted a number of permissions if the main application is installed.



(a) Requested permissions presented on Google Play

(b) Granted permissions listed in App info

Figure 3.2: Screenshots of permissions for an application using `sharedUserId`.

We found an interesting case where two applications with the same `sharedUserId` are signed with different signatures and published by different developers. One of the applications is a popular Sudoku game with hundreds of positive user ratings while the other is a new application that uses a similar name and has no user rating at all. A closer look reveals that the latter application requests a number of suspicious permissions (*e.g.*, `READ_CONTACTS` and `SEND_SMS`) that the popular one does not. We speculate that it is very likely a malware; it is no longer available on Google Play as of this writing.

Common Pitfalls

Besides developer errors and security concerns, we have observed a number of common pitfalls while analyzing manifest files. These issues usually do not affect an application's functionality

Type of Issue	Top 100 Apps			Top 1,000 Apps			All Apps		
	#issues	#apps	#devs	#issues	#apps	#devs	#issues	#apps	#devs
Wrong element order	824	76	64	6,579	806	597	509,025	84,743	25,642
Duplicate activity	8	5	5	101	78	72	13,909	12,518	1,675
Hardcoded debug mode	5	5	5	57	57	56	8,662	8,662	3,655

Table 3.7: Common issues in AndroidManifest.xml.

but may lead to subtle bugs or undesired behavior. Table 3.7 summarizes the common pitfalls found in the manifest files, with a breakdown on top applications.

Wrong Element Order While the order of elements at the same level does not matter in general, it is suggested that the `application` tag should appear after the elements that declare the requested API level, hardware/software features, permissions, *etc*. Subtle bugs may occur if the logical dependency order is not respected. We found 509,025 instances of this issue across 84,743 applications, indicating that almost 70% of the applications violate this rule. This is by far the most prevalent issue in the manifest files, which may be attributed to the fact that this suggested order was not documented in the past and bugs are triggered rarely. However, we believe application developers should follow this rule to avoid introducing bugs in the future.

Duplicate Activity We observed that some activity components are declared multiple times in the manifest. Each activity component should be declared in the manifest only once. If it is accidentally registered multiple times, the attributes are not merged, which can lead to subtle errors. We found 13,909 duplicate activity declarations in 12,518 applications. In one application, there are 779 activities, 316 of which are declared more than once in its manifest. We investigated applications containing duplicate activity components and found that a majority of such issues can be attributed to bugs in popular application builder tools. For instance, we observed that 6,315 applications built with *Appsbar* declare `StickersBackgrActivity` twice and *ReverbNation* is responsible for duplicate `BlogActivity` in 3,590 applications.

Hardcoded Debug Mode During application development and testing, developers can enable debug mode on the applications by setting the value of the `android:debuggable` attribute

to `true` in the manifest. This is a useful feature, but developers may forget to turn it off and accidentally publish their applications with debugging information. We found 8,662 published applications with the debug mode enabled. It is a recommended practice to omit the `android:debuggable` attribute from the manifest, and the ADT plugin [6] will automatically set its value according to the application's build type — debug or release.

Implications and Recommendations

Our study uncovered common developer errors and pitfalls in manifest files from a large number of applications. Many instances of these issues can be attributed to developer confusion and lack of quality documentation. We have observed improved documentation in the past months, and Google released Android Lint [9], a static analysis tool, to help application developers detect errors, such as misused elements and duplicate activity components, in the manifest files. We believe that this is a step in the right direction and recommend implementing additional checks in Android Lint, such as those for misspellings and structural issues.

For developers, we suggest reading the documentation carefully to better understand the platform and take advantage of the Lint tool to avoid common errors. It is also important to keep security concerns in mind during application development and testing. For instance, developers need to understand security implications of using intent filters, as inadvertently exposing components may render the application vulnerable.

We found that the `sharedUserId` feature may lead to user confusion and security threats. Specifically, users are not informed of additional permissions that applications may inherit from other applications at the time of install. Similarly, when installing a new application, Google Play fails to warn users that additional permissions may be granted retroactively to already installed applications. As a first step to mitigate this issue, we recommend displaying warning messages to users when they try to install any application that use `sharedUserId`. One possible solution is to accurately display all permissions granted, both explicitly and implicitly, by scanning all installed applications, whenever a new application using `sharedUserId` is being installed. To

avoid confusing users, the UI also needs to be better designed and organized to help users make informed decisions.

3.2.5 Permissions

Applications need to request permissions to access protected APIs and resources. We have observed that most applications require at least one permission, while only 7,784 applications request no permission at all. On average, each application requests 5.75 permissions, with a standard deviation of 4.37. We found that the most permission-hungry application, *Brick Breaking*, uses as many as 122 permissions. Surprisingly, it requests almost all documented permissions, including protected ones, in its manifest. This application implements a simple live wallpaper and we found no sign of malicious behavior. We hypothesize that the developer was either being lazy by declaring all the permissions or planning to turn it into malware later through updates. This application has been removed from Google Play as of this writing. As expected, we also observe that popular applications tend to use more permissions. The top 100 applications use 11 permissions on average, with a standard deviation of 8.43.

Table 3.8 shows the top 10 most requested system permissions defined by the Android framework. We can see that most applications request network related permissions including `INTERNET` and `ACCESS_NETWORK_STATE`. Table 3.9 lists the most commonly used non-system permissions defined by applications. Among these permissions, `RECEIVE` and `BILLING` are required to use "Google Cloud Messaging" (formerly "Cloud to Device Messaging") and in-app billing services respectively, both of which are extended Google APIs. Other popular non-system permissions are defined by the launcher application to allow modifying shortcut icons on the home screen (`INSTALL_SHORTCUT` and `UNINSTALL_SHORTCUT`) and regular access to launcher settings (`READ_SETTINGS`).

The Android platform organizes permissions in groups. Related permissions that belong to the same group are presented together in the UI. Figure 3.3 shows how frequent permissions in each group are used in applications. We can see that `NETWORK` is the most frequently used

Permission*	Protection Level	%
INTERNET	dangerous	90.25%
ACCESS_NETWORK_STATE	normal	72.90%
WRITE_EXTERNAL_STORAGE	dangerous	51.44%
READ_PHONE_STATE	dangerous	47.81%
ACCESS_FINE_LOCATION	dangerous	32.48%
VIBRATE	normal	28.23%
ACCESS_COARSE_LOCATION	dangerous	28.12%
WAKE_LOCK	dangerous	23.74%
RECEIVE_BOOT_COMPLETED	normal	20.13%
CALL_PHONE	dangerous	13.65%

Table 3.8: Top 10 most requested system permissions.

*All listed system permissions share the `android.permission.` prefix, which is omitted in this table.

Permission	#apps
com.google.android.c2dm.permission.RECEIVE	10,620
com.android.launcher.permission.INSTALL_SHORTCUT	5,494
com.android.vending.BILLING	2,894
com.android.launcher.permission.UNINSTALL_SHORTCUT	1,773
com.android.launcher.permission.READ_SETTINGS	467

Table 3.9: Most requested non-system permissions.

permission group, which is consistent with the top permission statistics presented in Table 3.8. The two top permissions, `INTERNET` and `ACCESS_NETWORK_STATE`, both belong to the `NETWORK` permission group. Our inspection of permission usage leads to a number of findings. In particular, we have found that developers often misuse permissions, likely resulting in bugs or even crashes.

Developer Errors

A common developer error is misspelling system-defined permissions. We found that 809 applications contain misspelled system permissions in their manifest files. Table 3.10 shows the most commonly misspelled permissions. When requesting a permission, the name of the permission must be exactly the same as defined by the platform. All predefined permission

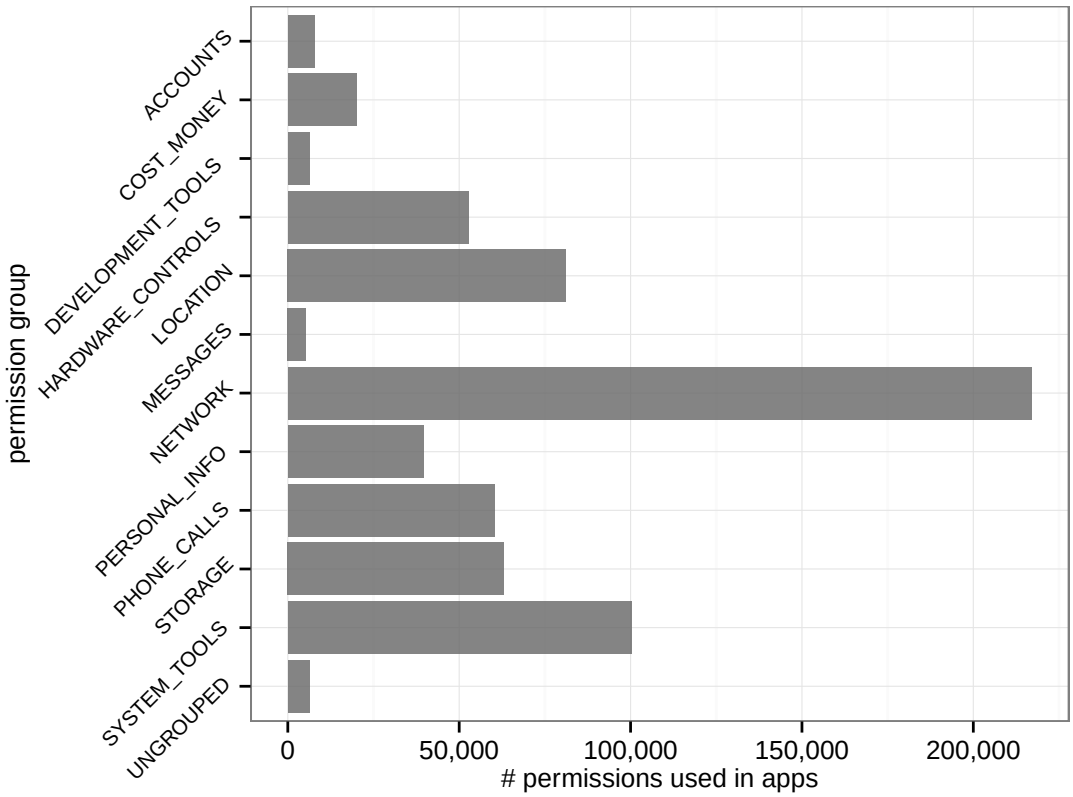


Figure 3.3: Requested permissions in each group.

names are case-sensitive and never contain spaces. Permission misspellings are sometimes subtle to notice (*e.g.*, case-sensitivity errors and superfluous whitespace), but such errors can be easily avoided with the help of a spell checker.

Requesting non-existent permissions is another common permission-related developer error. We have noticed that some applications request permissions that are likely non-existent. For instance, we found that 390 applications request the `RECORD_VIDEO` permission, which is not defined by the system and is also unlikely to be defined by third-party applications. The system does define the `RECORD_AUDIO` permission, so developers might mistakenly assume that they need a corresponding `RECORD_VIDEO` permission to capture video. Similarly, we found that 2,701 applications request the `READ_EXTERNAL_STORAGE` permission, which was introduced in Android 4.1. All the applications in our collection that request this permission were published

Category	Permission*	#apps
Misspelled	RECEIVE_BOOT_COMPLETED	397
	ACCESS_COARSE_LOCATION	250
	CALL_PHONE	61
	SET_WALLPAPER_HINTS	30
	ACCESS_FINE_LOCATION	27
Unknown	READ_EXTERNAL_STORAGE	2,701
	WRITE	2,060
	PLUGIN†	933
	RECORD_VIDEO	390
	READ_SETTINGS	349
Protected	READ_LOGS	3,320
	INSTALL_PACKAGES	1,987
	DELETE_PACKAGES	787
	MODIFY_PHONE_STATE	608
	INJECT_EVENTS	546
Deprecated	READ_OWNER_DATA	586
	WRITE_OWNER_DATA	522
	ACCESS_GPS	518
	ACCESS_LOCATION	426
	RAISED_THREAD_PRIORITY	242

Table 3.10: Common permission errors.

*Permission prefixes are omitted. The `PLUGIN` permission has the `android.webkit.permission.` prefix and all other permissions have the `android.permission.` prefix.

†The `PLUGIN` permission is defined by WebKit and is only meant to be used by the Adobe Flash browser plugin. Third-party applications cannot use this permission.

before this permission is supported. There is a `WRITE_EXTERNAL_STORAGE` permission, which was added in Android 1.6. Therefore, it is likely that developers mistakenly assumed that they need the `READ_EXTERNAL_STORAGE` permission to read from external storage (*e.g.*, the SD card). In addition, we have observed that over 100 applications (including top ones) request the `ACCESS_COARSE_UPDATES` permission, which is not defined by the Android platform but referenced in the documentation. Our inspection of the platform source code and documentation led us to believe that this is a documentation error. We have also found a bug report on this issue filed in August 2011 in the public Android Open Source Project issue tracker [5]. However, the documentation error remains to be fixed.

Our permission usage study also reveals that a number of applications request protected permissions. As discussed in Section 2.3, permissions with the signature or signatureOrSystem protection level are intended for system applications. We found that 8,457 applications request at least one such permission. Table 3.10 also lists the most requested protected permissions. When a third-party application tries to obtain such permissions, the system silently denies the request, unless the application is signed by the device manufacturer. We believe that most of these permissions are requested in error, and thus may lead to runtime crashes. We note that the protection level for permissions may change across API levels. For instance, `READ_LOGS` used to be a dangerous permission, but it has recently been promoted to the signatureOrSystem level in Android 4.1 to address potential security risks [65]. This change broke some applications that legitimately use the permission to collect and display system log data.

Common Pitfalls

Developers sometimes request deprecated permissions. We found that 2,578 applications request system permissions that are deprecated. For instance, the `READ_OWNER_DATA` and `WRITE_OWNER_DATA` permissions in Table 3.10 were defined in an earlier version of the SDK. They were never used by the platform, and thus have been removed in recent versions of the SDK. Other commonly used deprecated permissions include `ACCESS_LOCATION` and `ACCESS_GPS`, which were defined in pre-release versions of the platform and were deprecated in 2008. Developers likely included these deprecated permissions because they forgot to remove them or they were misled by outdated materials. We observed that 580 of the 584 applications that use either `ACCESS_LOCATION` or `ACCESS_GPS` also request the updated permissions, `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`.

Implications and Recommendations

We found that certain permissions (*e.g.*, `INTERNET`) are requested by an overwhelming number of applications. We hypothesize that many applications do not need such permissions for

their core functionalities. For instance, it is likely that many free applications request the `INTERNET` permission with the sole purpose of serving ads, which is a strong indication that some permissions are too coarse-grained. This finding is consistent with previous studies suggesting that the current permission mechanism in Android should be more fine-grained [47, 61]. We believe that it is worthwhile to split such permissions into multiple permissions to provide stronger privacy and security guarantees.

Our study revealed common errors and issues on permission usage. We believe that better tool support can help developers avoid many such errors. Developer confusion over permission names, protected permissions, and deprecated permissions can be addressed by improved API documentation. For instance, protection levels are currently undocumented for system permissions, and developers need to examine the framework source code to find out this information. To mitigate this issue, the latest version of Android Lint now checks for protected permissions in manifest files. We recommend including a spell checker to detect typos in permission names and implementing additional checks for non-existent and deprecated permissions.

3.2.6 Code

Since Android applications are mostly written in Java, the SDK supports a fairly large subset of the Java SE library. It also includes a large set of platform-specific APIs that expose various functionalities of the system to third-party applications. Table 3.11 and Table 3.12 lists the most popular Android and Java library APIs respectively, and Table 3.13 shows the most frequently used library classes. We can see that the most frequently used Android APIs are related to accessing UI elements and logging, while the most popular Java APIs are for string and container manipulation. We have also observed that many invocations of string related methods (*e.g.*, in the `StringBuilder` class) are likely generated by the compiler. We use bytecode analysis to understand the characteristics of application code and present our findings next. In particular, we have found that both Java reflection and obfuscation techniques are widely used among

API Signature	#	%
<android.app.Activity: android.view.View findViewById(int)>	5,470,941	2.45%
<android.widget.TextView: void setText(java.lang.CharSequence)>	4,639,965	2.08%
<android.view.ViewGroup: void addView(android.view.View)>	3,807,229	1.71%
<android.view.View: void setVisibility(int)>	3,471,128	1.56%
<android.util.Log: int d(java.lang.String,java.lang.String)>	3,398,838	1.52%
<android.view.View: void setLayoutParams(android.view.ViewGroup$LayoutParams)>	3,232,688	1.45%
<android.view.View: void setOnClickListener(android.view.View$OnClickListener)>	2,948,891	1.32%
<android.content.Intent: android.content.Intent putExtra(java.lang.String,java.lang.String)>	2,866,169	1.29%
<android.view.View: android.view.View findViewById(int)>	2,849,131	1.28%
<android.util.Log: int e(java.lang.String,java.lang.String)>	2,332,733	1.05%
<android.content.Context: java.lang.String getString(int)>	2,316,965	1.04%
<android.content.Intent: void <init>(android.content.Context,java.lang.Class)>	2,243,321	1.01%
<android.util.Log: int i(java.lang.String,java.lang.String)>	2,106,091	0.94%
<android.content.ContextWrapper: android.content.res.Resources getResources()>	2,064,183	0.93%
<android.app.Activity: void startActivity(android.content.Intent)>	2,017,514	0.91%
<android.view.View: android.content.Context getContext()>	1,875,192	0.84%
<android.content.Intent: java.lang.String getStringExtra(java.lang.String)>	1,788,578	0.80%
<android.content.ContextWrapper: android.content.Context getApplicationContext()>	1,707,885	0.77%
<android.app.Activity: void finish()>	1,701,531	0.76%
<android.widget.LinearLayout$LayoutParams: void <init>(int,int)>	1,699,667	0.76%

Table 3.11: Most frequently used Android library APIs.

API Signature	#	%
<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>	72,908,694	12.51%
<java.lang.StringBuilder: java.lang.String toString()>	31,262,497	5.36%
<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>	27,725,950	4.76%
<java.lang.Object: void <init>()>	24,023,819	4.12%
<java.lang.StringBuilder: void <init>()>	21,225,653	3.64%
<java.lang.String: boolean equals(java.lang.Object)>	18,609,356	3.19%
<java.lang.Integer: java.lang.Integer valueOf(int)>	10,611,595	1.82%
<java.lang.StringBuilder: void <init>(java.lang.String)>	9,411,002	1.61%
<java.lang.StringBuffer: java.lang.String toString()>	8,407,835	1.44%
<java.lang.StringBuffer: void <init>()>	7,637,498	1.31%
<java.lang.String: int length()>	7,468,616	1.28%
<java.util.HashMap: java.lang.Object put(java.lang.Object,java.lang.Object)>	7,002,028	1.20%
<java.util.Iterator: boolean hasNext()>	6,720,965	1.15%
<java.lang.StringBuilder: java.lang.StringBuilder append(int)>	6,714,107	1.15%
<java.util.Iterator: java.lang.Object next()>	6,666,693	1.14%
<java.lang.Throwable: void printStackTrace()>	6,378,071	1.09%
<java.util.Map: java.lang.Object put(java.lang.Object,java.lang.Object)>	5,514,263	0.95%
<java.lang.StringBuffer: java.lang.StringBuffer append(char)>	5,282,624	0.91%
<java.util.List: boolean add(java.lang.Object)>	4,803,839	0.82%
<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.Object)>	4,736,095	0.81%

Table 3.12: Most frequently used Java library APIs.

Rank	Android Class	Java Class
1	android.view.View	java.lang.StringBuilder
2	android.app.Activity	java.lang.String
3	android.content.Intent	java.lang.StringBuffer
4	android.util.Log	java.lang.Object
5	android.widget.TextView	java.lang.Integer
6	android.content.Context	java.util.ArrayList
7	android.view.ViewGroup	java.util.HashMap
8	android.database.Cursor	java.util.List
9	android.content.ContextWrapper	java.util.Iterator
10	android.os.Bundle	org.json.JSONObject
11	android.app.AlertDialog$Builder	java.util.Map
12	android.widget.ImageView	java.lang.Throwable
13	android.webkit.WebView	java.lang.Class
14	android.os.Handler	java.lang.Enum
15	android.content.res.Resources	java.io.File
16	android.content.SharedPreferences	java.lang.System
17	android.net.Uri	java.lang.IllegalArgumentException
18	android.content.SharedPreferences$Editor	org.json.JSONArray
19	android.content.ContentValues	java.lang.Thread
20	android.app.Dialog	java.lang.Long

Table 3.13: Most frequently used library classes.

Android applications, making it challenging to develop automated analysis tools.

Developer Errors

We observed that a large number of applications use API methods that are not supported in all targeted API versions. In general, application developers would like to take advantage of APIs introduced in newer versions of the system, while targeting the largest user base by making their applications backward-compatible with earlier system versions. However, if the developer makes a mistake and calls a new API that does not exist in an older system, the application may crash. Ideally, an application should only access methods introduced in API levels no more than the minimum API level declared in its manifest. To access new APIs, developers should use Java reflection or conditional execution and lazy loading to ensure that each API call is only executed on supported platforms [82]. To locate API calls that are not supported in all targeted

API versions, we extract all the API calls from each application, map each to its starting API level, and compare it to the minimum API level declared in the manifest. We found that 48,366 applications (39.46%) contain API calls not available on their supported platform versions. This may include cases where new APIs are guaranteed to be accessed only on supported platforms through conditional execution. In either case, these applications should be carefully vetted to ensure that they only execute supported API calls at run time.

Observations and Insights

Java Reflection New APIs are constantly added to each new Android SDK release. To use the new APIs while staying backward-compatible, developers frequently use Java reflection to target a wide range of platform versions. It is a common practice to invoke new APIs when they are available and fall back to earlier alternatives (or disable the affected functionality altogether) when necessary [71]. Reflection is also used to invoke hidden or private framework APIs. By looking for reflection-related APIs like `Class.forName` and `Method.invoke`, we found that 86,210 applications (70.34%) use Java reflection to make API calls. While reflection is a powerful technique that provides the flexibility to inspect and determine API characteristics at run time, it also incurs performance overhead. Therefore, developers should be judicious and avoid using it in loops and UI intensive tasks.

Obfuscated Code Obfuscation is often used in Android applications for two reasons: 1) paid applications are usually obfuscated to help defend against cracking and piracy; and 2) applications in general use obfuscation to protect intellectual property by making the code more difficult to reverse engineer. The Android SDK ships with an obfuscation tool, ProGuard [16], which Google encourages developers to use to obfuscate applications that integrate their licensing service [62]. Under the default configuration, ProGuard shrinks an application by removing unused code and obfuscates the code by renaming classes, fields and methods using semantically obscure names. We found that 38,285 applications (31.24%) are obfuscated, and many more applications contain obfuscated third-party library code. We have also observed that

top applications are more likely to be obfuscated. Specifically, 53 out of the top 100 applications have been obfuscated.

Native Code With the help of the NDK toolset, developers may reuse existing code libraries written in C or C++. However, using native code is discouraged for most applications as it brings no noticeable performance improvement but increases code complexity [12]. We found that 6,365 applications (5.19%) include and load libraries written in native code. Among these, 2,301 are games. We suspect that these games are originally written in native code and ported to the Android platform, or they take advantage of native code for better performance.

Root Access We have observed that applications occasionally request root access at run time for additional functionalities such as overclocking the processor. Such functionalities only work on devices that are hacked to allow superuser permissions [84]. Even a number of popular applications published on Google Play request root access. We found that 2,945 applications (2.40%) attempt to run the `su` command during execution. When this happens, the user is prompted and needs to decide whether or not to grant root access.

External Libraries Many developers utilize external libraries to facilitate development and enrich the functionalities of their applications. For instance, a lot of applications use ad libraries to display advertisements for monetization. To integrate with popular social network services, applications often include APIs from other platforms like Facebook and Twitter. In addition, since Android applications are written in Java, developers may also take advantage of existing open source Java libraries (*e.g.*, Apache Commons) to ease development. We have used some heuristics based on package names to detect external libraries and found that over 100,000 applications include at least one external library. Applications use 4.61 external libraries on average, and some even include as many as 50 external libraries. Table 3.14 shows the most commonly used external libraries, and more data on popular libraries can be found in Appendix A.

Library	Description	#apps
com.google.ads	AdSense/AdMob	39,462
com.google.android.maps	Google Maps API	23,345
org.apache.commons.logging	Apache Commons Logging	19,762
com.facebook.android	Facebook API	19,730
org.slf4j	SLF4J (logging)	17,718
twitter4j	Twitter API	17,020
org.apache.log4j	Apache log4j	16,680
org.apache.commons.codec	Apache Commons Codec	16,538
com.admob.android.ads	AdMob	11,125
com.google.android.apps.analytics	Google Analytics	10,762

Table 3.14: Most commonly used external libraries.

Implications and Recommendations

Our study identified frequently used library classes and methods. We believe that the API usage findings provide important feedback to the platform provider, and can be used to prioritize efforts on improving, optimizing, and fixing bugs in heavily used framework library code.

From the application development perspective, we found that developers often fail to take enough precautions in using new APIs while targeting multiple platform versions. Although the documentation contains necessary API level information to assist developers, it is still tedious and error-prone to ensure that only supported APIs are used across all platform versions. We believe that automated tools can help developers avoid such errors during development. For instance, recent versions of the Android Lint tool which includes an API level check can be used to detect accidental calls to newer APIs [10].

Our study finds that Java reflection is heavily used in Android applications. Therefore, it is important to handle reflection when developing analysis tools. While statically resolving reflective calls is very challenging in general, it is possible to take advantage of domain-specific knowledge for better resolution results in the Android setting [45].

Besides Java reflection, we also observed that obfuscation is commonly used. This indicates that it is important to consider obfuscation even when developing tools not intended for malware detection and analysis.

During our analysis of the application code, we found that some third-party libraries, especially advertising and analytics libraries, are included and shared in a large number of applications. Therefore, we recommend analyzing these popular libraries, summarizing and capturing the results, and reusing them in analysis tools to reduce analysis cost.

3.3 Related Work

Our study touches upon several active research areas, and we survey related work in this section.

Android Application Studies Researchers have performed various studies on Android applications to understand the emerging Android ecosystem. Bohmer *et al.* [23] propose AppSensor for analyzing application usage on smartphones and present both basic and contextual descriptive statistics. To help application developers monitor the performance of their applications, Ravindranath *et al.* [83] propose a lightweight system, AppInsight, that instruments application binaries to collect trace data and pinpoints critical paths in user transactions. ProfileDroid [94] offers multi-layer monitoring and profiling to capture characteristics of application specification, user activities, OS and network statistics. Crussell *et al.* [36] propose a static clone detection tool, DNADroid, and perform plagiarism detection on 75,000 Android applications. To enforce security policies on Android applications, Davis *et al.* [37] propose I-ARM-Droid, a bytecode rewriting framework for in-app reference monitors. Compared to previous studies, our work is the first large-scale study that promotes a systematic understanding from the coding and development perspective, with a unique focus on common errors and pitfalls. It also includes the largest number of unique applications from Google Play, making our findings representative and general.

Android Permission System Android adopts a decentralized approach to its permission model, and delegates permission decisions to Android users. A number of approaches have been proposed to enhance the Android permission system. Based on automated testing, Stowaway [45] constructs a mapping between API calls and permissions to detect overprivilege

in Android applications. Felt *et al.* [48] present permission re-delegation vulnerabilities which allow malicious applications to access privileged data and operations through confused deputies [56]. Dietz *et al.* propose QUIRE [38] to mitigate confused deputy attacks by extending the Android framework to allow applications to reason about the call-chain and data provenance of requests. Barrera *et al.* [21] propose a methodology based on self-organizing maps to visualize permission usage in 1,100 Android applications, and discuss the strengths and weaknesses of the Android permission model. Wei *et al.* [93] perform a long-term study on the permission evolution and usage in the entire Android ecosystem, from the platform to various applications. Our study, on a much larger scale, discusses in general how permissions are used and misused in Android applications, aiming to uncover common developer errors and pitfalls.

Application Security and Privacy There has been extensive research on the security and privacy of Android applications. Enck *et al.* [42] examine Android application security by studying 1,100 Android applications. They detected pervasive misuse of personal/phone identifiers and deep penetration of advertising and analytics networks. Their earlier work Kirin [40] performs lightweight certification of Android applications to defend against malware at install time. Chia *et al.* [32] focus on the effectiveness of rating systems in the application markets to measure privacy and security risks of third-party applications. RiskRanker [54] assesses potential security risks from untrusted applications and detects dangerous behaviors to spot zero-day malicious applications. DroidRanger [98] uses permission-based behavioral footprinting as well as heuristics-based filtering to detect malicious applications in official and alternative android markets. DroidMOSS [95] uses fuzzy hashing to detect repackaged (and potentially malicious) applications in third-party Android marketplaces. Bugiel *et al.* [26] propose system-centric and policy-driven runtime communication monitoring to protect against confused deputy and collusion attacks. Woodpecker [53] detects capability leaks in popular stock Android smartphones, where privileged permissions are unsafely exposed to other applications. Similarly, Chan *et al.* propose DroidChecker [31] that searches for the same class of vulnerabilities in third-party Android applications. To detect privacy leaks in Android

applications, TaintDroid [41] performs system-wide dynamic taint tracking and warns the user when sensitive information is exfiltrated through the network. To achieve similar goals, AndroidLeaks [52], a static analysis framework, detects privacy information leaked over the network on a collection of 24,350 Android applications. Stevens *et al.* [91] also investigate privacy leaks in Android applications, but with a focus on ad libraries. Again, in contrast to these security-focused analyses, ours aims at a systematic understanding of Android applications from all key aspects and leads to a broader range of new findings that can benefit the whole Android community.

Android Lint The Android SDK provides a static analysis tool called *lint* [9] which checks Android project files and warns developers of potential bugs and errors. It scans through the manifest file, resource files, Java source code, as well as compiled bytecode, and uses a set of pre-defined rules to find and report common issues. Developers may suppress lint warnings and extend lint by defining customized rules. The lint tool can already detect some of the errors and issues reported in our study, and we believe it can be enhanced, guided by our findings, to help developers avoid even more common errors and pitfalls.

3.4 Conclusion and Future Work

In this chapter, we have presented the first large-scale, systematic study of Android applications. We have analyzed over 120K applications from Google Play. Our analysis systematically examines Android applications, spanning across manifest files, permissions usage and application code. Our study has uncovered a large number of common errors and pitfalls during application development, revealed why such errors occur, discussed their implications, and provided actionable recommendations. We believe that our findings raise awareness and benefit the entire Android community to improve this large, new software ecosystem.

There are a few interesting directions for future research. First, we have focused on the applications' code-level characteristics, but not their meta information, such as developer,

category, and ratings. Considering such meta-data may reveal additional interesting insights. For example, applications across different categories may exhibit different features, such as different usage patterns of permissions and APIs. Another interesting direction is to analyze interactions among applications. We have focused on examining each application in isolation; interactions among applications may contain useful and interesting characteristics worth exploring in the future. It would also be interesting to conduct a similar study on iOS applications, which might be of higher quality in general as Apple has a rigorous vetting and approval process. Such a study may provide insights into the similarities and differences between the two dominating approaches to mobile application publishing and distribution.

Chapter 4

Analyzing and Understanding Sensitive Information Flow in Android Applications

Smartphones have become increasingly popular in recent years. Their popularity is driven in large part by an abundance of third-party applications. To achieve rich functionality, these applications often collect, use and store users' sensitive information, such as contacts, geolocation, and even bank account information. However, they may be vulnerable or even malicious, and expose sensitive information to unauthorized access. Although both static and dynamic approaches have been proposed to detect information leakage through the network, there has been little knowledge on the general problem of sensitive information usage in mobile applications. This work aims to gain a comprehensive understanding of how confidential information flows in Android applications. To this end, we identify common sources of sensitive information, perform a static taint analysis, and discover all possible sinks through which sensitive information may be leaked. We have designed and implemented Heimdall, a static analysis framework that automatically tracks the flow of sensitive information in Android applications. Using Heimdall, we have analyzed 122,570 unique Android applications and

discovered common unsafe sinks, including log output, network connection, public files, and SMS messages. We found 345,965 instances of potential unsafe flows across 46,508 applications. We have manually inspected a systematic selection of reported flows in top applications and found concrete evidence that developers do misuse or fail to sufficiently protect privacy-sensitive information. Results also show that our framework is precise and scalable for real-world deployment.

4.1 Introduction

As mobile devices grow in popularity, they have become indispensable in people's daily lives, keeping us connected to social networks, breaking news, and the entire Internet. While there are multiple competing platforms, Google's Android is currently the most popular operating system for mobile devices. Google takes an open stance toward Android — third-party applications are not rigorously scrutinized before they are made available to users on the Play Store. This open approach has led to security challenges and privacy concerns, and Google relies on community reviews (*e.g.*, user ratings and application flagging) to mitigate security threats. Google can also remove any detected or reported malware from the Play Store and remotely from affected devices [29]. To better defend against malware, Google recently launched Bouncer [66], a server-side security service that performs a series of analyses to detect hidden, malicious behavior when applications are uploaded to Google Play. However, it is found that Bouncer may be bypassed by sophisticated malware from knowledgeable adversaries [76]. More recently, Google launched a new application scanning service as an extension to Bouncer with the release of Android 4.2. This new security service is built into the platform to detect potentially malicious or harmful code when sideloading applications onto devices.

To implement rich features and achieve better user experience, applications frequently collect and compile sensitive data available on mobile devices. Many of these applications access users' personal information, such as geolocation, device identifiers and contact lists. Ideally, such accesses should be controlled to protect user privacy. Although applications

must ask for permissions to use sensitive information, it may not be clear to the end-user how such information is actually used after granting access. For instance, a photo-sharing application needs to access the network to transmit photos and acquire the user's location for geotagging the photos. In this example, the user needs to trust that the application only uses this privacy-sensitive information as indicated, which is not always the case. If the application is ad-supported, it may send the user's location along with other privacy-related data to advertisers' servers to display targeted ads. To mitigate this issue, recent legislation mandates that any application that collects personal data from users must conspicuously post a privacy policy describing how such information is collected, used and shared [57, 58]. Unfortunately, privacy policies often contain broad, vague legal language, and can be inaccurate or out-of-date [18, 19].

Besides applications that intentionally send privacy-sensitive data through the network, there are also applications that may inadvertently leak sensitive data. Some Android applications, including even system applications, were discovered to write sensitive information to the system log [65]. For instance, an application with location permissions may read the location data and later write it to the system log. Another (malicious) application, with the permission to read the log, can extract the location information from the log, although it cannot directly access the location data. In this example, the first application is vulnerable and may leak sensitive information unintentionally.

Similar to the system log example, vulnerable applications may obtain sensitive information and later write it to the file system (*e.g.*, the SD card). Unlike the system log, reading from the SD card does not require any permissions, making the vulnerability easier to exploit. In April 2011, a severe vulnerability was revealed in the widely used Skype Android application and may have affected over 10 million users [30, 88]. It was discovered that after login, the Skype application writes the user's contacts, profile, instant message logs, and other sensitive information in cleartext to a number of SQLite database files stored in the data directory. In addition, the Skype application mistakenly left those files with improper permissions that allowed any application to read the files.

To tackle the information leakage problem, researchers have proposed both static and

dynamic techniques to analyze the flow of sensitive information. While prior work focuses on *intentional leaks over the network*, little is known as to where and how sensitive information flows in general. In this work, we aim to gain a comprehensive understanding of sensitive information usage in third-party applications. In contrast to previous studies with limited scopes, our work considers both malicious and vulnerable applications and examines multiple sinks where sensitive information may flow. To this end, we design and implement Heimdall, a static taint analysis framework to track the flow of privacy-sensitive data in third-party Android applications.

Heimdall statically checks an application to identify suspicious code paths where it first accesses sensitive, permission-guarded information and subsequently transmits this information to a destination that can be accessed without the same guarding permissions. Since source code is rarely publicly available for Android applications, Heimdall performs analysis on bytecode. Leveraging an existing static information flow framework [72] built on top of the Soot library [92], we track the flow of sensitive information and determine if any such information reaches a potentially unsafe sink. We adapt the analysis to work on Android applications. In particular, we identify taint sources in event listeners as well as content providers, track the flow of sensitive information via programming constructs specific to the Android platform, and add support for the Android framework libraries (Section 4.3).

Using Heimdall, we have analyzed 122,570 free applications from the Google Play Store. Our comprehensive analysis uncovers four common types of unsafe sinks: log output, network connection, public files, and SMS messages. Among 79,354 applications that may access sensitive data, Heimdall has detected potential unsafe flows in 46,508 applications. We found pervasive flows of phone state and location data, and numerous flows to log and network sinks. As reported in previous studies [41, 42, 52, 59], we also observed extensive usage of advertising and analytics libraries, which result in many detected flows. To evaluate the precision of Heimdall, we systematically selected 100 flows from top applications and confirmed 65 true positives and 10 false positives through manual inspection. In addition, we were able to trigger detected leaks by running some of those applications on an Android device. Our results

demonstrate that applications frequently leak privacy-sensitive information and sometimes exfiltrate personal identifiable information (PII). We believe that our findings can help users make informed decisions regarding application usage and help security analysts detect malicious and vulnerable applications.

The rest of this chapter is organized as follows. Section 4.2 formalizes the problem that we tackle and presents our core approach. Next, we describe the design and implementation of Heimdall (Section 4.3). In Section 4.4, we present an extensive empirical evaluation of the effectiveness of Heimdall on a large number of third-party applications and discuss our analysis results. Section 4.5 surveys related work, and Section 4.6 concludes.

4.2 Problem and Approach Formulation

This section formalizes the notion of information leakage under our threat model. In this work, we do not consider vulnerabilities in the permission enforcement mechanism or other parts of the operating system that may be exploited to bypass restricted resource access. Our trusted computing base includes the Linux kernel, the Android framework and the Dalvik VM. In addition, we do not consider implicit flows, which in itself is indicative of suspicious behavior.

Under the security model of Android, applications are sandboxed from each other and the operating system. By default, they are not allowed to perform sensitive operations or access sensitive data.

Definition 4.1 (Protected Resource Access). Let R denote the universe of all resources defined in the system. Given a resource $r \in R$, $read(r)$ represents read access to r, and $write(r)$ represents write access to r. We define the universe of resource accesses as

$$S = \bigcup_{r \in R} \{ read(r), write(r) \}.$$

Each resource is protected with both a read and a write permission. Note that $write(r) \nRightarrow read(r)$, which means holding only the write permission to r does not automatically grant read

access to r. For instance, every application can write information to the system log, but not all applications can read from it.

Definition 4.2 (Permission Semantics). Let P denote the set of all permissions defined in the system. We define the semantics of a permission in terms of the resources it grants access to: $[\![\cdot]\!] : P \to \mathscr{P}(S)$. For permission $p \in P$, $[\![p]\!] = S_p \subseteq S$. We define the semantics of a permission set $P' \subseteq P$ as $[\![P']\!] = \bigcup_{p \in P'} [\![p]\!]$.

As of this writing, there are 200 permissions defined in the Android framework, some of which are not related to resource access. In this work, we focus on permissions p that involve granting accesses to resources (*i.e.*, $[\![p]\!] \neq \emptyset$).

Definition 4.3 (Application Permission Set). Let $a \in A$ denote an application. The application permission set $perm(a) \subseteq P$ is the set of permissions that are declared by application a in its manifest.

Definition 4.4 (Resource Content). To describe information stored within resources, we define the following function:

$$content : R \to \mathscr{P}(V),$$

where V is the value domain for sensitive information.

While some resources contain only one specific type of sensitive information, other resources may hold multiple types of sensitive data. For instance, the location provider only stores the geolocation, but the file system can be used to persist all types of sensitive information, such as contacts, SMS messages, and location data.

In this work, we focus on sensitive information protected by permissions. That is, an application $a \in A$ must hold appropriate permission $p \in P$ to retrieve sensitive information $v \in V$ from resource $r \in R$.

Definition 4.5 (Information Flow). Given an application $a \in A$, resources $r_1, r_2 \in R$ where $r_1 \neq r_2$, and sensitive information $v \in V$, we define the information flow of v from r_1 to r_2 as $r_1 \overset{a,v}{\rightsquigarrow} r_2$ if

(i) $\exists\ 0 \leq i < j < n$ on a program path $\pi = (\pi_m)_{0 \leq m < n}$;

(ii) at program point π_i, $v \in content(r_1)$, $v \notin content(r_2)$;

(iii) at program point π_j, $v \in content(r_2)$.

Based on the above definitions, we next define the notion of information leakage that we consider.

Definition 4.6 (Information Leakage). Application $a \in A$ leaks sensitive information $v \in V$ *w.r.t.* permission $p \in P$ if $\exists\ s, t \in R$ and $\exists\ p' \in P$ such that:

- $p \in perm(a)$
- $read(s) \in [\![p]\!]$;
- $read(s) \notin [\![p']\!]$;
- $write(t) \in [\![perm(a)]\!]$;
- $s \overset{a,v}{\rightsquigarrow} t$; and
- $read(t) \in [\![p']\!]$.

Each entity e with permission p' can leverage leakage from a to access sensitive data that are otherwise unavailable to e. Note that entity e can be another application or an external service (*e.g.*, an ads provider). In the case where sensitive information is sent over the network, no permission is required to access the sink. While such leaks are usually intentional, our formalization also captures inadvertent leaks where sensitive information v flows to unsafe sinks *w.r.t.* v's guarding permission p. For the log-based vulnerability described by Lineberry *et al.* [65], an application a has the permission to read location data v from the location provider s. The information v later flows into another resource t, which is the system log. Another (malicious) application a', with the only permission p' to read the log, can extract the location information from t, although it cannot request location data from s directly.

Based on Definition 4.6, we define the $isLeak$ function in terms of an application $a \in A$ and sensitive information $v \in V$ as follows:

$$\begin{aligned} isLeak(a,\ v) \quad = \quad & \exists\ p, p' \in P,\ s, t \in R \\ & (\ p \in perm(a) \\ & \wedge\ read(s) \in [\![p]\!] \\ & \wedge\ read(s) \notin [\![p']\!] \\ & \wedge\ write(t) \in [\![perm(a)]\!] \\ & \wedge\ s \overset{a,v}{\rightsquigarrow} t \\ & \wedge\ read(t) \in [\![p']\!]\ ). \end{aligned}$$

We say that an application $a \in A$ is free from information leakage if a does not leak any $v \in V$:

$$\forall v \in V : \neg\ isLeak(a,\ v)$$

For each application a, we check the flows of all possible sensitive data v to see if there is any leakage. If a leak is identified, we report the resources s and t, which can be used for inspection or further analysis.

4.3 Design and Implementation of Heimdall

We have designed and implemented Heimdall, a static analysis framework, to realize our formalization. This section discusses its technical details.

Figure 4.1 depicts the architecture of Heimdall. Before analyzing an application, we first preprocess each APK file to extract the manifest file and convert the Dalvik executable into standard Java bytecode using existing tools [14, 81]. Next, we determine whether the application may access sensitive information based on the permissions requested in its manifest. If so, we perform static taint analysis to track the flow of sensitive data within that application.

Unlike prior work that only focuses on network sinks, we are interested in a comprehensive

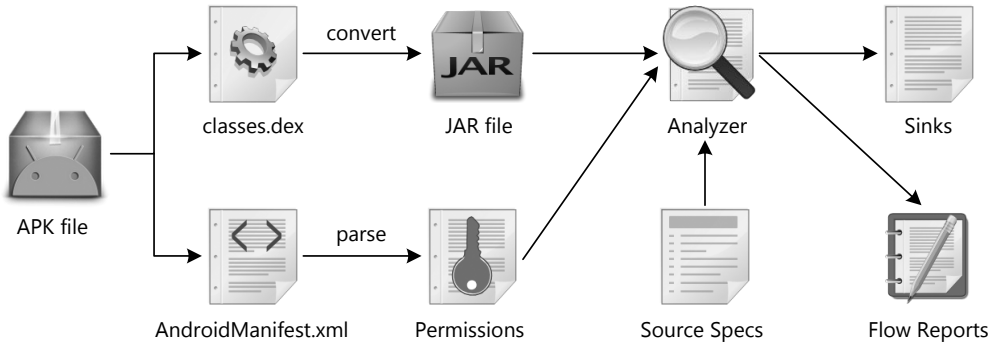


Figure 4.1: Heimdall system overview.

understanding of sensitive information usage in general. We aim to discover all possible unsafe sinks, so we do not specify pre-determined sinks in our analysis. For this reason, we cannot start with a reachability analysis on the CFG to identify paths connecting sources and sinks and perform on-demand dataflow analysis on reported paths. Instead, we start from program entry points, identify taint source accesses, and conservatively approximate the flow of sensitive information. The remainder of this section describes our design in detail.

4.3.1 Entry Points

Due to the event-driven nature of Android applications, each application may have multiple entry points. The Android framework uses callbacks extensively, and the execution of application code often interweaves with framework code. For instance, when a thread is created and started, the framework takes over control, and the execution later resumes at the `run` method for that thread. To focus on the analysis of application code, we need to know where execution may begin from an application's perspective. Fortunately, the behavior of the framework is well-documented, so we leverage this well-defined semantics to identify entry points for our analysis. Specifically, we consider the following entry points.

Lifecycle Methods Application components are the building blocks for Android applications and usually serve as top-level entries through which users or the system may enter an application. The Android framework supports four types of components, each of which has a well-defined

lifecycle. The system invokes a series of lifecycle methods, while each component transitions between its various states of its lifecycle. For instance, an activity component must subclass `android.app.Activity` (or one of its subclasses) and implement the `onCreate` method to perform initializations and set up the UI layout. The system invokes other lifecycle methods like `onStop`, `onResume`, `onPause`, and `onDestroy`, when the activity is being stopped, resumed, paused, or destroyed respectively. Different types of components have different lifecycle methods, and we consider all such methods as entry points for our analysis.

UI Callbacks Besides lifecycle methods, application components also implement various callbacks to respond to user interactions or system events. For example, the `onBackPressed` method of an activity is called when the user presses the back key, and `onLowMemory` is called when the system is running low on memory. Compared to components, user interactions happen more frequently on UI elements (subclasses of `android.view.View`) such as buttons and text fields. These UI elements usually set up listeners that will be notified of specific events. For example, calling `setOnClickListener` on a button registers a callback method (*i.e.*, `onClick`) that will be invoked by the system when the user clicks the button. We treat all these callback methods as entry points.

Asynchronous Callbacks In addition to UI callbacks, applications may register other callbacks and listeners to receive data asynchronously. Consider the code snippet in Figure 4.2. To retrieve user's real-time location, an application needs to define a `LocationListener` and register it with the `LocationManager` to receive location updates. When a new location fix is available, the `onLocationChanged` callback method defined in the `LocationListener` is called to receive new location data. We identify this and other asynchronous callbacks supported by the framework and consider them entry points.

Runnable Methods Another common type of entry points that we consider is the `run` method for classes that implement the `java.lang.Runnable` interface. For example, `run` is the

```
LocationManager locationManager = (LocationManager)
       getSystemService(Context.LOCATION_SERVICE);
LocationListener locationListener = new LocationListener() {
  public void onLocationChanged(Location location) {
    // use new location data
  }
  ...
};
locationManager.requestLocationUpdates(
       LocationManager.NETWORK_PROVIDER, 0, 0, locationListener);
```

Figure 4.2: Requesting location updates with asynchronous callback.

entry point when a new thread (*e.g.*, `java.lang.Thread`) is started or when a task (*e.g.*, `java.util.TimerTask`) is performed. To improve UI responsiveness and user experience, long-running operations (*e.g.*, network I/O) are usually performed in background worker threads. The Android framework also offers a few APIs, such as `Activity.runOnUiThread(Runnable)` and `View.post(Runnable)`, to post `Runnable` actions from background threads to the event queue of the UI thread.

Special Framework Methods The Android framework provides the `Handler` class to facilitate the interactions and communications between threads. A `Handler` can be used to send and process `Message` and `Runnable` objects in the message queue associated with a thread. To use a handler, applications need to subclass `android.os.Handler` and override the `handleMessage` method to process messages. Therefore, we consider `handleMessage` methods as entry points in our analysis. Another class in the Android framework, `AsyncTask`, encapsulates `Thread` and `Handler` and provides a convenient way for background processing. It runs an asynchronous task on a background thread and publishes results on the UI thread. Applications using `AsyncTask` need to subclass `android.os.AsyncTask` and complete the task in the following steps: `onPreExecute`, `doInBackground`, `onProgressUpdate` and `onPostExecute`, all of which we treat as entry points in our analysis. We discuss `AsyncTask` and `Handler` in detail in Section 4.3.5.

Source Type	Permission*
Accounts	AUTHENTICATE_ACCOUNTS, GET_ACCOUNTS, MANAGE_ACCOUNTS, USE_CREDENTIALS
Calendar	READ_CALENDAR
Call Log & Contacts	READ_CONTACTS, READ_CALL_LOG†
Camera	CAMERA
History & Bookmarks	READ_HISTORY_BOOKMARKS
Location	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
Microphone	RECORD_AUDIO
Phone State	READ_PHONE_STATE
SMS & MMS	READ_SMS
Tasks	GET_TASKS
User Dictionary	READ_USER_DICTIONARY
WiFi State	ACCESS_WIFI_STATE

Table 4.1: Sensitive sources and guarding permissions.

*Permission prefixes are omitted. Only `READ_HISTORY_BOOKMARKS` has the `com.android.browser.permission.` prefix and all other permissions have the `android.permission.` prefix.

†The `READ_CALL_LOG` permission was introduced in Android 4.1. Access to the call log is implicitly provided through `READ_CONTACTS` for applications that target earlier systems.

4.3.2 Taint Sources

To perform privacy analysis, we need to identify privacy-sensitive sources. In this work, we focus on tracking sensitive information that is guarded by permissions defined by the platform. Table 4.1 lists the types of sensitive information we consider along with permissions that guard those resources.

We use the permissions listed in Table 4.1 to identify taint sources. Specifically, we leverage the permission map presented in Stowaway [45] and examine the Android framework source code to identify APIs and content providers that are guarded by those permissions. Then we look for the usage of such APIs and content providers in the application bytecode to find potential accesses to sensitive information.

Sources of sensitive information cover many aspects of the Android platform, and such information is accessed by applications in a variety of ways. We have carefully studied each type

```
TelephonyManager telephonyManager = (TelephonyManager)
       getSystemService(Context.TELEPHONY_SERVICE);
String deviceId = telephonyManager.getDeviceId();
```

Figure 4.3: Obtaining device ID through direct API access.

of privacy-sensitive information and determined how to define taint sources.

Direct API Access This is the most straightforward way to obtain sensitive information. After calling these APIs, sensitive data is either available in the return value or in the parameters. For instance, device identifiers can be retrieved using such APIs. Figure 4.3 shows how to access the unique device identifier, *i.e.*, the IMEI, MEID, or ESN of the phone, depending on the network technology.

In this case, we simply mark the return value (or argument) of the protected API as tainted and subsequently track its flow.

Callbacks and Listeners Some types of privacy-sensitive information are acquired through sensors like the microphone and camera. Such information changes frequently and is often made available through asynchronous APIs. Recall the location listener example from Section 4.3.1. To receive location data, applications need to register a callback method (*i.e.*, `onLocationChanged` inside `LocationListener`) with a sensor manager (*i.e.*, `LocationManager`). When location information is available, the sensor manager invokes the registered callback method and passes the data through its arguments. To handle cases like this, we identify such callback and listener methods and mark their arguments that hold sensitive data as tainted.

Content Providers Shared information such as contacts and SMS messages are stored in content providers. Such data are encapsulated and accessed in a database-like manner. An application can access data from a content provider with a `ContentResolver` client object, which provides basic CRUD (*i.e.*, create, retrieve, update, and delete) operations. Figure 4.4 illustrates how to access the call log.

```
ContentResolver cr = getContentResolver();
Cursor c = cr.query(CallLog.Calls.CONTENT_URI, null, null, null, null);
while (c.moveToNext()) {
  String number = c.getString(c.getColumnIndex(CallLog.Calls.NUMBER));
  long date = c.getLong(c.getColumnIndex(CallLog.Calls.DATE));
  int type = c.getInt(c.getColumnIndex(CallLog.Calls.TYPE));
  ...
}
```

Figure 4.4: Retrieving call log data using a content provider.

In order to get a list of recent calls, an application invokes the `query` method on a `ContentResolver`, which dispatches the query to the call log content provider identified by its content URI. The `query` method returns a `Cursor` that provides random read access to the rows and columns of the results. Using the `Cursor` methods, applications can then iterate through the rows and retrieve data from each column.

Although privacy-sensitive data are retrieved using `Uri` and `Cursor` objects, operations on these objects do not leak sensitive information directly. To identify taint sources, we examine the first argument to the `query` method and determine if the content URI matches any of the important ones protected by permissions. In the case of a match, we attach to the returned `Cursor` object a tag, which indicates the type of sensitive information stored in the protected content provider. Then we track the tag and mark the return value as tainted when data is retrieved from the `Cursor` using methods like `getInt`, `getLong`, and `getString`.

The presence of sensitive API calls or content provider usage in the bytecode does not mean that the application would actually access sensitive information at run time. For better precision, we perform a filtering step to verify if an application has required permissions for sensitive information access. Specifically, for each application, we only consider taint sources that are guarded by permissions requested in the manifest. For instance, we discard location related sources if the application does not hold the `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permission. This step is important because advertising libraries usually check, at run time, if the host application has a certain permission before executing functionalities that require the permission. In particular, such libraries serve geo-targeted ads only when the

```
String data = getSensitiveData();
URL url = new URL("http://example.com/");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
DataOutputStream dos = new DataOutputStream(conn.getOutputStream());
dos.writeBytes(data);
dos.close();
```

Figure 4.5: An example of wrapped stream as a taint sink.

running application has proper location permissions. For applications using the `sharedUserId` feature, we conservatively assume that they may hold any permission and thus do not perform this filtering step.

4.3.3 Taint Sinks

Our privacy analysis aims to gain a comprehensive understanding of sensitive information usage in Android applications. Therefore, we do not specify taint sinks at the beginning of the analysis. Rather, we discover where sensitive information flows as we analyze the applications. For our analysis, a taint sink is a location in the application code (*i.e.*, an API call) where tainted data is consumed. We divide taint sinks into two categories: *safe sinks* and *unsafe sinks*. API methods that fall into safe sinks do not leak sensitive data or further propagate taint markings. For instance, we consider APIs that send information to UI elements as safe sinks. When sensitive data flows into unsafe sinks, it is either exfiltrated (*e.g.*, through the network) or can be later retrieved (by other applications) without the same permissions that are required to access the sensitive data. Such sinks include APIs that send information over the network. We describe the discovered taint sinks in detail in Section 4.4.

Android applications frequently take advantage of a stream wrapping feature in the Java I/O package. Specifically, simple stream objects are often wrapped inside other higher-level stream objects to gain additional capabilities. For instance, an `OutputStream` object is often wrapped by a `BufferedOutputStream` object for improved performance. This behavior poses a challenge while we try to identify the type of taint sinks. Consider the code snippet in Figure 4.5.

In this example, the `OutputStream` associated with an `HttpURLConnection` is wrapped by

```
String data = getSensitiveData();
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeBytes(data);
sendToServer(baos.toByteArray());
```

Figure 4.6: An example of wrapped stream as a non-sink.

a `DataOutputStream` to use the higher-level `writeBytes` method. Any sensitive information written to the `DataOutputStream` is sent over the network. However, without analyzing the Java I/O library, we only know that sensitive data flows into a `DataOutputStream` object without discovering the network sink. To address this issue, we keep track of API calls like `URLConnection.getOutputStream` and attach a tag to the returned `OutputStream` object. We leverage the same information flow analysis to propagate such tags, so that we know what the real sink is when sensitive data flows into wrapper streams.

Moreover, this stream wrapping feature also introduces another problem while we determine taint sinks. We describe the problem using an example in Figure 4.6.

While `DataOutputStream.writeBytes` is a sink method in the previous example, the same API is not a sink here. In this case, sensitive data flows to the wrapped `ByteArrayOutputStream` and is sent over the network later. This means that some API methods of higher-level streams may act either as a sink or as a non-sink, depending on the type of streams wrapped inside. In the case of non-sinks, we need to propagate taint markings back to the wrapped stream (`ByteArrayOutputStream` in this example). To this end, we perform an intra-procedural analysis to map the wrapper stream to potential wrapped streams with the help of a fast and simple alias analysis. We then mark the wrapped objects as tainted and continue to track the flow of information.

Similar to the filtering step used in identifying taint sources, we extract permissions used by each application and verify if the application has proper permissions to write sensitive information to taint sinks. For instance, we do not consider flows to network sinks if the application does not have the `INTERNET` permission. Again, we do not apply this filtering step

Taint Propagation	Description
arg ← arg	Taint one argument from another argument
arg ← base	Taint argument from base object
base ← ∅	Clear taint from base object
base ← arg	Taint base object from argument
return ← arg	Taint return value from argument
return ← base	Taint return value from base object

Table 4.2: Taint propagation rules.

on applications using the `sharedUserId` feature for a conservative analysis.

4.3.4 Dataflow Analysis

Heimdall is built on top of an existing information flow analysis framework [72]. The framework relies on Soot [92] to load classes and perform pointer analysis. To analyze Android applications, we have customized and extended the framework to handle Android-specific APIs and flows.

Since we do not analyze API methods in the standard Java library and Android framework, we capture the flows that may occur as a result of such method invocations in the form of derivation rules (*i.e.*, transfer functions). For each rule, we specify a method signature and provide a taint propagation logic, which describes relevant dataflow information between arguments and return values. Table 4.2 lists the different types of rules that we consider.

We specify transfer functions for commonly used Java and Android APIs by carefully studying the documentation and occasionally examining the framework source code. While the behavior of most API methods is straightforward, some APIs have complex behavior and we need to specify multiple transfer functions at the same time. For instance, the `String.concat` method concatenates the base string with the specified string passed in the argument, so we provide a combination of "return ← arg" and "return ← base" as the rule. Note that we only specify transfer functions for APIs that handle sensitive data, and additional transfer functions can be added easily when necessary.

Precise static taint analysis is a challenging (in fact undecidable) problem. For our analysis, we take a conservative approach while performing taint propagation. For instance, we treat an

entire array or collection as tainted if any member is tainted. Similarly, we taint a complex object (*e.g.*, `android.content.Intent`, `android.os.Bundle`) when sensitive information is stored inside the object. While this strategy leads to over-tainting and may introduce false positives, it allows us to find as many important flows as possible to meet our goal of being comprehensive. Our empirical results in Section 4.4 also show that our analysis is quite precise in practice.

Our information flow analysis is flow-sensitive, meaning that different information state is obtained at different program points. Our analysis is also context-sensitive, so calling contexts are considered and methods are analyzed separately for each call site. The analysis iterates until it reaches a fixpoint.

4.3.5 Android-specific Flows

As mentioned previously, the Android framework provides a few utilities that allow easy and convenient background processing in applications. Such utilities are widely used in Android applications, so we need to support them properly to track the flow of sensitive information.

The `AsyncTask` class offers a simple way to perform short background operations and publish results on the UI thread. An asynchronous task is started by `execute(Params...)` from the UI thread and completes in these steps: `onPreExecute()`, `doInBackground(Params...)`, `onProgressUpdate(Progress...)`, and finally `onPostExecute(Result)`, where `Params`, `Progress` and `Result` are generic types. Only `doInBackground` runs in a pool of background threads, while the other three methods run in the UI thread. Figure 4.7 shows an example of an `AsyncTask`.

While the use of `AsyncTask` greatly simplifies the execution of worker thread tasks that interact with the UI, it introduces some implicit control-flows and dataflows. We handle the control-flows by adding the step methods as entry points (Section 4.3.1), and here we describe how we connect the implicit dataflows. A thorough examination of the documentation reveals the following flows:

- The arguments of `execute` of generic type `Params` are sent to `doInBackground` upon

```
private class MyAsyncTask extends AsyncTask<String, Integer, Long> {
  protected void onPreExecute() {
    showProcessingDialog();
  }

  protected Long doInBackground(String... args) {
    int count = args.length;
    long result = 0;
    for (int i = 0; i < count; i++) {
      result += doSomeCalculation(args[i]);
      publishProgress((int) ((i / (float) count) * 100));
    }
    return result;
  }

  protected void onProgressUpdate(Integer... progress) {
    setProgressPercent(progress[0]);
  }

  protected void onPostExecute(Long result) {
    dismissProcessingDialog();
    showResult("Result: " + String.valueOf(result));
  }
}

new MyAsyncTask().execute(str1, str2, str3);
```

Figure 4.7: Example code with `AsyncTask`.

task execution.

- The arguments of `publishProgress` of generic type `Progress` are sent to the `onProgressUpdate` method during background computation.
- The return value of `doInBackground` of generic type `Result` flows to `onPostExecute` at the end of background computation.

We model the above behavior and automatically propagate taint markings accordingly when we encounter subclasses of `AsyncTask` in the application bytecode. Our analysis also ensures that sensitive information only flows between methods of the same `AsyncTask` subclass.

`Handler` is another commonly used class that needs special support. A `Handler` object is often used to send and process `Message` and `Runnable` objects. It offers a simple way for

```
private Handler mHandler = new Handler() {
  public void handleMessage(Message msg) {
    switch (msg.what) {
      case LOG_DATA:
        Bundle bundle = msg.getData();
        String data = bundle.getString("key");
        log(data);
        break;
    }
  }
};

Thread t = new Thread() {
  public void run() {
    String data = getSensitiveData();
    Message msg = mHandler.obtainMessage(LOG_DATA);
    Bundle bundle = new Bundle();
    bundle.putString("key", data);
    msg.setData(bundle);
    mHandler.sendMessage(msg);
  }
}.start();
```

Figure 4.8: Example code with `Handler`.

communications between different threads. Figure 4.8 illustrates how it is used. In this example, a `Handler` is created on the UI thread and overrides the `handleMessage` method to process messages. Sensitive data is accessed from a background thread, put into a `Bundle` object, and posted to the `Handler` through a `Message`. Then the data is retrieved from the `Bundle` and sent to the `log` function.

To support the `Handler` feature, we start by analyzing the `handleMessage` method (Section 4.3.1). We also specify proper transfer functions for methods in the `Message` and `Bundle` utility classes, which often process sensitive data. Finally, we propagate taint from `sendMessage` to `handleMessage` of the same `Handler` subclass in order to connect the flows.

Source Type	#apps	%
Accounts	5,625	4.59%
Calendar	9,112	7.43%
Call Log & Contacts	9,851	8.04%
Camera	9,469	7.73%
History & Bookmarks	3,735	3.05%
Location	46,936	38.29%
Microphone	3,904	3.19%
Phone State	58,601	47.81%
SMS & MMS	1,557	1.27%
Tasks	6,818	5.56%
User Dictionary	54	0.04%
WiFi State	15,754	12.85%

Table 4.3: Number of applications with permissions to access sensitive sources.

4.4 Empirical Study and Analysis

We have used Heimdall to study the same 122,570 unique Android applications described in Section 3.2.1 and analyzed how these applications use privacy-sensitive data.

By parsing and analyzing the manifest files, we found that 7,784 applications do not request any permissions. Among the remaining, 35,802 do not use any permissions listed in Table 4.1. We also observed that only 862 applications (0.70%) use the `sharedUserId` feature. In total, 43,216 applications cannot access sensitive information, and there is no need to analyze them. Table 4.3 shows the number of applications that have permissions to access each type of sensitive information. We observe that phone state and location are the most popular types of sensitive information applications access, followed by WiFi state. We use Heimdall to analyze these applications to find unsafe sinks and flows.

4.4.1 Identifying Taint Sinks

We analyze all the applications that may access sensitive information to track where such information flows. During this process, we have observed common API method sinks and grouped them into a few categories by their resource types. For instance, we group all methods

that write data to the system log to a single log category. For each sink category, we determine if it is safe or unsafe, depending on whether information may be exfiltrated or leaked through that resource.

In our study, we have observed the following safe sinks:

- ***Private Files*** Internal storage provides private data storage for applications. By default, files written to the internal storage are private to the application, so other applications or the user cannot access them. Since the Android system manages and properly guards such private files, we consider them safe sinks for private information.

- ***Shared Preferences*** The shared preferences provide applications a general framework for accessing and modifying persistent preference data in the form of key-value pairs. By default, shared preferences cannot be accessed by other applications, and are mostly used to store data that is private to the application. Therefore, we consider it safe to write sensitive data to shared preferences.

- ***SQLite Databases*** The Android framework provides full support for SQLite databases. In most cases, a database is only accessible to the owner application. While an application may implement a content provider to expose information stored in its database to other applications, the database itself is still considered safe.

- ***UI Elements*** Applications that retrieve sensitive data often process the data and then display the information on the UI to provide user feedback. This behavior is usually legitimate use of private information and does not pose threats to users' privacy[1]. Since applications cannot access UI elements of other applications, displaying information on the screen is considered safe.

By studying the flows, we have discovered and identified four types of unsafe sinks:

- ***Log Output*** Android applications frequently utilize the logging system for debugging and troubleshooting purposes. It is common for applications to write sensitive information

[1]We do not consider screen peeping attacks.

to the log [42, 65], although the documentation warns developers against doing so. It is possible for a malicious application to request the `READ_LOGS` permission and harvest sensitive information from the log that is otherwise unavailable to it. Since Android 4.1, the `READ_LOGS` permission has a higher protection level so that it will only be granted to system applications. This change also broke some third-party applications that legitimately use this permission.

- ***Network Connection*** Sensitive data exfiltration over the network is the focus of most previous studies. This type of leakage is frequently found in applications that are ad-supported. We capture such leaks by treating the network connection as an unsafe sink. Applications need the `INTERNET` permission to establish network connections.

- ***Public Files*** Although files written to internal storage are private by default, application may explicitly make those files publicly accessible (*e.g.*, due to misconfigurations). Such public files on internal storage are considered unsafe sinks. As opposed to internal storage, files saved on external storage is world-readable and world-writable, which means that other applications or the user may access or modify the files. For this reason, we consider it a leak when sensitive information flows to external storage. Since applications do not need any permissions to read files from external storage, such leaks can be very easily exploited. To mitigate this issue, Android 4.1 introduces the `READ_EXTERNAL_STORAGE` permission that protects read access to external storage. However, this permission is not enforced by default yet, and applications requesting the `WRITE_EXTERNAL_STORAGE` will be granted the read permission automatically.

- ***SMS Messages*** Similar to network sinks, sensitive information may leave the device through SMS messages. To detect such leaks, we consider SMS as an unsafe sink. Applications need the `SEND_SMS` permission to send SMS messages.

When sensitive information flows into unsafe sinks in an Android application, we consider it a potential privacy leak, because either 1) the application intentionally exfiltrates sensitive data

Source Type	#flows	flow%	#apps	app%
Accounts	3,478	1.01%	1,239	2.66%
Calendar	216	0.06%	43	0.09%
Call Log & Contacts	9,503	2.75%	1,339	2.88%
Camera	955	0.28%	696	1.50%
History & Bookmarks	526	0.15%	181	0.39%
Location	165,282	47.77%	24,081	51.78%
Microphone	287	0.08%	192	0.41%
Phone State	156,929	45.36%	30,109	64.74%
SMS & MMS	836	0.24%	225	0.48%
Tasks	1	0.00%	1	0.00%
User Dictionary	8	0.00%	6	0.01%
WiFi State	7,944	2.30%	2,181	4.69%

Table 4.4: Number of flows and apps by source type.

and sends it to an external entity; or 2) another application may exploit the vulnerability of this application to indirectly access sensitive information.

Now we present the results of our study and report the potential leaks through unsafe sinks.

4.4.2 Results

We have analyzed all the applications that have permissions to access sensitive data. For each application, we limit the analysis time to two minutes [2], and Heimdall is able to complete the analysis on over 83% of these applications. We found a total of 345,965 flows to unsafe sinks in 46,508 of the 79,354 applications.

Table 4.4 and Table 4.5 present the number of flows and applications for each source and sink type respectively. We can see that phone state and location data are the most popular types of sources flowing to unsafe sinks, which is consistent with our findings in Table 4.3. For sinks, most sensitive information flows to the system log or the network, while they are rarely leaked via SMS messages.

We further examine the most popular types of sources, *i.e.*, phone state and location data, and break them down to better understand which and how frequently sensitive information is

[2]We have also tried longer timeout threshold (*e.g.*, five minutes). However, given the large number of applications, we did not discover significantly increased number of flows but observed considerably longer analysis time.

Sink Type	#flows	flow%	#apps	app%
Log Output	169,924	49.12%	27,227	58.54%
Network Connection	171,192	49.48%	34,146	73.42%
Public Files	3,945	1.14%	1,776	3.82%
SMS Messages	904	0.26%	323	0.69%

Table 4.5: Number of flows and apps by sink type.

Source		#flows	flow%
Phone	getDeviceId	113,681	72.44%
	getLine1Number	27,722	17.66%
	getSimSerialNumber	1,274	0.81%
	getSubscriberId	14,022	8.94%
	Other	230	0.15%
Location	getCellLocation	2,802	1.70%
	getLastKnownLocation	74,991	45.37%
	getNeighboringCellInfo	173	0.10%
	onLocationChanged	87,316	52.83%

Table 4.6: Breakdown of phone state and location flows.

leaked. Table 4.6 shows the flows of phone state sources, divided into different phone state data. We can see that the vast majority of flows leak the unique device identifier (*e.g.*, the IMEI for GSM devices and the MEID or ESN for CDMA devices). A device ID uniquely identifies a mobile device and can be used to prevent a stolen device from accessing the cellular network. More than 17% of the flows may leak the phone number (*e.g.*, the MSISDN for a GSM phone) and approximately 9% flows potentially exfiltrate the subscriber identifier (*e.g.*, the IMSI for a GSM phone). While a phone number can be linked to a specific user (*e.g.*, via social network [43]), device and subscriber identifiers cannot be used to immediately identify users. However, it is still possible for popular library providers (*e.g.*, advertisers) to aggregate detailed application usage information to learn about users' behavior. Therefore, leaking the phone state data poses privacy risks to users.

Table 4.6 also shows how leaked location data is collected. More than half of the flows leak the current location, which is acquired by registering a location listener and receiving asynchrous location updates in `onLocationChanged`. For better user experience, it is a common practice

Sink		#flows	flow%
Log	android.util.Log	165,742	97.54%
	java.util.logging.Logger	595	0.35%
	org.apache.commons.logging.Log	3,587	2.11%
Network	HttpClient	81,111	47.38%
	OutputStream	30,232	17.66%
	Socket	5,364	3.13%
	URL	35,793	20.91%
	WebKit	18,692	10.92%

Table 4.7: Breakdown of log and network sinks.

for applications to utilize a cached location that represents the last known location fix before a more accurate current location is received. Over 45% of the location flows leak cached locations. Both current and cached locations can be retrieved from two location providers: the network provider that serves coarse location data based on cell towers and WiFi access points, and the GPS provider that serves fine location data using satellites. Heimdall tracks location flows from both providers. Location data retrieved via the `TelephonyManager` (*i.e.*, `getCellLocation()` and `getNeighboringCellInfo()`) is not as common and consitutes less than 2% of the flows.

To gain a better understanding of how sensitive information is leaked, we have taken a close look at the most popular sinks, *i.e.*, log output and network connection. Table 4.7 indicates the number of flows to the log through the Android framework logging library, the general Java logging library, and the Apache Commons logging library. The default logging handler in Android is configured such that log messages written with `java.util.logging` and `org.apache.commons.logging.Log` are passed on to `android.util.Log`[3]. We can see that the vast majority of flows end up in methods in the Android logging library while Java and Apache Commons logging libraries are rarely used in detected flows.

Table 4.7 also summarizes different ways sensitive information exfiltrates through the network. We have observed that leaks most frequently occur via HTTP requests (*e.g.*, GET and POST) sent with `HttpClient` connections. It is also common to leak data through URL

[3]Log messages with level lower than `INFO` are ignored.

Flow Type (Source → Sink)	#flows	flow%
Phone State → Network Connection	96,198	27.81%
Location → Log Output	95,690	27.66%
Location → Network Connection	67,852	19.61%
Phone State → Log Output	59,580	17.22%
Call Log & Contacts → Log Output	7,358	2.13%
WiFi State → Log Output	4,167	1.20%
WiFi State → Network Connection	3,631	1.05%
Accounts → Log Output	1,752	0.51%
Accounts → Network Connection	1,588	0.46%
Call Log & Contacts → Network Connection	1,561	0.45%

Table 4.8: Top flows by source and sink types.

objects, where sensitive data is used to construct URL strings, and through `OutputStream` objects associated with `URLConnection`s. Another way of sending data over HTTP is to use WebKit (*e.g.*, via `WebView`), which constitutes over 10% of the flows. Finally, a small number of flows use Java Socket API to leak sensitive data.

For each flow, we consider the pair of source and sink types to be the flow type. We have observed a total of 42 different types of flows. Table 4.8 shows the top 10 most popular flow types, and Appendix B lists all 42 types of flows detected by Heimdall.

During our analysis, we have noticed that third-party libraries are responsible for many flows related to phone state and location data. As developers frequently rely on advertising to monetize their free applications, we have observed that a large number of applications bundle one or more popular advertising libraries. Such libraries often access and utilize the user's location, when available, to serve geo-targeted ads. Moreover, we have also found extensive use of analytics and tracking libraries, which also collect sensitive information such as device identifiers to compile application usage statistics. Table 4.9 shows the top 10 popular advertising and analytics libraries used in our application collection. Among them, Google Analytics and Flurry belong to analytics and tracking libraries, while the other 8 are from mobile ad networks. Note that not all the applications embedding these libraries leak sensitive information. It is a common practice among the libraries to check if the host application has certain permissions and

Library Name	Package Name	#apps
AdSense	com.google.ads	39,462
AdMob	com.admob.android.ads	11,125
Google Analytics	com.google.android.apps.analytics	10,762
Millennial Media	com.millennialmedia.android	9,489
Flurry	com.flurry.android	6,619
InMobi	com.inmobi.androidsdk	6,363
MdotM	com.mdotm.android.ads	5,242
AdWhirl	com.adwhirl	4,923
Quattro	com.qwapi.adclient.android	4,667
Mobclix	com.mobclix.android	4,470

Table 4.9: Top advertising and analytics libraries.

only access sensitive data when the check succeeds. Altogether, these libraries are responsible for 76,534 of the 345,965 flows (22.12%) reported by Heimdall.

Given the large number of reported flows, it would be impossible to manually verify each of them. In the following section, we present a more in-depth manual analysis by focusing on the detected flows in top applications.

4.4.3 In-depth Manual Analysis

We prioritize our efforts and focus on the top 1,000 applications, while we manually verify the validity of reported flows. We suspect such flows are more important as popular applications have much larger install bases. Among the top 1,000 applications, 238 do not have any permissions listed in Table 4.1 or use the `sharedUserId` feature, so we have excluded them from our analysis. Generally, top applications are larger in size and take more time to analyze. Therefore, we allow the analyzer to run for 10 minutes on each application, and Heimdall is able to fully analyze over 86% of the 762 applications.

Heimdall reports 6,420 flows in 528 of the 762 applications. We systematically select 100 flows for manual inspection. First, we categorize all the flows by their source and sink types like we did in Table 4.8. For the top 1,000 applications, we have observed 31 different combinations of source and sink types. One flow is picked uniformly at random from each of these 31

categories, and then we select the remaining 69 flows such that the number of flows picked from each category is proportional to the size of that category. Again, flows within the same category are chosen uniformly at random. The 100 selected flows span across 88 applications (see Appendix C).

To aid our manual inspection, Heimdall provides for each reported flow the sensitive resource, the sink API method, and the context (*i.e.*, stack trace from an entry point leading to the sink). We manually verified 100 flows and found that 65 of them are true positives. We were also able to confirm that 10 flows are false positives. Such false positives are mostly caused by overtainting on containers (*e.g.*, arrays and the Java collections framework) and sometimes by imprecision of the alias analysis implemented in the information flow library. Due to complex functionalities in top applications and code obfuscation in some cases, we were not able to classify the remaining 25 flows as true positives or false positives. Table 4.10 and Table 4.11 present the results of our manual analysis, broken down by the types of sources and sinks respectively. In both tables, column `#TP` shows the number of confirmed true positives, `#FP` the number of false positives, and `#Unk` the number of unconfirmed flows. The last two columns list the number of selected flows for manual inspection and the total number of flows in each category. Table 4.12 lists all 31 categories of source and sink pairs and the results of our manual validation. We can see from the results that Heimdall is quite precise and has a reasonably low false positive rate. We have also found that advertising and analytics libraries are responsible for 23 of the 100 flows that we examined.

4.4.4 Case Studies

We now discuss in detail a few of our findings with respect to flows to different types of sinks. For this study, we have installed applications under analysis on a Galaxy Nexus smartphone running Android 4.1.2 and confirmed the detected flows by triggering them at run time.

Log Output Symantec offers a security application called *Norton Security and Antivirus* for free with basic features including anti-theft and anti-malware. Users have the option to upgrade

Source Type	#TP	#FP	#Unk	#Sel	#All
Accounts	4	0	4	8	460
Calendar	2	0	0	2	8
Call Log & Contacts	10	0	6	16	982
Camera	3	0	0	3	15
History & Bookmarks	1	0	1	2	10
Location	18	5	6	29	2,374
Microphone	1	2	0	3	45
Phone State	20	2	5	27	2,131
SMS & MMS	0	0	2	2	7
Tasks	0	0	0	0	0
User Dictionary	1	0	0	1	2
WiFi State	5	1	1	7	386
Total	65	10	25	100	6,420

Table 4.10: Number of verified flows by source type.

Sink Type	#TP	#FP	#Unk	#Sel	#All
Log Output	34	5	12	51	3,762
Network Connection	24	5	8	37	2,442
Public Files	5	0	3	8	173
SMS Messages	2	0	2	4	43
Total	65	10	25	100	6,420

Table 4.11: Number of verified flows by sink type.

to the full version through in-app billing, which unlocks additional features such as call & SMS blocking and Web protection. Heimdall detected a few flows of sensitive information in this application. The license manager, which determines a user's subscription status on application launch, accesses the device's unique identifier and MAC address and then dumps them into the system log. We manually verified this behavior on our Android device by monitoring the system log with logcat[4].

We observed that the *Smooth Calendar* application writes information retrieved from a user's calendar to the log. Heimdall detected flows from the calendar content provider to the system log using the `Log.d` method. We were able to trigger this flow on our Android device. When

[4]http://developer.android.com/tools/help/logcat.html

Flow Type (Source → Sink)	#TP	#FP	#Unk	#Sel	#All
Accounts → Log Output	2	0	3	5	395
Accounts → Network Connection	2	0	0	2	57
Accounts → SMS Messages	0	0	1	1	8
Calendar → Log Output	1	0	0	1	4
Calendar → Network Connection	1	0	0	1	4
Call Log & Contacts → Log Output	6	0	3	9	693
Call Log & Contacts → Network Connection	2	0	2	4	233
Call Log & Contacts → Public Files	2	0	0	2	52
Call Log & Contacts → SMS Messages	0	0	1	1	4
Camera → Log Output	1	0	0	1	3
Camera → Network Connection	1	0	0	1	1
Camera → Public Files	1	0	0	1	11
History & Bookmarks → Log Output	1	0	0	1	6
History & Bookmarks → Network Connection	0	0	1	1	4
Location → Log Output	10	2	3	15	1,321
Location → Network Connection	7	3	2	12	1,002
Location → Public Files	0	0	1	1	33
Location → SMS Messages	1	0	0	1	18
Microphone → Log Output	0	1	0	1	32
Microphone → Network Connection	0	1	0	1	11
Microphone → Public Files	1	0	0	1	2
Phone State → Log Output	10	1	1	12	1,056
Phone State → Network Connection	9	1	2	12	996
Phone State → Public Files	0	0	2	2	66
Phone State → SMS Messages	1	0	0	1	13
SMS & MMS → Log Output	0	0	1	1	5
SMS & MMS → Network Connection	0	0	1	1	2
User Dictionary → Log Output	1	0	0	1	2
WiFi State → Log Output	2	1	1	4	245
WiFi State → Network Connection	2	0	0	2	132
WiFi State → Public Files	1	0	0	1	9
Total	65	10	25	100	6,420

Table 4.12: Verified flows by source and sink types.

updating widgets on the home screen, this application dumps calendar information to the log. Specifically, it loops over the events in the calendar and writes the name of each event as well as its start and end timestamps to the system log. This is presumably for debugging purposes, and

we found that the developer added an option to allow users to enable or disable such debug messages in the latest version.

During our analysis, we noticed a few common patterns in flows to the system log. First, flows to the log sometimes precede flows of the same information to the network. Before making a HTTP request, some applications write the full URL to the log, likely for debugging purposes. This means sensitive data that is exfiltrated over the network is often leaked to the log as well. We also discovered that some write operations to the log are guarded by a `Boolean` value, which serves as a debugging switch. Developers may set it to `true` during development and testing, and flip the switch when exporting a release build. In some cases, applications offer an option to allow users to enable or disable debugging output.

Network Connection A major means of exfiltrating sensitive information in applications is leaking to network sinks. In one application, *Ant Smasher Free Game*, Heimdall found several flows of account information to the network. During gameplay, the application retrieves the user's primary account name (*e.g.*, Google account) from the `AccountManager` and exfiltrates it by sending two network requests to the developer's server at `androidsdk.bestcoolfungames.com`. To trigger and confirm this behavior, we set up a wireless access point on a laptop and tethered its network connection to our Android phone. When we started the game, we captured and monitored network traffic using Wireshark[5]. The primary account name, which is our Gmail address in this case, appeared in two HTTP GET requests sent to the developers. The account name may be used as a unique user identifier, but this tracking mechanism is uncommon and clearly constitutes a privacy violation. We also examined the latest version of this game and found that the account name, along with the device identifier and MAC address, is sent to the developer's server at `android.bcfads.com` via HTTPS. We observed the transmitted data with Wireshark, but we were not able to decipher the encrypted payload.

Heimdall detected a few leaks to the network in *SoundHound*, a popular music tagging

[5]http://www.wireshark.org/

application. Although it also transmits data over HTTPS, sensitive data is sent in cleartext with request headers. We found that it includes the user's geolocation (*i.e.*, latitude and longitude) and the first six digits of the subscriber ID (*i.e.*, MCC and MNC) in a custom user agent string, which is sent with each tagging request to the server at `api.midomi.com`. We confirmed this tracking behavior by monitoring the network traffic while using this application to identify music.

As Table 4.12 shows, phone state and location data is frequently collected and exfiltrated through the network. We observe that such data is often sent to advertising and analytics servers. While most libraries (*e.g.*, AdMob and Mobclix) send sensitive data in plaintext, one library (*i.e.*, Flurry) encodes and transmits data in binary format. Specifically, it sends reports to the server at `http://data.flurry.com/aap.do` and includes location data in the HTTP POST body. Thousands of applications integrate the Flurry library, and we have confirmed this exfiltration behavior by monitoring network traffic with one such application.

Public Files Heimdall detected a flow of phone state data to the external storage in the *Talking Tom Cat 2 Free* application. Manual examination revealed that the `onCreate` method of the application's `Main` activity class calls another method, which retrieves the unique device identifier, writes it to a file on the device's external storage, and returns the identifier to the caller. We confirmed this privacy leak by installing and running this application on our Android device. We observed that a file named `.udid` was created under the `Android` directory on the SD card immediately after the application started. Just as expected, this file contains exactly our unique device identifier. This flow is clearly a privacy leak, as any other application may access the protected device identifier by reading the external storage without the `READ_PHONE_STATE` permission.

Heimdall found a flow to the external storage in *Pudding Camera*, a third-party camera application. After taking each photo, a file named `temp_img.jpg` is saved under the `/PuddingCamera/.BTH` directory of the external storage. We were also able to confirm this behavior by triggering it on our Android device. While this file is overwritten every time a photo

is taken, a scaled-down photo is saved under the `PuddingCamera` directory, whose file name is based on the timestamp of the photo. We have also tested the latest version, which has similar behavior, but both the thumbnail and original photo share the same timestamp-based file name. Note that the stock camera application that is pre-installed on Android devices also saves photos on the external storage. We believe this is because that media data tend to be large in size and devices generally offer more limited internal storage space for application data. To mitigate this issue, Google recently introduced a new `READ_EXTERNAL_STORAGE` permission in Android 4.1 to better control read accesses to external storage. However, this permission is not enforced by default, and it does not apply to devices running older versions of the system.

SMS Messages Heimdall only detected a few flows of sensitive information to the SMS sink. For instance, *Wheres My Droid*, an application designed to help users find their Android devices using text messages, contains flows of location data to SMS messages. To verify the flows, we installed this application on a test Android phone P and sent it an SMS message containing the attention words (*e.g.*, "GPS My Droid") from another phone P'. We received a confirmation message[6] on P' immediately and three additional messages in a few minutes. The second message contains the location of P (*i.e.*, latitude and longitude); the third message is a link to Google Maps showing that location; and the last message provides the nearest street address for that location. This behavior can also be triggered in the latest version of this application with a different default attention word (*i.e.* "wmd gps"). Given its core functionality of locating lost or stolen devices, we believe the sensitive data transmission via SMS messages is legitimate and expected.

Heimdall reported similar flows of phone and location data to the SMS sink in a few other applications. Although we did not verify all such flows, we noticed that most of these applications offer the anti-theft functionality. Therefore, we believe that the top applications do not maliciously exfiltrate sensitive data through SMS messages.

[6] "WMD: Attention word received, phone is on and GPS has been started. Please wait up to 5 mins for a response."

4.4.5 Discussions

While our work focuses on sensitive data on the system level, we do not track sensitive information that is specific to an application. For instance, users' Skype contact and profile information is also private and should not be leaked to an outside entity. However, to handle such cases, we need application-specific knowledge to identify taint sources. Although this process may be largely manual, some heuristics may help automate the process. For example, we may consider data guarded by a login process as potentially sensitive. How to automatically identify application-specific sensitive data in general warrants further research and we plan to explore this area in the future.

In our evaluation, we consider four types of potentially unsafe sinks. However, malicious applications may still leak privacy-sensitive information through API methods that we consider safe. For instance, an application can write sensitive information to its private preferences or UI elements, retrieve what is stored there later, and send it to unsafe sinks. To alleviate this issue, we may model additional APIs in the Android framework and track the flows as necessary.

Currently, Heimdall cannot track the flow of information through inter-component communications. Specifically, we do not propagate taint from one component sending an intent to another component that receives the intent. This means we only analyze flows within the same application component. Since an intent is a bundle containing various information, tainting the whole intent may lead to excessive false positives. In addition, intents allow dynamic target resolution and runtime binding, and this dynamic nature poses additional challenge for our static analysis. We leave supporting intent-based information flows for future work.

As most Android applications are written in Java, Heimdall does not analyze native code. We have found that only 5.2% of the applications we analyzed include and load libraries written in native code. For applications that do handle sensitive information using native code, we are still able to identify the taint sources in most cases, as private data is accessed using Android's Java APIs. In this case, it is possible to integrate existing binary analysis techniques [25, 90] to further track the information flows in native code.

4.5 Related Work

Privacy Leak Detection As privacy leaks become a growing concern among smartphone users, researchers have proposed various techniques to detect such leaks. PiOS [39], a static analysis tool, analyzes iOS applications to detect leaks of sensitive information through reachability analysis on control-flow graphs. It has found that a majority of applications leak unique device IDs to third parties. SCanDroid [49] statically reasons about the security of applications using modular dataflow analysis on Java source code. It can be used for automated security certification, although the analysis has not been applied to real-world applications. AndroidLeaks [52], another static analysis framework, finds potential leaks of sensitive information in Android applications. It uncovers various leaks from a large collection of 24,350 applications. On a smaller scale, ScanDal [64] considers multiple pre-defined taint sources and sinks, and detects privacy leaks in 98 Android applications. Similarly, Mann and Starostin [68] identify taint sources and sinks by studying Android APIs, but their framework has not been evaluation on real-world applications. Stevens *et al.* [91] also look for privacy vulnerabilities in Android applications, but with a focus on popular advertising libraries. Besides static techniques, dynamic approaches have also been proposed to detect privacy leaks. TaintDroid [41] monitors the flow of sensitive data through system-wide dynamic taint tracking. When sensitive data leaves the system via third-party applications, TaintDroid warns the user and logs the suspicious flow which can be audited by the user or security analysts. In contrast to previous work that focus on network-based sinks, our work aims to understand information leakage comprehensively and provides insights into how third-party applications handle and use private data.

Privacy Leak Prevention A number of techniques have been proposed to mitigate information leakage on smartphones. MockDroid [22] protects privacy by replacing sensitive user information with mock data, while still allowing applications to run, possibly with reduced functionalities. TISSA [97] provides users fine-grained control and runtime adjustment capability

over what kinds of sensitive user information will be available to untrusted applications and how that information is accessed (*e.g.*, anonymized, empty, mock). Nauman *et al.* [73] propose Apex, a policy enforcement framework that allows users to selectively grant permissions to Android applications and impose runtime resource access constraints. AppFence [59], a follow-up work to TaintDroid, retrofits the Android system to impose privacy controls on third-party applications. It allows users to provide mock data to permission-hungry applications and blocks sensitive data obtained by applications from leaving the device. While our work does not directly prevent leaks, we believe that a comprehensive understanding of how applications use sensitive data will enable researchers to design and improve systems for better thwarting privacy leaks.

Android Permissions and Security There have been extensive studies on Android's permission system and its close connections to security and privacy. Kirin [40] is a lightweight security service that checks permissions requested by applications against a set of security rules to look for suspicious permission combinations at install time. Felt *et al.* [45] leverage automated testing techniques to uncover the permissions required by each API method. They developed a tool, Stoaway, to analyze permission usage in 940 Android applications and found that about one third of them are overprivileged. Enck *et al.* [42] implement the ded decompiler for Android applications to look for dangerous functionalities and vulnerabilities. Using ded, they studied the decompiled source code of 1,100 popular Android applications and found pervasive exfiltration of phone identifiers and location information. Jeon *et al.* [61] propose a finer-grained permission system for stock Android devices. They developed a tool that rewrites applications to access sensitive resources via a dedicated service that enforces finer-grained permissions.

4.6 Conclusion

With the increased popularity of smartphones and tablets, the number of third-party applications is rapidly growing. To implement appealing features, such applications often access a wide range

of sensitive data, which has led to increased privacy concerns among users. This chapter has presented the first comprehensive study on sensitive information usage in Android applications. Specifically, we aim to understand the overall privacy threats they pose to user data. To this end, we have designed and implemented Heimdall, a static analysis framework, to automatically track the flow of sensitive information and detect dataflows to unsafe sinks. We have used Heimdall to study the behavior of 122,570 unique Android applications downloaded from the Google Play Store. Our results have shown that a large number of applications either intentionally exfiltrate sensitive data or inadvertently leak such data through unsafe sinks. We believe that our findings can benefit both end-users and security professionals.

Chapter 5

Conclusion

This dissertation has presented techniques and tools for understanding the emerging, large, and important software ecosystem of Android applications.

We have presented the first large-scale, systematic study of Android applications. From the coding and development perspective, we have examined unique aspects of Android applications and uncovered a number of common errors and pitfalls. Leveraging our findings, platform developers (*i.e.*, Google) may prioritize their efforts to improve frequently used APIs, clarify platform documentation, and implement security enhancements. With deeper understanding of applications, tool developers may build better tool support, such as testing, analysis, and development environments, to assist application developers. Researchers and analysts should address challenges of Java reflection and code obfuscation when developing automated tools. Application developers can learn from uncovered programming errors and pitfalls so that they can avoid similar mistakes. We believe that these actionable recommendations, along with raised awareness among developers and users, benefit the entire Android community.

Focusing on security and privacy, we have also presented the first comprehensive study on sensitive information usage in Android applications. Our study aims to understand the overall privacy threats to privacy-sensitive user data from third-party Android applications. To this end, we have designed and implemented a static analysis framework to detect unsafe information

flows. We have found that a large number of applications either intentionally exfiltrate sensitive data or inadvertently leak such data through unsafe sinks. To better protect user privacy, we suggest application developers exercise caution when handling sensitive user data to avoid unintentional leaks. We also believe that such a system may be deployed to automatically analyze third-party applications before they are released and published.

This dissertation focuses on understanding third-party Android applications. While mobile applications continue to become increasingly popular and more feature-rich, there are many outstanding open problems in this area. We highlight and discuss a few interesting future directions below.

Inter-application Interactions and Communications The Android operating system provides a rich, powerful inter-application messaging mechanism that promotes inter-application collaboration and component reuse. Application developers may leverage existing functionalities provided by other applications while still giving users the impression of a single, seamless experience. For instance, instead of implementing its own layout and rendering engine, an application may direct users to a dedicated browser application to navigate a Web page. Researchers have started examining such application interactions from a security perspective to identify security risks [33], but there is still a lack of general understanding. Studying the interactions and communications between applications will provide valuable insights into characteristics of the Android platform and enable comprehensive understanding of application behavior.

Malicious Application Detection With the rapid adoption of smart mobile devices, mobile malware has been on the rise [44, 67, 69] and will both evolve in technique and grow in volume [24, 55]. In Chapter 4, we have touched upon malicious applications that exfiltrate sensitive information. However, there are other types of malicious applications (*e.g.*, toll fraud malware, ransomware) that also pose significant security threats. To tackle this problem, researchers have taken first steps toward characterizing and understanding existing mobile malware [46, 96]. While various techniques have been developed to detect traditional malware,

new challenges must be addressed to mitigate mobile malware threats, as some of the attack vectors are unique to mobile platforms.

Mobile Applications on Other Platforms In this dissertation, we have focused on understanding mobile applications on the Android platform. However, there are other popular and burgeoning mobile platforms, such as Apple's iOS and Microsoft Windows Phone. We believe that similar methodologies can be adopted and extended to analyze and understand applications on those platforms. Additional efforts in this research area may also provide insights into the similarities and differences among difference mobile platforms.

A fundamental goal of understanding mobile applications is to improve the quality of such applications as well as the platform, facilitate the development and testing processes, and promote best security practices for the mobile software ecosystem. As the landscape of mobile computing continues to develop and evolve, we hope that the approaches and techniques presented here will be more broadly applied.

Appendix A

List of Common External Libraries

In Chapter 3, we listed the top 10 external libraries used in third-party Android applications. We believe that this knowledge of external library usage can benefit researchers and analysts studying Android applications. For completeness, the top 100 mostly used libraries are listed in Table A.1, which continues for multiple pages.

Library	Description	#apps
com.google.ads	AdSense/AdMob	39,462
com.google.android.maps	Google Maps API	23,345
org.apache.commons.logging	Apache Commons Logging	19,762
com.facebook.android	Facebook API	19,730
org.slf4j	SLF4J (logging)	17,718
twitter4j	Twitter API	17,020
org.apache.log4j	Apache log4j	16,680
org.apache.commons.codec	Apache Commons Codec	16,538
com.admob.android.ads	AdMob	11,125
com.google.android.apps.analytics	Google Analytics	10,762
oauth.signpost	Simple OAuth	9,880
com.google.gdata	Google Data APIs	9,698
com.millennialmedia.android	Millennial Media	9,489
com.android.vending.billing	In-app Billing	8,070
org.codehaus.jackson	Jackson JSON Processor	7,795
org.joda.time	Joda Time API	7,149
com.airpush.android	AirPush	7,056
com.flurry.android	Flurry	6,619

Library	Description	#apps
org.mortbay.jetty	Jetty Server	6,464
com.inmobi.androidsdk	InMobi	6,363
com.nikkoaiello.mobile.android	Android Pinch	6,349
com.google.common	Google Collections Library	5,989
com.google.protobuf	Protocol Buffers	5,703
com.urbanairship	Urban Airship	5,574
com.zestadz.android	ZestADZ	5,568
com.readystatesoftware	MapView Balloons	5,310
com.mdotm.android.ads	MdotM	5,242
com.google.gson	Gson	4,976
com.adwhirl	AdWhirl	4,923
com.qwapi.adclient.android	Quattro	4,667
com.mobclix.android	Mobclix	4,470
com.google.android.c2dm	Android C2DM	3,746
org.apache.james	Apache James	3,396
com.oneriot	OneRiot	3,317
com.phonegap	PhoneGap	3,066
com.wooboo.adlib_android	Wooboo	2,841
jp.Adlantis.Android	Atlantis	2,708
net.youmi	Youmi	2,426
com.mobclick.android	Umeng	2,399
com.greystripe.android.sdk	Greystripe	2,397
com.mobfox.sdk	MobFox	2,340
com.madhouse.android.ads	Madhouse	2,199
cn.domob.android	DOMOB	2,160
com.google.zxing	Zebra Crossing	2,017
com.casee	CASEE	1,994
com.smaato.SOMA	Smaato	1,966
com.adwo.adsdk	Adwo	1,958
com.adchina.android	AdChina	1,928
com.vpon	Vpon	1,828
org.apache.commons.lang	Apache Commons Lang	1,815
jp.co.nobot	Nobot	1,759
org.acra	ACRA	1,740
com.energysource.szj	AdTouch	1,739
org.jaxen	Jaxen	1,730
com.android.vending.licensing	Google Play Licensing	1,613
com.j256.ormlite	ORMLite	1,602
com.Leadbolt	LeadBolt	1,589
com.mt.airad	airAD	1,493
kankan.wheel	Android Wheel Widget	1,415
com.waps	WAPS	1,391

Library	Description	#apps
com.baidu	Baidu	1,369
org.apache.avalon.framework	Apache Avalon	1,350
org.apache.log	Avalon LogKit	1,347
com.scoreloop.client.android	Scoreloop	1,245
org.jsoup	Java HTML Parser	1,218
org.kxml2	kXML	1,154
org.simpleframework	Simple Framework	1,123
org.apache.commons.httpclient	Apache HttpClient	1,106
com.admarvel.android	AdMarvel	1,077
org.appcelerator	Appcelerator	1,067
com.nullwire.trace	Android Remote Stacktrace	1,066
com.google.devtools.simple	Simple	1,053
gnu.*/kawa.*	GNU Kawa	1,050
com.google.youngandroid	App Inventor for Android	1,049
com.l.adlib_android	LSense	1,025
com.tapjoy	Tapjoy	1,005
org.scribe	Scribe	978
com.thoughtworks	Selenium Java Client Driver	955
com.jumptap.adtag	Jumptap	938
com.vdopia.client.android	iVdopia	936
com.winad.android	WinAds	930
winterwell.jtwitter	JTwitter	924
lgpl.haustein	Campyre	920
com.badlogic	Badlogic	916
org.jdom	JDOM	913
org.apache.harmony	Apache Harmony	912
com.openfeint	OpenFeint	894
org.apache.commons.io	Apache Commons IO	856
net.jcip.annotations	Java Concurrency in Practice	845
com.Localytics.android	Localytics	835
com.guohead.sdk	GuoheAd	835
org.kobjects	kObjects	827
com.adserver.adview	mOcean	814
org.ksoap2	KSoap	803
com.smartadserver.library	Smart AdServer	782
com.unity3d	Unity	780
jp.co.imobile.android	i-mobile	742
com.innerActive.ads	inneractive	738
com.skyd	SkypeKit SDK	728
com.admogo	AdmOgo	726

Table A.1: Top 100 external libraries.

Appendix B

List of Sensitive Flows by Type

In Chapter 4, we used Heimdall to analyze the flow of sensitive information in 122,570 Android applications from the Google Play Store. Among all the flows, we have observed 42 different combinations of source and sink types. Table B.1 shows all 42 types of flows, ordered by the number of flows for each type.

Source Type	**Sink Type**	**#flows**
Phone State	Network Connection	96,198
Location	Log Output	95,690
Location	Network Connection	67,852
Phone State	Log Output	59,580
Call Log & Contacts	Log Output	7,358
WiFi State	Log Output	4,167
WiFi State	Network Connection	3,631
Accounts	Log Output	1,752
Accounts	Network Connection	1,588
Call Log & Contacts	Network Connection	1,561
Location	Public Files	1,315
Phone State	Public Files	889
Camera	Public Files	733
SMS & MMS	Log Output	541
Call Log & Contacts	Public Files	461
History & Bookmarks	Log Output	432
Location	SMS Messages	425
Phone State	SMS Messages	262

Source Type	Sink Type	#flows
SMS & MMS	Network Connection	175
Calendar	Log Output	157
Microphone	Public Files	148
WiFi State	Public Files	139
Camera	Log Output	133
Accounts	Public Files	126
Call Log & Contacts	SMS Messages	123
Microphone	Log Output	107
Camera	Network Connection	89
SMS & MMS	SMS Messages	67
History & Bookmarks	Network Connection	54
SMS & MMS	Public Files	53
Calendar	Public Files	41
History & Bookmarks	Public Files	38
Microphone	Network Connection	28
Calendar	Network Connection	16
Accounts	SMS Messages	12
WiFi State	SMS Messages	7
User Dictionary	Log Output	6
Microphone	SMS Messages	4
History & Bookmarks	SMS Messages	2
User Dictionary	Public Files	2
Calendar	SMS Messages	2
Tasks	Log Output	1

Table B.1: Flows by Source and Sink Types.

Appendix C

List of Reviewed Android Applications

To evaluate the effectiveness of Heimdall, we systematically selected and manually reviewed 100 flows from 88 popular Android applications from Google Play. These 88 applications are listed in Table C.1, which spans across several pages.

App Name	Package	Version
Angry Birds Rio	com.rovio.angrybirdsrio	1.4.0
Basketball Shots 3D	com.sas.basketball	1.9.1
BIG TIME GANGSTA	com.glu.android.brawler	2.0.1
Bluetooth File Transfer	it.medieval.blueftp	5
Brightest Flashlight Free™	goldenshorestechnologies- .brightestflashlight.free	1.9.3
Bunny Shooter - Best Free Game	com.bestcoolfungamesfreegame- appcreation.bunnyshooter	1.3.0
Call Control - Call Blocker	com.flexaspect.android- .everycallcontrol	3.0.2
Cardiograph	com.macropinch.hydra.android	2.6
chompSMS	com.p1.chompsms	5.26
Cleopatra's Pyramid	com.gameduell.cleopatraspyramid	1.4
Coin Pirates	com.nubee.coinpirates	1.0.10
Crazy Snowboard	com.ezone.Snowboard	1.1.3
Digitally Imported Radio	com.audioaddict.di	0.9.17
Dropbox	com.dropbox.android	1.2.4
Family Locator	com.life360.android.safetymapa	5.0.2 build 2795
Fashion Story™	com.teamlava.fashionstory	1.0.6

App Name	Package	Version
Foursquare	com.joelapenna.foursquared	2011.11.18
FriendCaster for Facebook	uk.co.senab.blueNotifyFree	4.1.2
GameFly	com.gamefly.android.gamecenter	2.0.5
Gmail	com.google.android.gm	2.3.5.2
GO SMS Pro	com.jb.gosms	3.78
Goal.com Mobile - Football	com.handmark.mpp.goal	3.1.3.1
Google Books	com.google.android.apps.books	1.4.6
Google Korean IME	com.google.android.inputmethod-.korean	0.78
Google Music	com.google.android.music	4.0.9
Grindr - Gay guy finder	com.grindrapp.android	1.1.7
Handcent SMS	com.handcent.nextsms	3.9.9.8
HeyTell	com.heytell	2.0.4.256
HeyWire - Free Texting	com.mediafriends.chime	2.7.12
Horoscope	fr.telemaque.horoscope	2.0.2
ICQ messenger - free chat	com.icq.mobile.client	3.0.1
K-9 Mail	com.fsck.k9	3.802
KakaoTalk Messenger	com.kakao.talk	2.6.0
KAYAK Flight Hotel Car Search	com.kayak.android	4.2.0.3
LINE	jp.naver.line.android	1.6.2
McAfee WaveSecure (Trial)	com.wsandroid	4.2.5.199
Meet New People on Badoo	com.badoo.mobile	1.7.0
MixZing Media Player	com.mixzing.basic	3.6.1
Moon Chaser	com.reverie.game.ninja	1.1.1
Mr. Number	com.mrnumber.blocker	1.1.14
My Country	com.gameinsight.mycountry	1.12
NinJump	com.bfs.ninjump	1.03
Norton Security and Antivirus	com.symantec.mobilesecurity	2.5.0.384
OpenSignalMaps	com.staircase3.opensignal	1.13
Paradise Island	com.seventeenbullets.android-.island	1.1.21
PhoneFlicks-Netflix Android	com.nextmobileweb.phoneflix	1.4
PhotoWonder	cn.jingling.motu.photowonder	1.1.2
Police Scanner Radio Scanner	com.berobo.android.scanner	7
Pool Master Pro	com.forthblue.pool	1.8
Reckless Racing Lite	com.polarbit.RecklessRacingLite	1.0.3
Retro Camera	org.urbian.android.tools-.vintagecam	3.71
Ringtone Maker	com.mobile17.maketones-.android.free	1.1.4
Sex Offender Search	com.life360.android.safetymapc	5.0.2 build 2795
ShopSavvy Barcode Scanner	com.biggu.shopsavvy	5.0.1

App Name	Package	Version
SKOUT - Meet, Chat, & Flirt	com.skout.android	2.8.1
Skype	com.skype.raider	2.5.0.160
Slot Machine	com.apostek.SlotMachine	5.7.1
Smart App Protector(app lock)	com.sp.protector.free	3.9.1
Smooth Calendar	se.catharsis.android.calendar	1.0.0.2
SMS Backup	tv.studer.smssync	1.1.1
SoundHound	com.melodis.midomiMusic-Identifier.freemium	2.8.4
Speed Skater	com.incrementalsoft.SkaterIsBack	2.0.1
T-Mobile Video Chat by Qik	com.qiktmobile.android	0.11.62
Talking Gina the Giraffe Free	com.outfit7.talkinggina	1.2
Talking Santa Free	com.outfit7.talkingsantafree	2.0.1
Tap Tap Revenge 4	com.tapulous.taptaprevenge4	4.1.3 (807)
TeamViewer for Remote Control	com.teamviewer.teamviewer-.market.mobile	7.0.238
Toddler Lock	marcone.toddlerlock	2.7.5
Trapster	com.trapster.android	3.0.3
TripAdvisor	com.tripadvisor.tripadvisor	1.3.3
TweetCaster for Twitter	com.handmark.tweetcaster	5.2
UberSocial for Android	com.twidroid	7.2.2
UC Browser	com.uc.browser	7.9.3
Uloops Studio Lite	net.uloops.android	2.11.2
USAA Mobile	com.usaa.mobile.android.usaa	4.1
VKontakte	com.vkontakte.android	2.0.1
Vlingo Virtual Assistant	com.vlingo.client	3.3
Wheres My Droid	com.alienmanfc6.wheres-myandroid	3.5.0
Whoopee Cushion! (fart)	com.foncannoninc.wc	1.52
Words With Friends Free	com.zynga.words	4.56
World Newspapers	com.world.newspapers	2.2
Yahoo! Fantasy Football '11	com.yahoo.mobile.client.android-.fantasyfootball	2.2.0
Yandex.Maps	ru.yandex.yandexmaps	2.11
YP Yellow Pages & Gas Prices	com.yellowpages.android-.ypmobile	3.3.2
ZombieBooth	com.motionportrait.ZombieBooth	3.05
Mobile QQ2011	com.tencent.qq	2011
Daum	net.daum.android.daum	2.8.5
TicToc	kr.co.tictocplus	1.6.3

Table C.1: Manually reviewed Android applications.

Bibliography

[1] The AndroidManifest.xml File. http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[2] Application Fundamentals. http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[3] Dalvik Virtual Machine. http://code.google.com/p/dalvik/.

[4] Dalvik Executable Format. http://source.android.com/tech/dalvik/dex-format.html.

[5] Issue 19192: documentation error - ACCESS_COARSE_UPDATES is not a real permission. https://code.google.com/p/android/issues/detail?id=19192.

[6] ADT Plugin. http://developer.android.com/tools/sdk/eclipse-adt.html.

[7] Filters on Google Play. http://developer.android.com/guide/google/play/filters.html.

[8] Intents and Intent Filters. http://developer.android.com/guide/components/intents-filters.html.

[9] Android Lint. http://tools.android.com/tips/lint/.

[10] Lint API Check. http://tools.android.com/recent/lintapicheck.

[11] Manifest Permissions. http://developer.android.com/reference/android/Manifest.permission.html.

[12] Android NDK. http://developer.android.com/tools/sdk/ndk/overview.html.

[13] Android Permissions. http://developer.android.com/guide/topics/security/permissions.html.

[14] apktool. http://code.google.com/p/android-apktool/.

[15] AXMLPrinter2. http://code.google.com/p/android4me/.

[16] ProGuard. http://proguard.sourceforge.net/.

[17] undx. http://undx.sourceforge.net/.

[18] Aaron Tilton. Vlingo Privacy Breach: Data Sent to Remote Servers Without Consent. http://www.androidpit.com/Vlingo-security-flaw, January 2012.

[19] Aaron Tilton. Vlingo Makes Official Statement: Success At Customers' Expense. http://www.androidpit.com/vlingo-user-data-leak, January 2012.

[20] Apple Inc. Apple's App Store Marks Historic 50 Billionth Download. http://www.apple.com/pr/library/2013/05/16Apples-App-Store-Marks-Historic-50-Billionth-Download.html, May 2013.

[21] David Barrera, H. Gunes Kayacik, P.C. van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 73–84, 2010. ISBN 978-1-4503-0245-6. doi: {http://dx.doi.org/10.1145/1866307.1866317}.

[22] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *HotMobile '11:*

Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, pages 49–54, 2011. ISBN 978-1-4503-0649-2. doi: {http://dx.doi.org/10.1145/2184489.2184500}.

[23] Matthias Bohmer, Brent Hecht, Johannes Schoning, Antonio Kruger, and Gernot Bauer. Falling Asleep with Angry Birds, Facebook and Kindle – A Large Scale Study on Mobile Application Usage. In *MobileHCI '11: Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 47–56, 2011. ISBN 978-1-4503-0541-9. doi: {http://dx.doi.org/10.1145/2037373.2037383}.

[24] Sebastian Bortnik. Trends for 2013: Astounding Growth of Mobile Malware. http://blog.eset.com/2012/12/11/trends-for-2013-astounding-growth-of-mobile-malware, December 2012.

[25] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *CAV '11: Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469, 2011. ISBN 978-3-642-22109-5. doi: {http://dx.doi.org/10.1007/978-3-642-22110-1_37}.

[26] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS '12: Proceedings of the 19th Annual Network and Distributed System Security Conference*, 2012.

[27] Canalys. Smart phones overtake client PCs in 2011. http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011, February 2012.

[28] Canalys. Smart mobile device shipments exceed 300 million in Q1 2013. http://www.canalys.com/newsroom/smart-mobile-device-shipments-exceed-300-million-q1-2013, May 2013.

[29] Rich Cannings. Exercising Our Remote Application Removal Feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html, June 2010.

[30] Justin Case. Exclusive: Vulnerability In Skype For Android Is Exposing Your Name, Phone Number, Chat Logs, And A Lot More. http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/, April 2011.

[31] Patrick P.F. Chan, Lucas C.K. Hui, and S.M. Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *WISEC '12: Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136, 2012. ISBN 978-1-4503-1265-3. doi: {http://dx.doi.org/10.1145/2185448.2185466}.

[32] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *WWW '12: Proceedings of the 21st International Conference on World Wide Web*, pages 311–320, 2012. ISBN 978-1-4503-1229-5. doi: {http://dx.doi.org/10.1145/2187836.2187879}.

[33] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys '11: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011. ISBN 978-1-4503-0643-0. doi: {http://dx.doi.org/10.1145/1999995.2000018}.

[34] comScore. comScore Reports March 2013 U.S. Smartphone Subscriber Market Share. http://www.comscore.com/Insights/Press_Releases/2013/5/comScore_Reports_March_2013_U.S._Smartphone_Subscriber_Market_Share, May 2013.

[35] Josh Constine. Mary Meeker Gives Mid-Year Internet Trends Report: Android Adoption Ramping Up 6X Faster Than iPhone. http://techcrunch.com/2012/11/05/mary-meeker-internet-trends/, November 2012.

[36] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *ESORICS '12: Proceedings of the 17th European Symposium on Research in Computer Security*, pages 37–54, 2012. ISBN 978-3-642-33166-4. doi: {http://dx.doi.org/10.1007/978-3-642-33167-1_3}.

[37] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *MoST '12: Proceedings of IEEE Mobile Security Technologies*, 2012.

[38] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *SEC '11: Proceedings of the 20th USENIX Conference on Security*, 2011.

[39] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS '11: Proceedings of the 18th Annual Network and Distributed System Security Conference*, 2011.

[40] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 235–245, 2009. ISBN 978-1-60558-894-0. doi: {http://dx.doi.org/10.1145/1653662.1653691}.

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI '10: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[42] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *SEC '11: Proceedings of the 20th USENIX Conference on Security*, 2011.

[43] Darrell Etherington. Have Just A Phone Number? iOS 6 Facebook Integration Can Fill In The Blanks. http://techcrunch.com/2012/09/27/have-just-a-phone-number-ios-6-facebook-integration-can-fill-in-the-blanks/, September 2012.

[44] *Mobile Threat Report Q4 2012*. F-Secure Labs, March 2013.

[45] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *CCS '11: Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638, 2011. ISBN 978-1-4503-0948-6. doi: {http://dx.doi.org/10.1145/2046707.2046779}.

[46] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steven Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *SPSM '11: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2011. ISBN 978-1-4503-1000-0. doi: {http://dx.doi.org/10.1145/2046614.2046618}.

[47] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The Effectiveness of Application Permissions. In *WebApps '11: Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[48] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *SEC '11: Proceedings of the 20th USENIX Conference on Security*, 2011.

[49] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, 2009.

[50] Dan Galpin, Ilya Firman, Andy Stadler, Michael Siliski, and Ellie Powers. Android Apps Break the 50MB Barrier. http://android-developers.blogspot.com/2012/03/android-apps-break-50mb-barrier.html, March 2012.

[51] Gartner. Gartner Says Asia/Pacific Led Worldwide Mobile Phone Sales to Growth in First Quarter of 2013. http://www.gartner.com/newsroom/id/2482816, May 2013.

[52] Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *TRUST '12: Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, pages 291–307, 2012. ISBN 978-3-642-30920-5. doi: {http://dx.doi.org/10.1007/978-3-642-30921-2_17}.

[53] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS '12: Proceedings of the 19th Annual Network and Distributed System Security Conference*, 2012.

[54] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *MobiSys '12: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 281–294, 2012. ISBN 978-1-4503-1301-8. doi: {http://dx.doi.org/10.1145/2307636.2307663}.

[55] Derek Halliday. 2013 Mobile Threat Predictions. https://blog.lookout.com/blog/2012/12/13/2013-mobile-threat-predictions/, December 2012.

[56] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22:36–38, 1988.

[57] Kamala D. Harris. Attorney General Kamala D. Harris Secures Global Agreement to Strengthen Privacy Protections for Users of Mobile Applications. http://oag.ca.gov/news/press-releases/attorney-general-kamala-d-harris-secures-global-agreement-strengthen-privacy, February 2012.

[58] Kamala D. Harris. Attorney General Kamala D. Harris Notifies Mobile App Developers of Non-Compliance with California Privacy Law. http://oag.ca.gov/news/press-

releases/attorney-general-kamala-d-harris-notifies-mobile-app-developers-non-compliance, October 2012.

[59] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. "These Aren't the Droids You're Looking For": Retroffiting Android to Protect Data from Imperious Applications. In *CCS '11: Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 639–652, 2011. ISBN 978-1-4503-0948-6. doi: {http://dx.doi.org/10.1145/2046707.2046780}.

[60] IDC. Android and iOS Combine for 92.3% of All Smartphone Operating System Shipments in the First Quarter While Windows Phone Leapfrogs BlackBerry, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS24108913, May 2013.

[61] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *SPSM '12: Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2012. ISBN 978-1-4503-1666-8. doi: {http://dx.doi.org/10.1145/2381934.2381938}.

[62] Trevor Johns. Securing Android LVL Applications. http://android-developers.blogspot.com/2010/09/securing-android-lvl-applications.html, September 2010.

[63] Simon Khalaf. Mobile Apps: We Interrupt This Broadcast. http://blog.flurry.com/bid/92105/Mobile-Apps-We-Interrupt-This-Broadcast, December 2012.

[64] Jinyung Kim, Yongho Yoon, and Kwangkeun Yi. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *MoST '12: Proceedings of IEEE Mobile Security Technologies*, 2012.

[65] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These Are Not the Permissions You Are Looking For. In *DefCon 18*, 2010.

[66] Hiroshi Lockheimer. Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, February 2012.

[67] *State of Mobile Security 2012*. Lookout Mobile Security, 1 Front Street, Suite 2700, San Francisco, CA 94111, September 2012.

[68] Christopher Mann and Artem Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462, 2012. ISBN 978-1-4503-0857-1. doi: {http://dx.doi.org/10.1145/2245276.2232009}.

[69] *McAfee Threats Report: Third Quarter 2012*. McAfee Labs, 2821 Mission College Boulevard, Santa Clara, CA 95054, November 2012.

[70] Darren McCarra. Google Play will hit one million apps this June. http://sociable.co/mobile/google-play-will-hit-one-billion-apps-this-june/, January 2013.

[71] Andy McFadden. Backward compatibility for Android applications. http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html, April 2009.

[72] Anders Møller and Mathias Schwarz. Automated Detection of Client-State Manipulation Vulnerabilities. In *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*, pages 749–759, 2012. ISBN 978-1-4673-1066-6. doi: {http://dx.doi.org/10.1109/ICSE.2012.6227143}.

[73] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ASIACCS '10: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, 2010. ISBN 978-1-60558-936-7. doi: {http://dx.doi.org/10.1145/1755688.1755732}.

[74] Nielsen. State of the Appnation - A Year of Change and Growth in U.S. Smartphones. http://blog.nielsen.com/nielsenwire/?p=31891, May 2012.

[75] Nielsen. Two Thirds of New Mobile Buyers Now Opting For Smartphones. http://blog.nielsen.com/nielsenwire/?p=32494, July 2012.

[76] Jon Oberheide and Charlie Miller. Dissecting Android's Bouncer. In *SummerCon 2012*, 2012.

[77] Damien Octeau, Patrick McDaniel, and William Enck. ded: Decompiling Android Applications. http://siis.cse.psu.edu/ded/.

[78] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *FSE '12: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 6:1–6:11, 2012. ISBN 978-1-4503-1614-9. doi: {http://dx.doi.org/10.1145/2393596.2393600}.

[79] Drew Olanoff. Google Announces It Has Reached 900M Android Activations, Sundar Pichai Calls It "Most Popular Mobile OS In The World". http://techcrunch.com/2013/05/15/google-announces-it-has-reached-900m-android-activations/, May 2013.

[80] Gabor Paller. Dedexer. http://dedexer.sourceforge.net/.

[81] Xiaobo Pan. dex2jar. http://code.google.com/p/dex2jar/.

[82] Adam Powell. How to have your (Cup)cake and eat it too. http://android-developers.blogspot.com/2010/07/how-to-have-your-cupcake-and-eat-it-too.html, July 2010.

[83] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *OSDI '12: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 107–120, 2012. ISBN 978-1-931971-96-6.

[84] Matthew Rollings. Android: Requesting root access in your app. http://www.stealthcopter.com/blog/2010/01/android-requesting-root-access-in-your-app/, January 2010.

[85] Jamie Rosenberg. Celebrating Google Play's first birthday. http://officialandroid.blogspot.com/2013/03/celebrating-google-plays-first-birthday.html, March 2013.

[86] Dan Rowinski. Google Play Will Beat Apple App Store To 1,000,000 Apps. http://readwrite.com/2013/01/08/google-play-to-hit-1-million-apps-before-apple-app-store, January 2013.

[87] Michael Saylor. *The Mobile Wave: How Mobile Intelligence Will Change Everything*. Vanguard Press, June 2012. ISBN 978-1593157203.

[88] Skype blogs. Privacy vulnerability in Skype for Android. http://blogs.skype.com/security/2011/04/privacy_vulnerability_in_skype.html, April 2011.

[89] American Dialect Society. "App" voted 2010 word of the year by the American Dialect Society (UPDATED). http://www.americandialect.org/app-voted-2010-word-of-the-year-by-the-american-dialect-society-updated, January 2011.

[90] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*, pages 1–25, 2008. ISBN 978-3-540-89861-0. doi: {http://dx.doi.org/10.1007/978-3-540-89862-7_1}.

[91] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating User Privacy in Android Ad Libraries. In *MoST '12: Proceedings of IEEE Mobile Security Technologies*, 2012.

[92] Raja Vallee-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode using the Soot Framework: Is it Feasible? In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 18–34, 2000. ISBN 978-3-540-67263-0. doi: {http://dx.doi.org/10.1007/3-540-46423-9_2}.

[93] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission Evolution in the Android Ecosystem. In *ACSAC '12: Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40, 2012. ISBN 978-1-4503-1312-4. doi: {http://dx.doi.org/10.1145/2420950.2420956}.

[94] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. ProfileDroid: Multi-layer Profiling of Android Applications. In *Mobicom '12: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, pages 137–148, 2012. ISBN 978-1-4503-1159-5. doi: {http://dx.doi.org/10.1145/2348543.2348563}.

[95] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *CODASPY '12: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, pages 317–326, 2012. ISBN 978-1-4503-1091-8. doi: {http://dx.doi.org/10.1145/2133601.2133640}.

[96] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *S & P '12: Proceedings of the 33th IEEE Symposium on Security and Privacy*, pages 95–109, 2012. ISBN 978-1-4673-1244-8. doi: {http://dx.doi.org/10.1109/SP.2012.16}.

[97] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *TRUST '11: Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, pages 93–107, 2011. ISBN 978-3-642-21598-8. doi: {http://dx.doi.org/10.1007/978-3-642-21599-5_7}.

[98] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS '12: Proceedings of the 19th Annual Network and Distributed System Security Conference*, 2012.