# An Empirical Study of Android API Use in Google and Non-Google Applications

by

Nima Ahmadi

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
March 2018

# Abstract

Android is the most popular mobile platform powering hundreds of millions of mobile devices. Android provides an API that helps developers create Android applications by simplifying the reuse of core system components and services. Different developers may use the Android API in different ways for several reasons, including: their background, their knowledge of the platform and their team's policies. This variety in how developers interact with the Android API, makes Android API use analysis an interesting research topic.

Android is owned and maintained by Google. In each release of the Android API, Google introduces new capabilities and deprecates some of the old ones. In this thesis, we investigate the fact that, since Google is itself both maintaining and using the API at the same time, it may be using Android differently than other developers.

In order to explore our research questions, since existing methods could not address some of the challenges in our study, we proposed a new method for analyzing Android API use in closed source Android applications. This method works on obfuscated applications and radically reduces the noise produced by embedded API calls in the unused parts of third-party libraries.

Using the proposed method, we conducted an experiment for comparing Android API use in 19 popular Google applications to 19 non-Google similar applications and

answered four specific research questions that show the differences between these two groups of applications.

The analysis in this thesis can be helpful for Android developers to find out how Google, as the owner and maintainer of Android, is using the Android API. Additionally, Google can use this information to observe how the changes they make affect the developers' interaction with the Android API.

# Acknowledgments

First of all, I would like to express my sincerest gratitude to my supervisor, Professor James R. Cordy for giving me the opportunity to work under his supervision and for all his support and patience during my study.

I would like to thank my lovely wife, Ghazal, who took this long journey with me away from her friends and family, for all her love and support during my research.

And finally, I would like to thank my parents, who have always encouraged me, and my brother, who was there for me any time I needed him.

# Statement of Originality

I hereby certify that the work of the author in this thesis is original, conducted under the supervision of Dr. James R. Cordy. Ideas and techniques that are not from the work of the author are fully acknowledged using citations or by paraphrasing if citations are not available, to indicate that these are the works of others.

Nima Ahmadi

February, 2018

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Android is the world's most popular mobile platform powering hundreds of millions of mobile devices. Most Android applications are written using the Android Software Development Kit (SDK) and the Java programming language. The Android SDK provides APIs for application developers.

Android API analysis is the study of how Android applications use Android APIs. API use information can be extracted from APK (Application Package) files. The extracted information can be used in areas including but not limited to: malware detection, energy usage analysis and license violation detection.

In addition to Android APIs, developers can include third-party libraries to add functionalities to their applications and speed up the development process. Third-party libraries can add large amounts of noise to static API analysis because there can be a large number of embedded API calls in the unused parts of these libraries. Android applications are usually obfuscated before compilation in order to make the process of reverse engineering harder. Obfuscation makes it challenging to detect and exclude third-party libraries from the analysis.

## 1.1 Goal of the Thesis

Android is owned by Google, and Google Play is the main marketplace for distributing Android applications. Along with other developers, Google has published a lot of applications on Google Play that are developed by its own developers. Android is evolving fast and in each release of the Android API, Google introduces some new functionalities and marks some existing functionalities as deprecated, which means that developers should avoid using them. Since Google is itself both maintaining and using the API at the same time, it may be using Android differently than other teams. The main goal of this thesis is to investigate whether Google is using Android differently than other developers by comparing API use in Google and non-Google applications.

*Thesis statement:* By analyzing how Google's Android applications are using the Android API and comparing it to how similar applications developed by other teams are using it, we can find out whether Google developers are using Android differently than other teams.

To perform the analysis, we select some of the popular Google applications and corresponding similar popular applications developed by other teams. As previous approaches for analyzing API use in Android applications have not addressed the third-party and obfuscation challenge adequately, we propose a new approach that produces reliable results for this analysis. After extracting the API information from the selected applications, we address four specific research questions:

RQ1. Do Google applications use Android differently from other applications?

RQ2. To what extent do Google applications use Android differently from other applications?

RQ3. Do Google applications adopt new Android capabilities more quickly than other applications?

RQ4. Do Google applications abandon deprecated Android capabilities more quickly than other applications?

## 1.2 Contributions

This thesis makes three contributions:

**A new method for analyzing API use in Android Applications:** This thesis proposes a new approach for analyzing API use in Android applications. This method is robust to common obfuscation techniques in Android applications and does not require third-party library detection in order to reduce the noise produced by unused parts of these libraries.

**API use data of popular Android applications:** In order to explore our research questions, we extracted API use data from 38 popular Android applications (19 Google applications and 19 non-Google applications). This data is available online for further studies [9].

**A comparison of Android API use in Google and non-Google applications:** After data extraction, we examined our data to find differences in how Google and non-Google applications use the Android API.

## 1.3 Thesis Outline

In Chapter 2, we present some background about Java, Android and Android applications. We discuss the challenges involved with Android API analysis and give a brief overview of some of the related studies in Android API analysis and third-party

library detection.

Chapter 3 serves as an overview of the research, providing a high-level description of our process. It includes our motivation, a brief overview of our proposed method, and an overview of our experiment with Google and non-Google applications.

Chapter 4 focuses on our method for analyzing API use in Android applications and how it addresses the challenges that previous methods could not address adequately.

Chapter 5 describes our process of application selection and API data extraction in our experiment. Additionally, this chapter presents some of the results and discusses our observations and the answers our research questions.

Finally, in Chapter 6, we conclude this thesis with a summary of our observations and suggestions for future work.

# Chapter 2

# Background and Related Work

In this chapter, we provide the relevant background information for this thesis and survey some of the related work. First, we describe the Java programming language and its runtime environment. Second, we describe some basic information about Android and its applications. Third, we discuss challenges in Android applications analysis. Fourth, we review some of the relevant studies. Finally, we discuss the limitations of previous work.

## 2.1 Java

Java is a general purpose, high level and object-oriented programming language developed by and released by Suns Microsystems in 1995. The primary motivation of creating Java was the need for a platform-independent language that could be used to create software to run on different platforms which is called "write once, run anywhere". Java applications are portable because the Java compiler compiles source code to bytecode to be interpreted by the Java Virtual Machine (JVM) instead of machine-dependent binary machine code.

To clarify the "write once, run anywhere" concept, Figure 2.1 shows how gcc (the

GNU Compiler Collection) compiler compiles a C function. C compilers compile the code directly in binary code which is executable on the machine that the compiler is designed for. On the other hand, in Figure 2.2, Java compiler compiles the code to JVM bytecode. Bytecode is executable by the JVM, which makes them portable to any machine that has a running JVM.

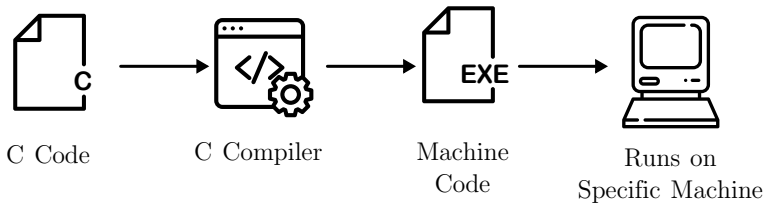


Figure 2.1: C code compilation

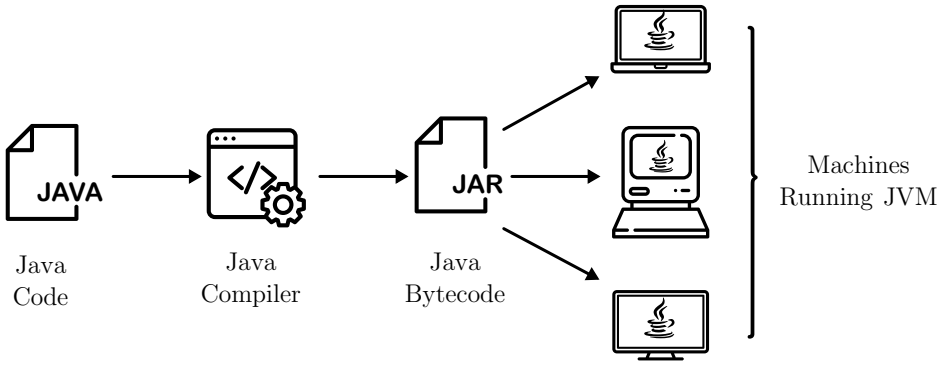


Figure 2.2: Java code compilation

## 2.2 Android

Android is a mobile operating system developed by Google. It is the most popular mobile operating system, running on more than 87% of mobile phones [11]. Android is an open-source project and its source code is released under an open source license.

Android applications are developed using the Java programming language. Java classes are compiled into proprietary bytecode format and run on Android Runtime (ART), a specialized virtual machine (VM) designed for Android. Android applications are packaged in APK files, which are actually zip archives. The contents of APK files can be extracted using zip archive tools. In Figure 2.3 the contents of a sample Android application is shown. We can divide files in an APK file in three major groups: Compiled Classes, AndroidManifest.xml and Resources. Later in this chapter, we will explain how these important parts fit together and run an Android application.

```
DemoApp/
├── AndroidManifest.xml
├── META-INF
│   └── [certificate files]
├── classes.dex
└── res
    ├── anim
    ├── color
    ├── drawable
    └── ...[32 other directories]
```

Figure 2.3: Contents of an APK file

### 2.2.1 The Android API

Most Android applications are written using the Android Software Development Kit (SDK) and the Java programming language. The Android SDK provides an API

for application developers. The entire feature-set of the Android OS is available to developers through this API. The Android API helps developers create Android apps by simplifying the reuse of core system components and services. Developers interact with the Android API in variety of ways, including: calling class or instance methods, instantiating objects and extending a class. In this thesis, we use the keyword "end point" to describe elements of the Android API that developers can interact with in any form.

### 2.2.2 Compiled Classes

In addition to the Android API, developers usually use third-party libraries in order to add functionalities to their applications. Libraries such as mobile analytic tools and UI components are among the most popular libraries in Android applications [2].

When building an APK file from an application's source code, the classes written by the application developers and the used libraries are compiled into one or more DEX files. Usually, each application contains one DEX file but because of the "64K reference limit" [8], some applications have more than one. These files are executable on the Dalvik virtual machine which was used until Android version 4.4 "KitKat". Dalvik was replaced by Android Runtime (ART) later but DEX files are still supported. Dalvik bytecode is different than Java bytecode. Thus, DEX files are not executable on Java Virtual Machines. DEX files and JAR files are translatable to each other, meaning there are tools that can translate from Dalvik bytecode to Java bytecode and vice-versa.

### 2.2.3 Resources

Resources are non-code assets associated with the application. For example: images, audios, XML files, etc. The Android SDK tools compile the application's resources into binary at build time. Developers can choose to keep the raw version of the resource by placing them in *assets* folder. For example, a font can not be used by the application at run time if it gets compiled to binary at build time. Therefore, resources like fonts, texts, videos and audios should be placed in *assets* directory.

### 2.2.4 AndroidManifest.xml

Every Android application must have a file named AndroidManifest.xml in their root directory which provides essential information about the application for the Android operating system. The Android operating system must have the information in AndroidManifest.xml file before it can run any of the application's code. Listing 2.1 shows a basic AndroidManifest.xml file which is generated by Android Studio for a new Android application project. It provides information like the application package name, application icon, application label and it defines the the application's components (in this case a single `Activity`).

**Listing 2.1** Decoded AndroidManifest.xml file from a sample application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.demo">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="Demo Application">

        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The application package name is used by Android OS to find other classes mentioned in the manifest file. For example, in Listing 2.1, the defined activity's name is `".MainActivity"`. In order to find the class associated with this activity, Android OS will look for a class named `MainActivity` inside `com.example.demo` package.

The application icon tag points to an image resource in the resource directory of the application. This image will be shown as the application icon to users when they want to launch the application.

Inside the `<activity>` tag, developers define the components of their application. Each component presents a class in the application from which the application's code execution can begin. Each application might have several components defined in AndroidManifest.xml file.

According to the Android documentation [6], developers can define four kinds of

components: Activities, Services, Receivers and Providers. In order to explain these elements, we first take a look at how the Android documentation explains them and then we give examples of their use in an imaginary chat application.

- Activity

  "`<activity>` declares an activity (an `Activity` subclass) that implements part of the application's visual user interface. All activities must be represented by `<activity>` elements in the manifest file. Activities that are not declared there will not be seen by the system and will never be run."

  For example, in the imaginary chat application, we have at least two activities. One activity showing a list of recent conversations and another one, that will launched when the user selects a chat and show the messages of that conversations.

- Service

  "`<service>` declares a service (a `Service` subclass) as one of the application's components. Unlike activities, services lack a visual user interface. They're used to implement long-running background operations or a rich communications API that can be called by other applications. All services must be represented by `<service>` elements in the manifest file. Services that are not declared there will not be seen by the system and will never be run."

  In the imaginary chat application, there is a service that is always running regardless of whether the application is running in the foreground or not. It will wait for new messages to arrive and push a notification to users in order to attract their attention.

- Receiver

  "`<receiver>` declares a broadcast receiver (a `BroadcastReceiver` subclass) as one of the application's components. Broadcast receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running."

  One example of broadcast receivers in the imaginary chat application can be a receiver that receives system broadcasts about network status changes to let the service listening for new messages know about current network status.

- Provider

  "`<provider>` declares a content provider component. A content provider is a subclass of `ContentProvider` that supplies structured access to data managed by the application. All content providers in an application must be defined in a `<provider>` element in the manifest file; otherwise, the system is unaware of them and doesn't run them."

  For example, in the imaginary chat application, we can share list of the user's contacts with other applications using a content provider.

## 2.3 Challenges in Android application API analysis

There are a few challenges involved in Android application API analysis. In this section we discuss these challenges.

### 2.3.1 Analyzing closed-source Android applications

There are a lot of open-source Android applications published in various online markets. For example, F-Droid [10] is a software repository for Android applications that contains only open-source applications. As of the day of writing this thesis, it contains more than 2,600 open-source applications. But considering 3.5 billion applications published on Google Play Store, it can be concluded that most of the popular Android applications are closed-source and we do not have access to their source code, thus we can not analyze them using source code parsing methods.

### 2.3.2 Obfuscation and Third-Party Libraries

It is a common practice in Android application development to obfuscate the code before compilation in order to provide security against reverse engineering. One of the most commonly used tools for obfuscation is ProGuard [14] which is preinstalled with Android Studio. ProGuard can rename classes and methods in order to make the decompiled code harder to understand. Additionally, it can destroy the packaging structure by merging all the classes from different packages into the default package and make it impossible to determine purpose of a class by its package.

In this thesis, by "third-party libraries" we mean the libraries that are not developed by the developer of the application. As third-party libraries are included in application DEX files and get obfuscated along with the code that developer writes, it is very challenging to distinguish the libraries' code in obfuscated Android applications. For this reason, in our work we compare the entire application's use of the Android API, rather than attempting to identify and remove third party libraries.

Third-party libraries can add huge amounts of code (compared to the code that

the developers write) to an application. We could not find a study that shows what percentage of third-party library code is actually used by the applications, but as third-party libraries are usually generalized to meet a broader set of requirements, it can be inferred that usually a small portion of libraries are used by the applications. Thus, these libraries can cause a lot of noise in any analysis.

To clarify this challenge, we developed a sample application that contains a single activity. When user launches the application, it will show some text to user. We included the *Google Support library* even though we did not use any of its features.

Figure 2.4 shows the internal structure of the *classes.dex* file in our application which was built without any obfuscation. In this figure, there are two main packages: *com.example.demo* that contains the classes written by the developer and *android.support* that contains classes related to *Android Support library*. Because of the large number of files we could not show all the files here, but we have included some of the class names in *android.support.v7.app* package in order to show that the purpose of these classes is usually predictable from their names. Although this library was not actually used in our sample application, the whole library, containing more than 1600 classes, was included in the APK file. As mentioned earlier, these unused library classes can cause a lot of noise in API analysis results.

Figure 2.5 shows the classes from the same application after it is obfuscated using ProGuard. In the obfuscation process, in the majority of cases, class names are changed so they can no longer be helpful for a human to understand its purpose and classes are moved out of their packages into the root package. More than 300 classes were moved to root package in our sample application. This example shows that in obfuscated applications, library classes and the developers' classes are really hard to

distinguish from each other.

```
classes/
├── android [57 directories, 1606 files]
│   └── support
│       ├── annotation
│       ├── compat
│       ├── fragment
│       ├── graphics
│       │   └── drawable
│       │       └── animated
│       ├── v4
│       │   ├── accessibilityservice
│       │   ├── database
│       │   ├── graphics
│       │   │   └── drawable
│       │   ├── hardware
│       │   │   ├── display
│       │   │   └── fingerprint
│       │   └── [...]
│       └── v7
│           ├── app
│           │   ├── ActionBarDrawerToggle$Delegate.class
│           │   ├── ActionBarDrawerToggle$DelegateProvider.class
│           │   ├── ActionBarDrawerToggle$SetIndicatorInfo.class
│           │   ├── ActionBarDrawerToggle$SlideDrawable.class
│           │   ├── ActionBarDrawerToggle.class
│           │   ├── ActivityCompat$1.class
│           │   ├── ActivityCompat$OnRequestPermissionsResultCallback.class
│           │   ├── ActivityCompat$RequestPermissionsRequestCodeValidator.class
│           │   ├── ActivityCompat$SharedElementCallback21Impl.class
│           │   ├── ActivityCompat$SharedElementCallback23Impl$1.class
│           │   ├── ActivityCompat.class
│           │   └── ... [86 other files]
│           ├── appcompat
│           ├── content
│           │   └── res
│           ├── graphics
│           │   └── drawable
│           ├── text
│           ├── view
│           │   └── menu
│           └── widget
└── com [2 directories, 3 files]
    └── example
        └── demo
            ├── BuildConfig.class
            ├── MainActivity.class
            └── R.class
```

Figure 2.4: Contents of an unobfuscated classes.dex file

```
classes/
├── a.class
├── aa$a.class
├── aa$b.class
├── aa$c.class
├── aa$d.class
├── aa$e.class
├── aa$f.class
├── aa$g.class
├── aa.class
├── ... [329 other classes]
├── android
│   └── support
│       ├── annotation
│       ├── constraint
│       ├── v4
│       │   ├── media
│       │   └── widget
│       └── v7
│           ├── app
│           │   ├── a$a.class
│           │   ├── a$b.class
│           │   ├── a$c.class
│           │   ├── a.class
│           │   ├── b$a.class
│           │   ├── b.class
│           │   ├── c.class
│           │   ├── d.class
│           │   ├── e.class
│           │   ├── f$1.class
│           │   ├── f$a.class
│           │   ├── f.class
│           │   ├── g$a.class
│           │   └── ...
│           ├── view
│           │   └── menu
│           └── widget
└── com
    └── example
        └── demo
            └── MainActivity.class
```

Figure 2.5: Contents of an obfuscated classes.dex file

### 2.3.3 Detecting Android API Calls

In Android, application developers can call APIs from their Java classes just like any other method calls. In order to analyze Android API calls we need to be able to distinguish API calls from other method calls.

It is easy to detect API calls to static or final methods. In these calls, the compiler decides which method should be dispatched and this information is present in the bytecode. On the other hand, non-final, instance methods are dynamically dispatched, meaning that the compiler does not specify the dispatched method at compile time. Later at run time, the JVM chooses a method to dispatch depending on the type of object. Dynamic dispatching makes it hard to detect API calls statically, without running the application, because we do not have a reference to the actual despatched method in the bytecode.

For example, Listing 2.2 shows a Java class named *MainActivity* which extends *android.app.Activity*. This class overrides the *onCreate* method from its superclass. In this method, there is a call to the *setContentView* method which is an instance method defined in the super class. In this case, the Java compiler compiles this line to a dynamic invocation of the *com.example.demo.MainActivity.setContentView* method. The JVM will later decide which method to invoke for this method call. In this case, as *MainActivity* does not have its own implementation of *setContentView*, the JVM will search for the method in its super class (*android.app.Activity*).

**Listing 2.2** Dynamic dispatch example in Java

```
package com.example.demo;

import android.app.Activity;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        setContentView(R.layout.activity_main);
    }
}
```

## 2.4 Related Work

In this section we discuss some studies related to API analysis in Android applications.

### 2.4.1 Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations

Azad [15], in their PhD thesis, mined the API method calls used by 230 open source Android applications to cluster co-changing API methods in order to predict the changes to API methods that individual application developers will make to their application. They extracted the data by parsing the source code of the applications. Additionally, they used API data extracted from highly-voted Android related answers in StackOverflow for the same purpose.

In the second part of the thesis, they propose an approach to find similarities between proprietary applications and open source applications based on their Android

API use. They used 150 open source applications from F-Droid and 950 proprietary applications from Google Play Store. Their goal is to use this information to determine whether closed source applications are copying code from open source applications. Since the source code of proprietary applications is not available, they extract API use data from compiled binaries (combination of developer and library code) for both groups and considered any function call belonging to the **Android.*** package an Android API call.

### 2.4.2 Understanding Reuse in the Android Market

Ruiz et al. [19] studied class reuse and inheritance in five categories of Android applications in Google Play using the concept of Software Bertillonage. They generated class signatures from classes in DEX files (combination of developer and library code) of applications and try to find similar classes across different applications. A class signature includes the fully qualified name (name of the class along with the package that it belongs to) of a Java class and a sorted set of method signatures in that class. Additionally, they studied API class inheritance in applications and calculate the percentage of classes that inherit from classes in **Android.*** package.

In this study, they did not mention how they dealt with obfuscation which can effect the class signatures. But later, this study was expanded by Ruiz et al. [18] to analyze more applications and it is mentioned there that the obfuscated classes were omitted and the analysis was ran on the classes that were not obfuscated.

### 2.4.3 LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps

Zing Ma et al. [17] presented a tool, LibRadar, in their study that can detect third-party libraries in Android applications. This tool can help other researchers separate developers' code from third party libraries in obfuscated applications. LibRadar detects libraries based on their use of the Android API. This feature does not change in most obfuscation techniques because API calls can not be renamed. They analyzed one million free Android applications from Google Play Store and used clustering-based techniques to identify possible libraries. Over 10,000 library versions were identified through clustering. After the learning phase, LibRadar can identify third-party libraries in a given applications.

## 2.5 Limitations of Previous Work

In this section we explain the limitations in previous studies that we had to address in our study.

### 2.5.1 Obfuscation and Third-Party Libraries

As mentioned before, third-party libraries can cause a lot of noise in API use analysis as they can have relatively huge number of API calls compared to the code that the developers write. In order to remove this noise, studies have taken different paths. We will discuss these solutions and why we could not use them in following sections.

### Detecting and Removing Third-Party Libraries

A few studies try to detect libraries used in obfuscated applications, like [17] and [16]. Basically, they generate a database of "signatures" from libraries and try to find matches in a given application. Signatures are minimal version of classes in libraries with normalized names (class names, variable names, etc). Name normalization makes the signatures suitable for detecting libraries in obfuscated applications.

We tried LibRadar [13] on a few applications. According to its website, is trained with more than 1 million applications from Google Play, so it can identify virtually all popular libraries in any given Android application. The results did not include all libraries in the applications specially the less popular libraries.

In our study, even missing one library can result in a lot of noise in the outcome. Additionally, this approach will omit the Android API calls from the used parts of the libraries as well. Therefore, we could not use this approach.

### Removing All Obfuscated Classes

Android obfuscation tools usually offer features for including/excluding classes from the obfuscation process and users can decide which packages or classes they want to obfuscate. Additionally, some classes are excluded automatically. For example if an application has an Activity class that should start when the application is launched, it can not be renamed in obfuscation phase because there is a pointer to that class in the *AndroidManifest.xml* file and Android OS should be able to find that class using its name. Therefore, each application has some classes that are not obfuscated. Some studies like [18], deal with obfuscation by removing all obfuscated classes and analyzing the remaining classes.

It is tempting to use this approach because we would end up with some classes which are not obfuscated and we could decide if each one is from a library or not, by simply looking at its package name. But, on the other hand, we would be losing a lot of data by removing all obfuscated classes and it should be considered that the remaining classes were excluded from obfuscation for a reason. If we take this approach, we will end-up analyzing specific kinds of classes like activities, services, receivers and providers which had a pointer in AndroidManifest.xml file and these are just a small portion of any Android application.

### 2.5.2 Detecting Android API Calls

As API calls are just Java function calls, API use analysis requires a method to separate API calls from other function calls. Android application studies usually use package name for detecting API calls. The package name is extracted from Java bytecode using different tools. This approach suggests that when a function is called on a class in a package starting with "`android.`", that function is an Android API call. As mentioned in subsection 2.3.3, this approach can only detect static and final API methods and it can not detect instance methods which are not bound at compile time.

## 2.6 Conclusion

In this chapter we provided some background information about Java and Android and the structure of Android applications. We provided some challenges involved with Android application API analysis and reviewed some of the related research. Finally, we reviewed some of the limitations of the previous research.

# Chapter 3

# Overview

This chapter discusses our motivation and goal, it explains our thesis statement and gives an overview of the contributions of this thesis.

## 3.1 Motivation

As a result of the popularity of the Android OS, there are a lot of Android application developers who have different backgrounds and have learned their skills from different resources. The Android API changes very frequently (4 releases in the last 2 years [3]) and because of the amount of change in each release (for example, 12.19% of the API changed from version 22 to 23 [1] ), some teams may not be able to keep up with these changes. Additionally, each company (or each team) has a different set of standards and best practices. Changes in the Android API and diversity in learning resources, backgrounds and team policies can cause different groups to use the Android API in different ways.

As Google owns Android and is in charge of maintaining it, we can assume that Google developers may know how to use the Android API better than other companies and its developers may use the Android API in a different way. The main goal of

this thesis is to investigate whether Google developers actually use the Android API differently than developers in other teams.

## 3.2 Goal of the Thesis

The goal of this thesis is to explore differences in how the Android API is being used by developers inside and outside of Google. The results can be helpful for the community of Android developers by giving them better insight into how they can more effectively use the Android API. Additionally, it can help Google by showing how the Android API is actually used by developers who are outside of Google.

*Thesis statement:* By analyzing how Google's Android applications use the Android API and comparing it to how similar applications developed by other teams use it, we can see if Google developers are using Android differently than other teams.

## 3.3 Contributions

To perform this analysis, we selected two sets of applications, one containing applications developed by Google and the other set containing similar applications developed by other companies. These applications are mostly closed-source, obfuscated, and usually contain third-party libraries. To analyze and compare how they use the Android API, we need an analysis method that works on closed-source, obfuscated applications and can reduce the noise produced by the unused portion of third-party libraries.

Moreover, some Android API endpoints are called via instance methods which are dynamically dispatched at the run time. In order to have a more precise view of how an application uses the Android API, we need a method that can statically predict

the dispatched methods of these calls without running the application.

Previous methods have not addressed these two important challenges in analyzing how Android applications use the Android API. Consequently, in addition to the main purpose of this thesis, we have developed a new method to address these challenges. Therefore, in this thesis we have three main contributions:

- A new method for analyzing API use in Android applications in the presence of third-party libraries and obfuscation called the "Transitive Closure Method".
- A comparison in Android API use between applications developed by Google and similar applications developed by other teams.
- Detailed API use data extracted from 38 popular Android applications [9].

A brief overview of each contribution is given in the following sections.

## 3.4 The Transitive Closure Method

Android applications usually include a few third-party libraries in order to extend their functionalities. These libraries are normally generalized so they can meet broader requirements and attract more users. Thus, they usually provide more functionalities than a single application needs. Consequently, applications actually use only a small portion of the third-party libraries they include. There can exist a large number of API calls in the unused portion of these libraries that can introduce a lot of noise into API use analysis results.

In this thesis we propose a new method for analyzing API use in Android applications that can be used on closed-source applications. Our method omits the noise

introduced by unused parts of third-party libraries and is robust to popular obfuscation techniques in Android applications. The method starts from the entry-points of the applications and uses a transitive closure algorithm to discover reachable classes. It reduces noise by analyzing reachable classes and excluding unreachable classes from the analysis.

The development of the Transitive Closure Method, along with how it overcomes the challenges involved with Android API analysis, is described in detail in Chapter 4.

## 3.5 Google vs. Others Experiment

After proposing our new method for analyzing API use in Android applications, we conducted an experiment to explore our thesis statement and hopefully find meaningful differences in Android API use among applications developed inside and outside of Google. Using a set of filters including popularity of the applications and their convertibility to JAR format, we selected 19 Google applications and 19 corresponding similar applications developed by other teams and analyzed them in hope of finding differences in Android API use among these two groups.

Chapter 5 presents full description of the experiment and the results and findings.

## 3.6 Summary

This chapter gave an overview of this thesis, including its goal, thesis statement and contributions. In the following chapters each of the contributions is outlined in detail.

# Chapter 4

# The Transitive Closure Method

This chapter describes our proposed method for analyzing API use in Android applications. This approach does not require access to applications' code, thus, it can be used for closed-source applications. Additionally, this approach radically reduces the noise caused by unused library classes which, to our knowledge, has never been done prior to this study. This method is robust to common obfuscation techniques in Android applications, which makes it usable on the majority of available applications in Android markets.

## 4.1 Sample Application

In order to explain our method more clearly, we demonstrate how it works using a sample application. This application is a simple Android application that waits for the user to input a text and logs the text in a file. In order to support the Google Drive backup feature, this application includes *com.google.android.gms.drive* as a third-party library. The compiled application includes two classes (five methods) that are written by the developer and 44 classes (more than 1200 methods) from the third-party library.

Figure 4.1 shows the internal structure of our sample application before obfuscation. There are two packages inside this application. One contains the classes developed by the developer and the other contains classes of the third-party library. This application has one entry point, indicated by a green arrow. Method calls inside the application is indicated by blue arrows initiated from and ended inside the application.

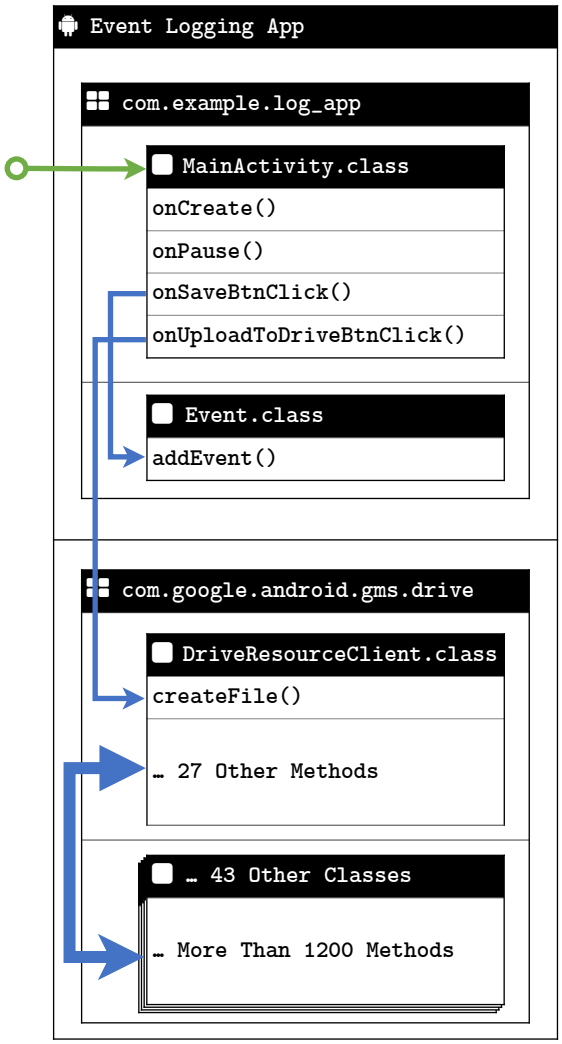


Figure 4.1: Internal structure of our sample application before obfuscation

In order to show that our method is robust to common obfuscation techniques and to make this sample application more similar to popular applications in Android markets, we explain this method using the obfuscated version of the application. Figure 4.2 shows the internal structure of our sample application after obfuscation. There is one package inside this version of application which contains the entry point

of the application. As mentioned in Chapter 2, entry points can not be obfuscated because they should be locatable using their names in AndroidManifest.xml file. Other classes inside the application are moved out of their packages and renamed to short meaningless names such as *a.class*. Methods inside these classes are renamed as well.

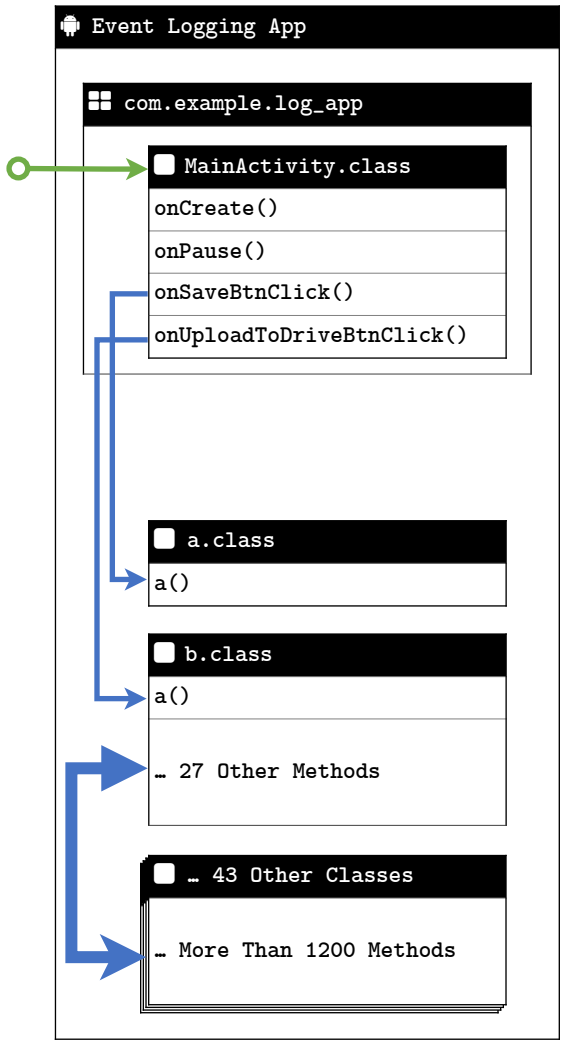


Figure 4.2: Internal structure of our sample application after obfuscation

## 4.2 Transitive Closure Method

This method consists of three main steps that we explain each in detail in the following subsections.

### 4.2.1 Detecting Entry Points of the Application

Our method starts with analyzing the entry points of the given application in order to detect reachable methods and Android API calls . To do so, we need to detect the entry points specified in the *AndroidManifest.xml* file by parsing it.

As discussed in Section 2.2.4, Android applications usually have several entry points from which execution of the applications can begin. Developers define these entry points in *AndroidManifest.xml* file which is located in the root directory of an application's APK file. This file is compiled to a binary version when the application is built. In order to convert it back to a parsable version, the Apktool [5] is used.

After producing the parsable version of the *AndroidManifest.xml* file from the binary version, we parsed it using an XML parser and searched for entry point tags described in Section 2.2.4: `<activity>`, `<service>`, `<receiver>` and `<provider>`. In each entry point tag, there is an `android.name` attribute pointing to a class. We pass a list of entry point classes from this step to the next step.

The decoded version of the *AndroidManifest.xml* file from our sample application is shown in Listing 4.1. Parsing this file and searching for entry point tags will detect one entry point for this application, the *MainActivity* class.

**Listing 4.1** Decoded AndroidManifest.xml file from our sample application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.log_app">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name">
        <activity android:name="com.example.log_app.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

### 4.2.2 Detecting Android API Calls

In our method, we look at the given application as a directed graph in which the methods inside the application code (including third-party libraries) are nodes and the method calls between these methods are the edges. Thus when there is a call inside method $A$ to method $B$, there is an edge from $A$ to $B$. Some of the edges are Android API calls. These calls originate from a method inside the application to a method outside of the application because Android API endpoints are not implemented inside applications. Android API calls can be separated from other calls outside the application (like Java API calls) because they are all implemented in the *"android"* package.

Edges in this graph are not defined from the beginning of our approach. For each method that we analyze, we determine its edges as we go through its bytecode. There

are three different situations in which we decide to add edges between methods:

- **Static Method Calls**

  Java uses static binding for private, final and static methods meaning that, at compile time, the Java compiler decides which exact method should be dispatched for each of these calls. In these cases, it is easy to detect the dispatched method because the Java compiler specifies it in the bytecode.

  In our approach, when we visit a static method call, we try to locate the class in which the callee exists. If we find the class in the JAR file, we consider it to be an edge from the current method to that method. When we are unable to find the class, by checking the package name of the class we can decide if it is an Android API call or not. If so, we add the Android API endpoint to the list of used API endpoints.

- **Dynamic Method Calls**

  Java uses dynamic binding for methods which are not private, final or static. At compile time, the Java compiler can not decide which exact method should be called because different methods might get dispatched on different execution paths. Thus, the Java compiler leaves call information in the bytecode, and the JVM will try to find the method at run time using that information and the stored object of the variable on which the method is called.

  In our approach, we want to statically analyze applications without running them. Therefore, we can not always accurately predict the method that will get dispatched at runtime.

Detecting the dispatch method without running is a complex issue due to inheritance and method overriding in Java. We try to predict the dispatched methods using the information in the bytecode provided by the Java compiler. The Java compiler provides the type of variable on which the method is called and the name of the method. One efficient way is to recursively search the variable class and its superclasses for an implementation of the called method.

This solution can be imprecise in some cases. For example if there is an upcasting before calling a method which is overridden by the subclass, the method specification in the compiled file will point to the method in the superclass, but at runtime, the method in the subclass will be dispatched. Listing 4.2 shows a class and its subclass which overrides one of its methods. At runtime, the *greet* method of the *FrenchPerson* class will be dispatched, but statically analyzing the *main* method of the *Main* class will show a method call to the *Person* class. In such cases, this approach will not predict the correct dispatched method at runtime.

**Listing 4.2** Upcasting Example

```
public class Person {
    public void greet(){
        System.out.println("Hello!");
    }
}

public class FrenchPerson extends Person {
    @Override
    public void greet() {
        System.out.println("Bonjour!");
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new FrenchPerson();
        p.greet(); //Prints "Bonjour!"
    }
}
```

Our method always returns the best prediction even when it can not find the actual implementation for a function. For example, when we want to predict the dispatched method for an Android API call, the actual implementation does not exist in the application's JAR file. In such cases, this method will search as far as possible in the superclasses and return the first superclass that could not be found in the JAR file.

If an implementation of the method was found in the JAR file, we consider it to be an edge from the current method to that method. Otherwise, when we are unable to find the class in JAR file, if the fully qualified class name of the class in which the predicted method exists stars with *"android."*, we can infer

that it is an API call to the Android API. In such cases, we add this Android API endpoint to the list of used API endpoints.

#### Class Instantiation

In Java, instantiating a class automatically calls the method `<init>` in the instantiated class. In these cases we consider the whole class as used and consider there to be an edge from the current method to each method implemented in the instantiated class.

We traverse this graph in order to find all of the reachable Android API endpoints from the entry points of the given application. We use a breadth-first approach that starts with visiting the methods in the entry points of the application. In our sample application, there is a single entry point which has four methods implemented inside it. We add all of them to a queue and analyze them one by one. Figure 4.3 shows the initial state of our traversal algorithm.

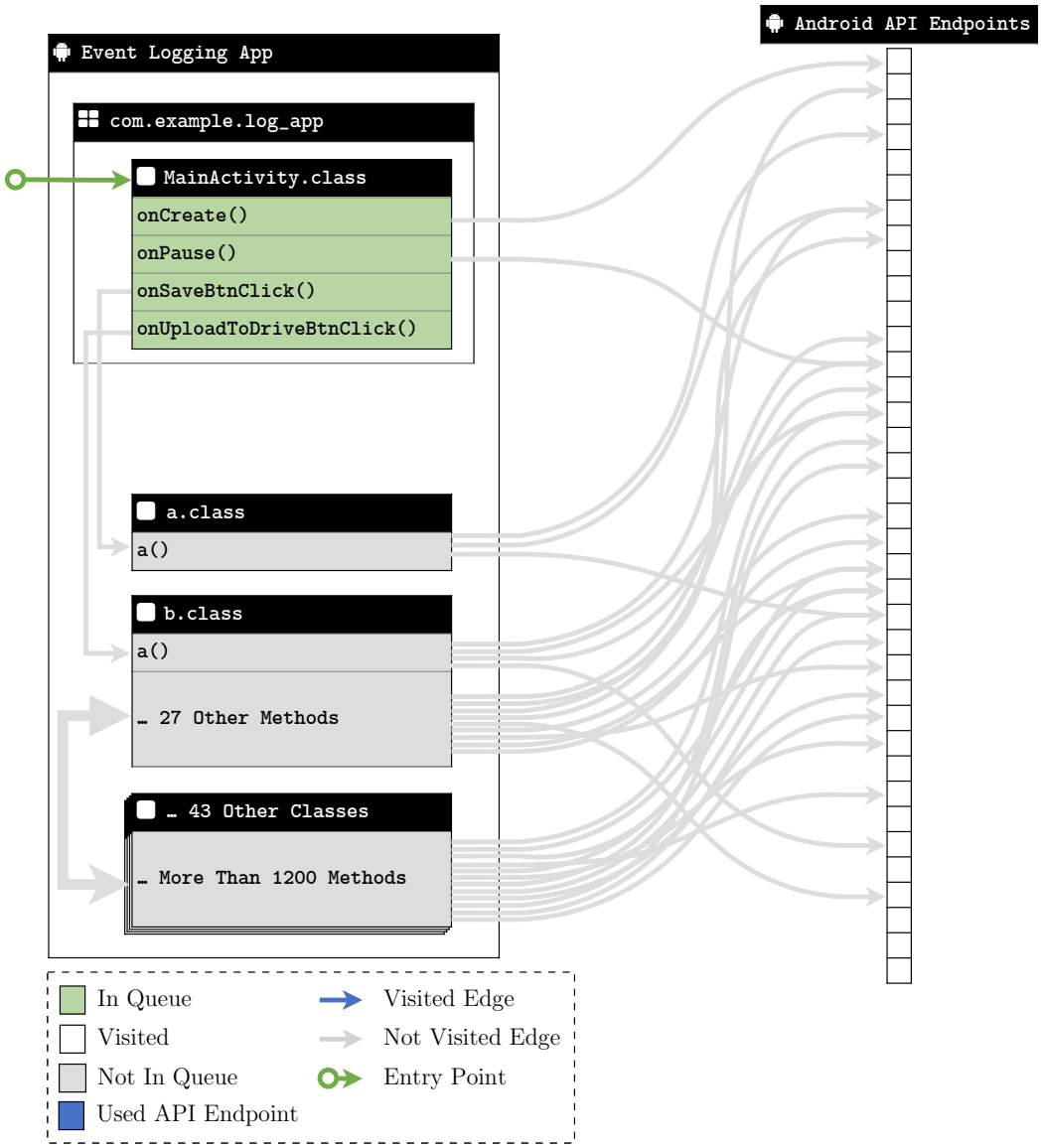


Figure 4.3: Initial state of our traversal algorithm for the sample application

In each iteration, we pull one method from the queue and analyze it in order to discover new reachable methods inside the application and new Android API calls. If we find new reachable methods, we add them to the queue, and if we find a new API call in the method, we add it to another list in order to generate a report afterwards.

As this graph can have cycles inside it (e.g. method $A$ calls method $B$ and method $B$ calls method $A$), we visit each method at most once. Figures 4.4, 4.5, 4.6, 4.7, 4.8 and 4.4 demonstrate each iteration that happens in analyzing the sample application.

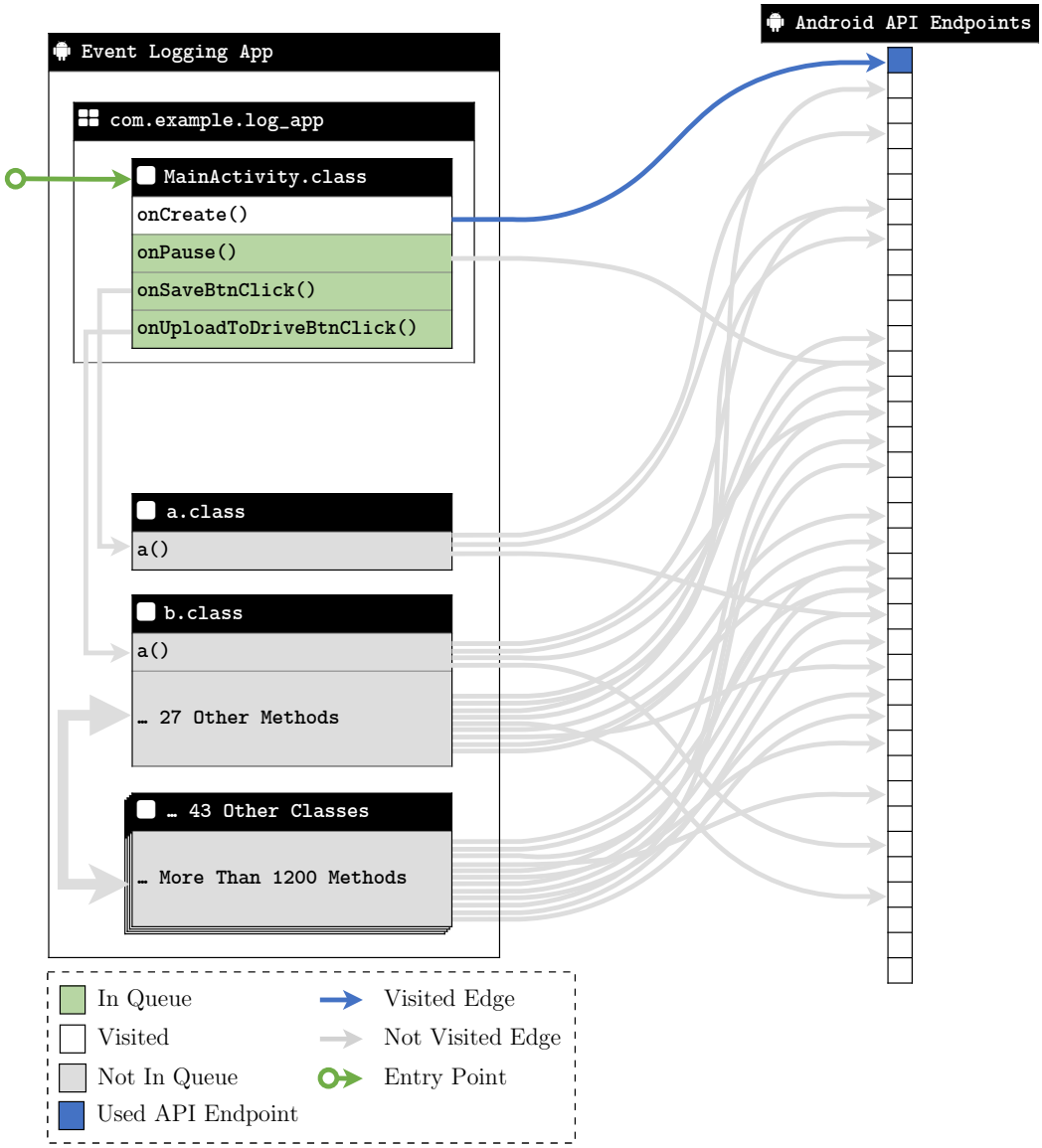


Figure 4.4: Sample application traversal: Iteration 1

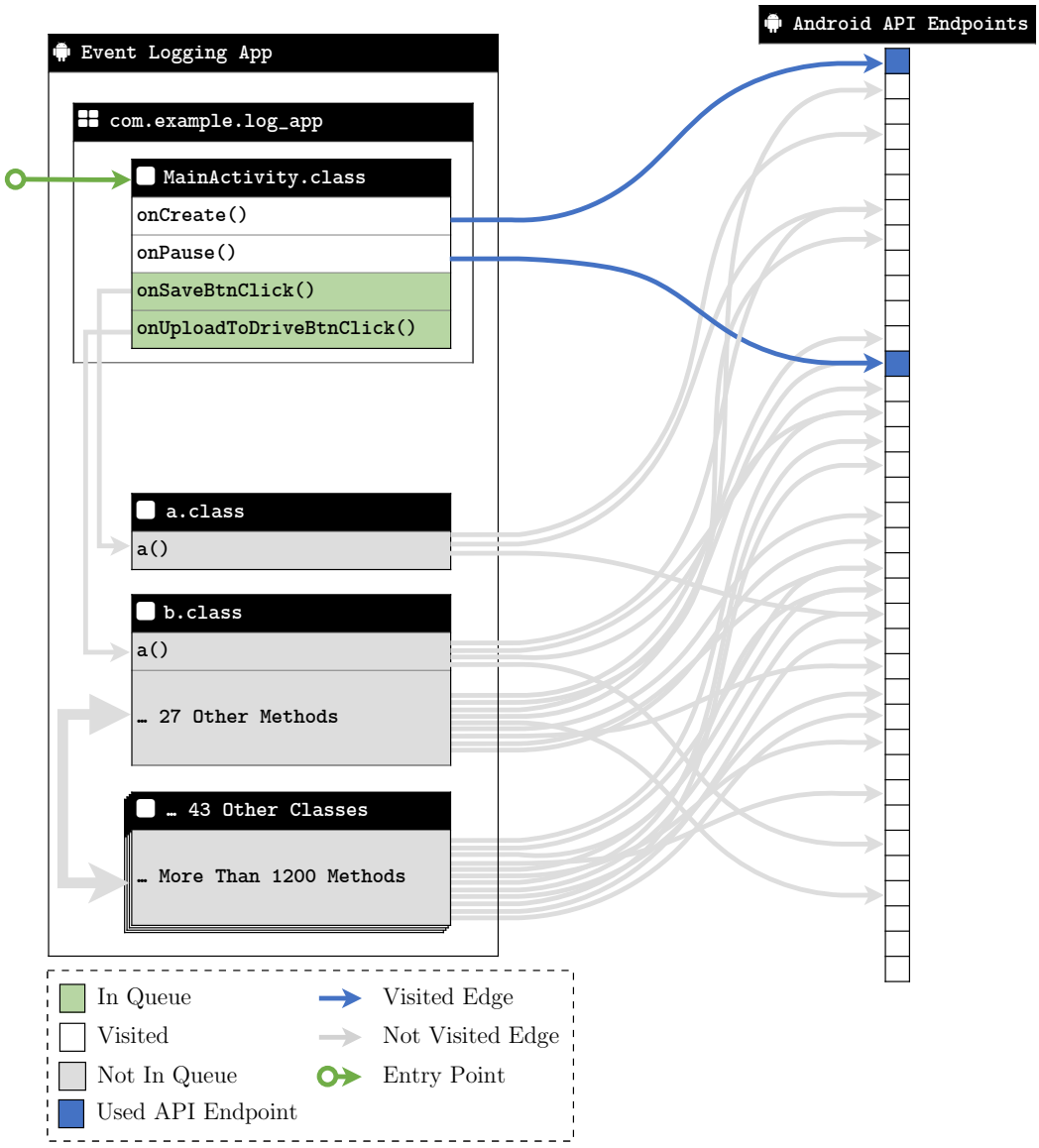


Figure 4.5: Sample application traversal: Iteration 2

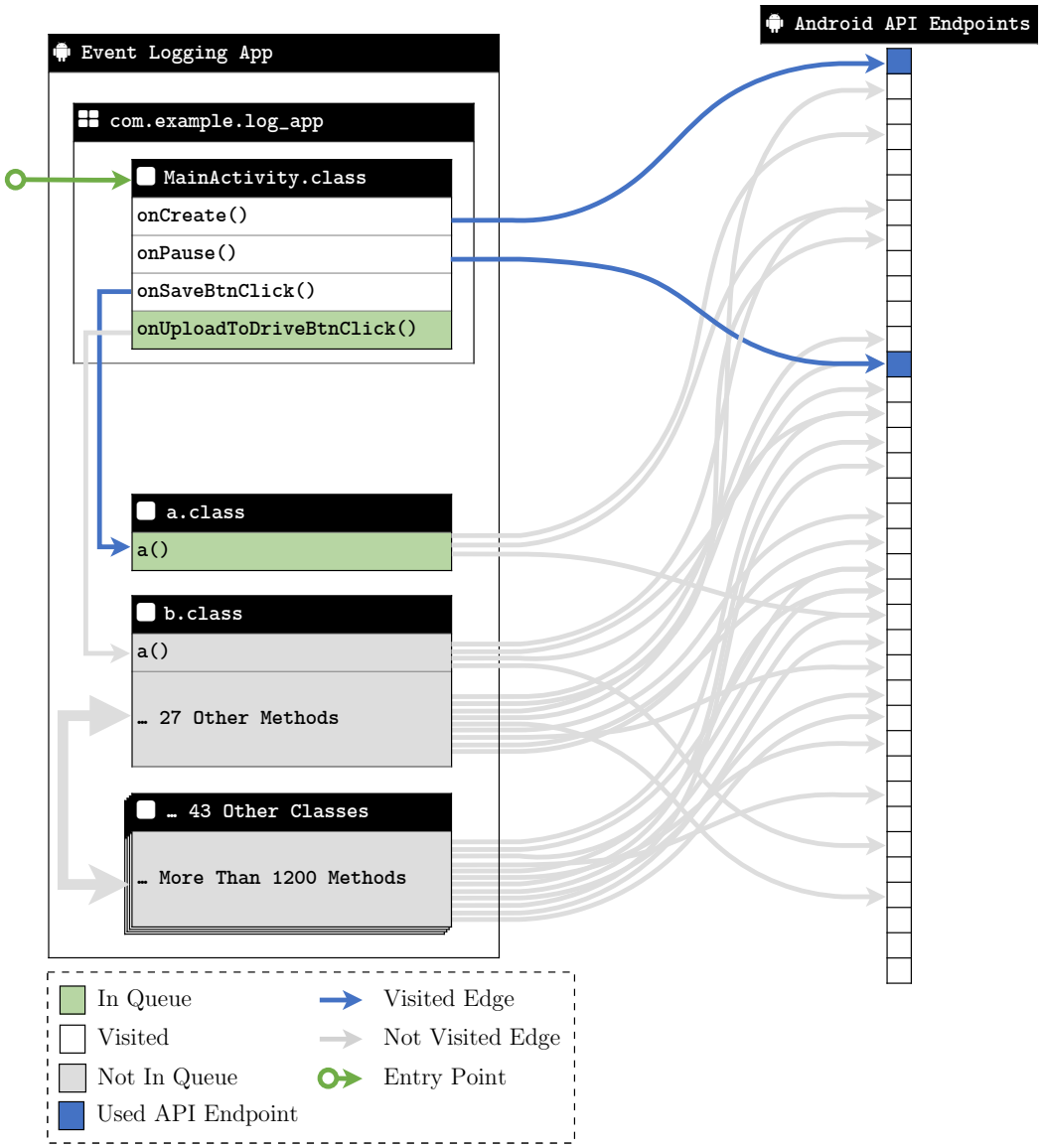


Figure 4.6: Sample application traversal: Iteration 3

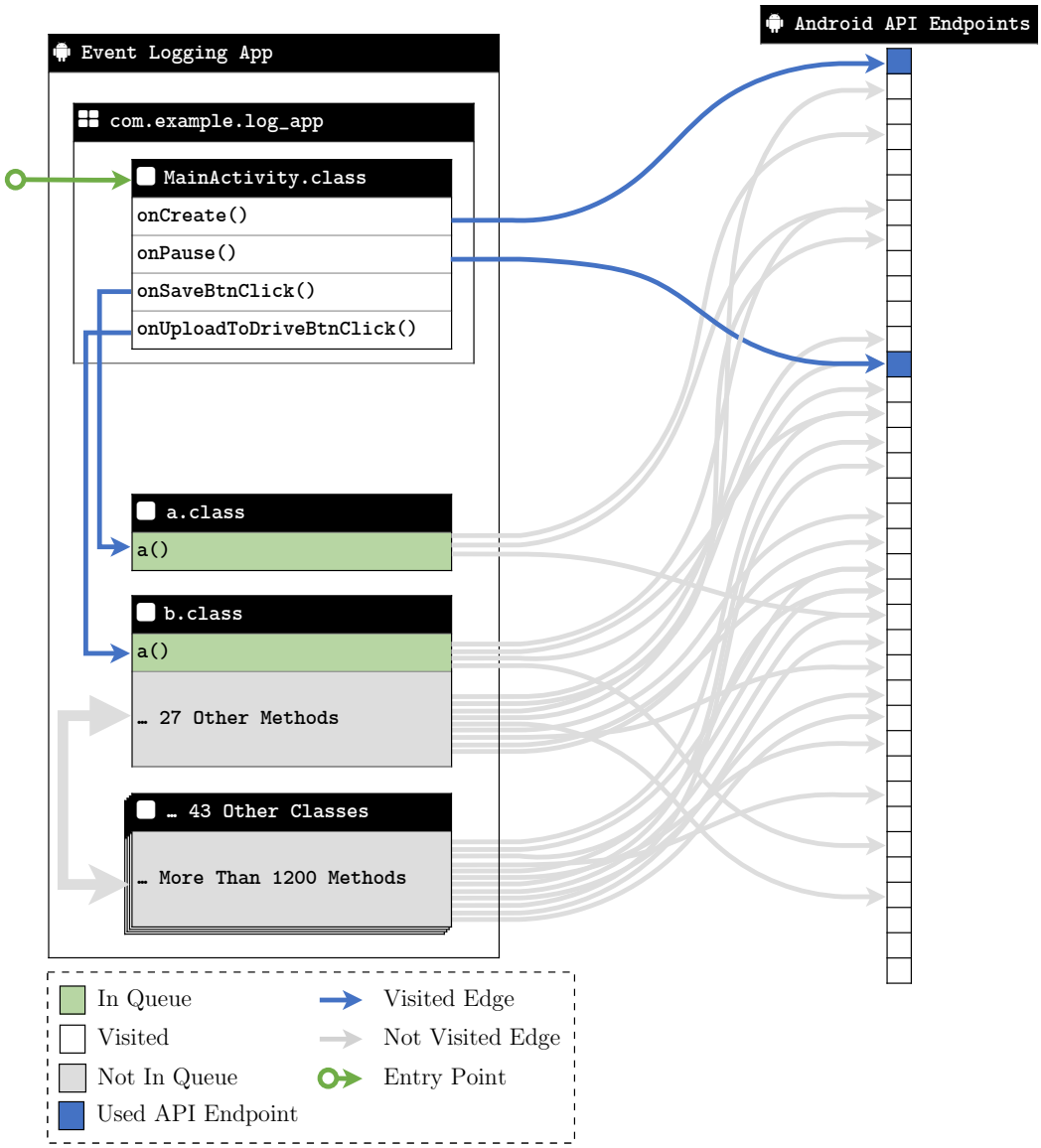


Figure 4.7: Sample application traversal: Iteration 4

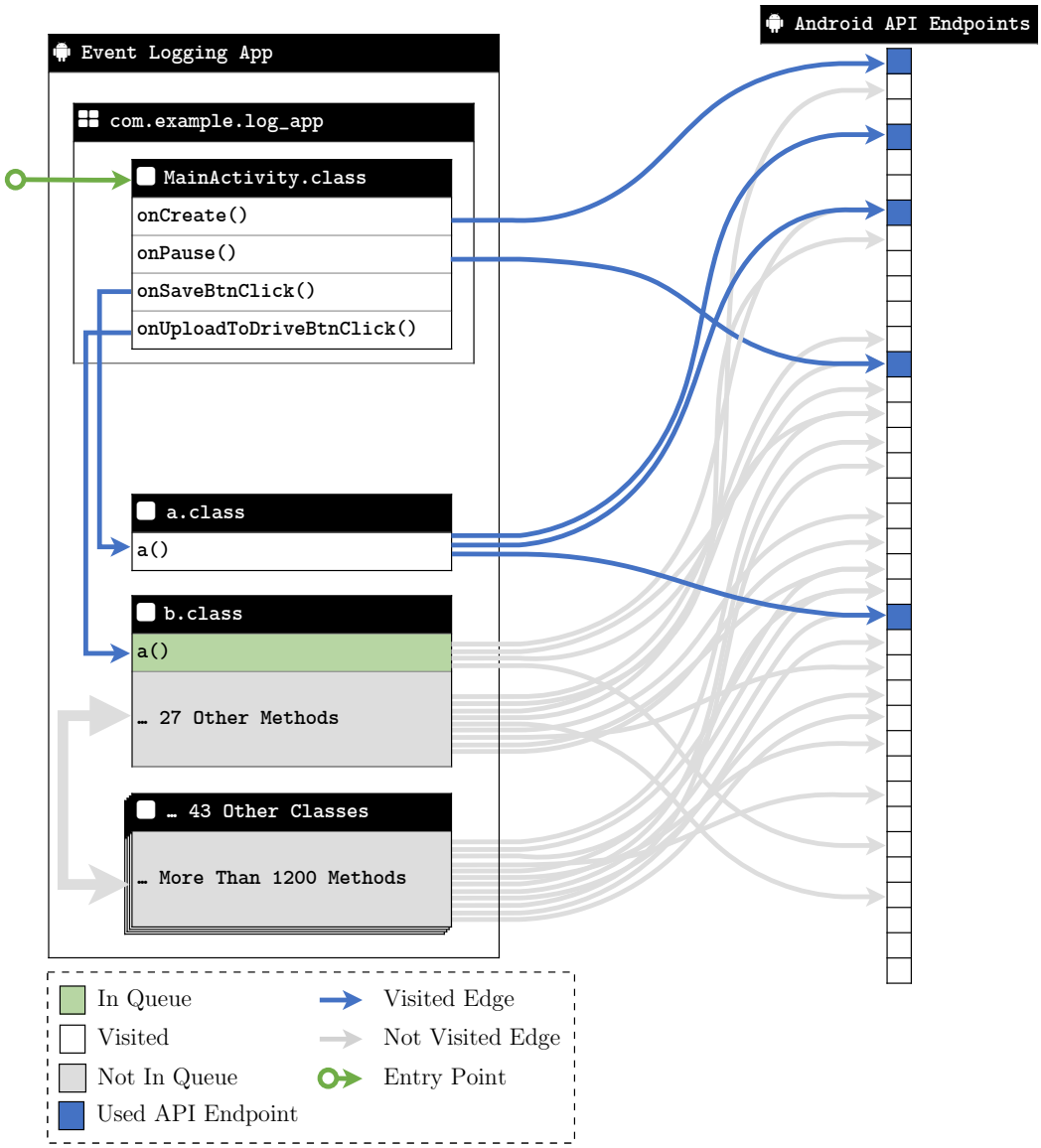


Figure 4.8: Sample application traverse: Iteration 5

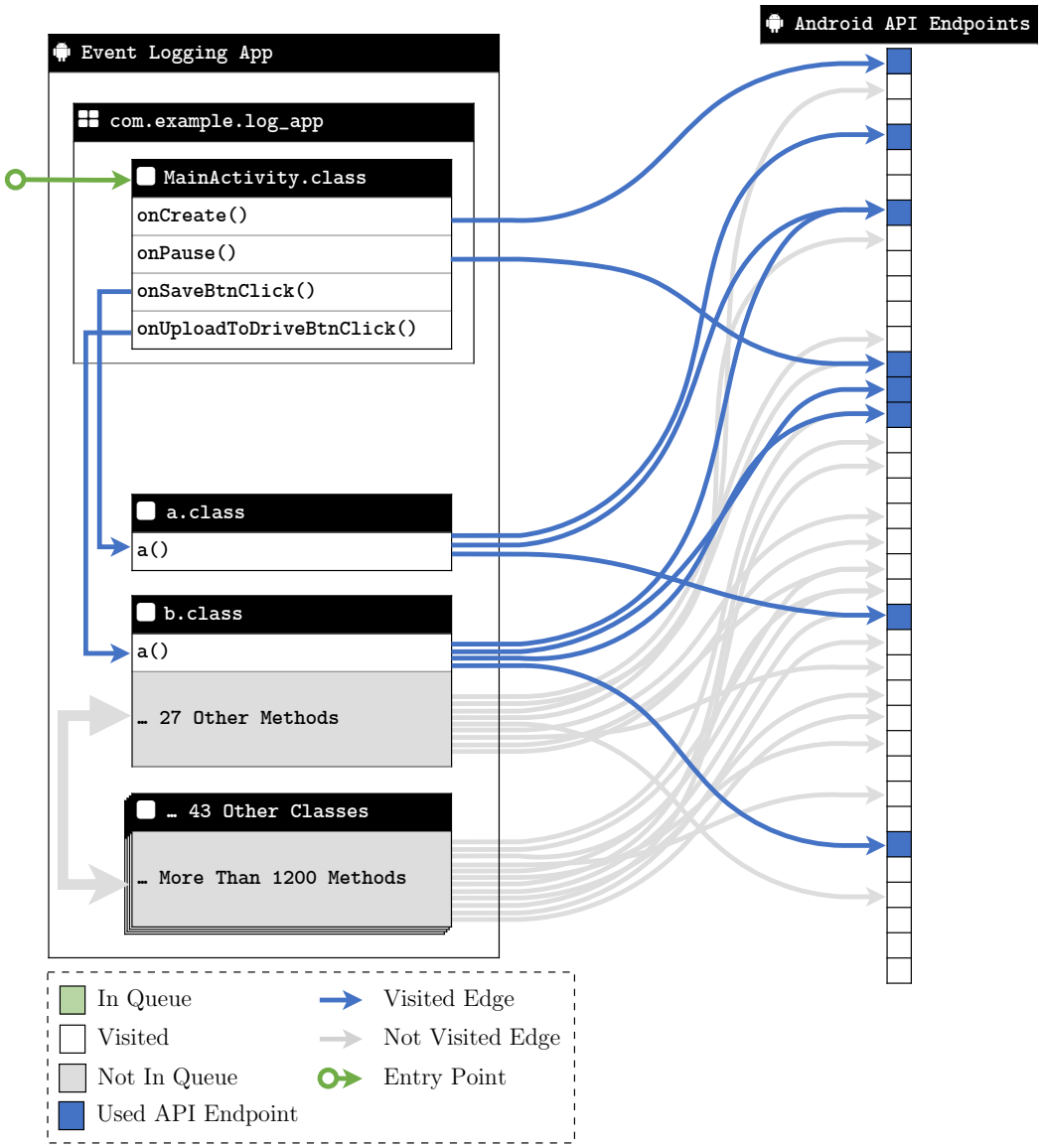


Figure 4.9: Sample application traverse: Iteration 6

As demonstrated in Figure 4.9, when the last iteration is finished, all of the reachable Android API calls from the entry points are detected. There can be a huge number of Android API calls from unused parts of third party libraries (shown as grey arrows) which would cause a lot of noise in the results if we included all of the

classes in the application in our analysis. Our method reduces this noise radically by analyzing the reachable methods in the applications and excluding the unreachable ones.

### 4.2.3 Report Generation

In the final stage of our approach, we output a list of Android API endpoints that are reachable from the application's entry points. Listing 4.3 shows a part of an API use report. Each line consists of the fully qualified name of the class in which the API endpoint appears, followed by a colon and the name of the API method.

**Listing 4.3** Sample of generated report

```
...
android.view.ActionMode:<init>
android.view.ActionProvider:hasSubMenu
android.view.ActionProvider:isVisible
android.view.ActionProvider:onCreateActionView
android.view.ActionProvider:onPerformDefaultAction
android.view.ActionProvider:onPrepareSubMenu
android.view.ActionProvider:overridesItemVisibility
android.view.ActionProvider:setVisibilityListener
android.view.CollapsibleActionView:onActionViewCollapsed
android.view.CollapsibleActionView:onActionViewExpanded
android.view.ContextMenu:add
...
```

## 4.3 Conclusion

This chapter has described the details of our approach in detail and how we implemented each step. We used a sample application to demonstrate each iteration. In order to explore our research questions, we used this approach in an experiment to

compare Android applications developed by Google to similar non-Google applications, which we discuss in the next chapter.

# Chapter 5

# Experiment

The goal of this thesis is to explore differences in how the Android API is used by developers inside and outside Google. To explore Android API use in these two groups, we selected two sets of applications, one consisting of popular applications developed by Google and the other consisting of corresponding similar applications developed by other groups.

## 5.1 Application Selection Process

In this section, we discuss our constraints and process for selecting applications for our experiment.

### 5.1.1 Popularity

Google's Android applications database [12] in the Google Play Store shows that, as of the day of writing this thesis, Google has published 127 Android applications. The number of installs of these applications varies from less than 100 for *Local Services ads by Google* to more than 50 billion for *Google Play Services*. Google applications with less than one hundred million (100,000,000) installs are usually designed for a

specific event, device or a small group of users (e.g. *Google I/O*,*Cardboard*, *Mobile Ads Showcase*). These applications might not be as up to date and well maintained as the popular applications. Therefore, we decided to analyze applications with more that one hundred million installs.

### 5.1.2 Existence of Similar Applications

For this experiment, we must have two groups of applications, one containing applications developed by Google and the other one containing corresponding applications developed by other companies. Considering the popularity constraint, we have 35 applications which are fairly popular according to their number of installs. In order to keep our comparison fair, for each application developed by Google, we found a similar application and put it in the other group.

In order to find similar applications we used the Google search engine. Because users normally use Google's search engine to find alternative applications for a particular purpose, it can be a reliable source of information for our purpose. For each application, we tried searching "*[Google Application's Name] vs*" and waited for Google to suggest the similar application. After recording the suggestion, we checked the suggested application manually by reading through the application description in the Google Play Store and checking the screenshots to make sure they have similar functionalities. For some of the applications such as *Gmail*, which are also available in other platforms, we had to add the keyword "Android" at the beginning of our search query in order to find similar applications on the Android platform. This method revealed similar applications for 21 of Google's most popular applications.

For 14 applications, despite their popularity, we could not find similar applications

with the same set of functionalities. For example, we could not find any similar application for *Google Play Services*, *Google Earth*, and *Google Search Bar*. In most cases, the reason was that the applications were related to Google's services for Android devices.

### 5.1.3 Convertability to JAR File

As mentioned in the previous chapter, we used Dex2Jar [7] for converting DEX files to JAR in order to make them analyzable by JAR file tools such as the BCEL library. In two cases, *Google Hangouts* and *Google launcher*, Dex2Jar failed to convert DEX files to JAR files and we had to remove those applications from our analysis.

## 5.2 Selected Applications

After applying all of our constraints and finding the similar applications, we ended up with 38 applications, 19 Google applications and 19 corresponding non-Google similar applications, shown in Table 5.1.

Table 5.1: Selected applications

| # | Application Name | Developer | Downloads | Size |
|---|---|---|---|---|
| 1 | Google Play Books | Google | 1+ Billion | 10.9 MB |
| | Amazon Kindle | Amazon | 100+ Million | 57.6 MB |
| 2 | Google Photos | Google | 1+ Billion | 37.2 MB |
| | Flickr | Yahoo | 10+ Million | 20.3 MB |
| 3 | Maps - Navigation & Transit | Google | 1+ Billion | 27.3 MB |
| | Waze | Waze | 100+ Million | 57.2 MB |
| 4 | Google+ | Google | 1+ Billion | 20.6 MB |
| | Facebook | Facebook | 1+ Billion | 64.2 MB |
| 5 | Google Chrome: Fast & Secure | Google | 1+ Billion | 63.8 MB |
| | Firefox | Mozilla | 100+ Million | 40.6 MB |
| 6 | Google Drive | Google | 1+ Billion | 22.5 MB |
| | Dropbox | Dropbox, Inc. | 500+ Million | 60.1 MB |
| 7 | YouTube | Google | 1+ Billion | 23.7 MB |
| | Vimeo | Vimeo Inc. | 10+ Million | 13.1 MB |
| 8 | Google Play Movies & TV | Google | 1+ Billion | 14.3 MB |
| | Netflix | Netflix, Inc. | 100+ Million | 18.5 MB |
| 9 | Gmail | Google | 1+ Billion | 19.4 MB |
| | Outlook | Microsoft | 100+ Million | 42.7 MB |
| 10 | Google Play Newsstand | Google | 1+ Billion | 12.3 MB |
| | Flipboard | Flipboard | 700+ Million | 7.6 MB |
| 11 | Google Play Music | Google | 1+ Billion | 16.5 MB |
| | Spotify | Spotify Ltd. | 100+ Million | 42.3 MB |
| 12 | Google Translate | Google | 500+ Million | 16.2 MB |
| | Microsoft Translator | Microsoft | 5+ Million | 52.5 MB |
| 13 | Gboard - the Google Keyboard | Google | 500+ Million | 23.7 MB |
| | Swype Keyboard | Nuance, Inc | 1+ Million | 28.0 MB |
| 14 | Google Keep | Google | 100+ Million | 12.7 MB |
| | Evernote | Evernote | 100+ Million | 31.1 MB |
| 15 | Google Slides | Google | 100+ Million | 47.2 MB |
| | Microsoft PowerPoint | Microsoft | 100+ Million | 60.7 MB |
| 16 | Google Duo | Google | 100+ Million | 12.7 MB |
| | Skype | Microsoft | 1+ Billion | 30.0 MB |
| 17 | Android Messages | Google | 100+ Million | 18.3 MB |
| | GO SMS Pro | GOMO Apps | 100+ Million | 7.94 MB |
| 18 | Google Docs | Google | 100+ Million | 33.7 MB |
| | Microsoft Word | Microsoft | 100+ Million | 67.2 MB |
| 19 | Google Sheets | Google | 100+ Million | 40.6 MB |
| | Microsoft Excel | Microsoft | 100+ Million | 61.7 MB |

## 5.3 Data Extraction Process

In order to extract API use data from the applications, we obtained the APK files from APKpure.com, which is a mirror of the Google Play Store that allows downloading of APK files directly from the browser. We placed applications in two directories: *Google* and *Others*, and ran a bash script to extract data from each application. The script had 4 steps:

- **Extracting classes.dex files:** For each APK file, we used *unzip* to extract the application's *classes.dex* file. The *classes.dex* file is located in the main directory of each application.
- **Converting DEX files to JAR files:** In order to convert DEX files to JAR we used *dex2jar*[7].
- **Extracting AndroidManifest.xml files:** Each Android application has a manifest file which provides basic information about the application. There is an encoded version of the *AndroidManifest.xml* in each APK file. In order to make them parseable, we used *apktool*[5].
- **Listing Android API calls:** We developed a Java application that uses the *BCEL library*[4] to explore classes inside JAR files. This application takes a JAR file and an Android manifest file as input. Using our transitive closure trace method, it returns a list of the Android API endpoints the Android application is using. This list is sorted alphabetically and duplicated API calls are removed.

Our approach produces a list of API calls for each application. We merged all of the reports from individual applications into a single file and removed duplicated lines

to generate a list of all Android API endpoints in our dataset. 13,278 unique Android API endpoints were found in the 38 selected applications. Afterwards, we combined all of the reports and the list of Android API endpoints into a table. In this table, each column represents an application in our dataset, each row represents an Android API endpoint and each cell has a boolean value indicating whether each application is using the API endpoint. Figure 5.1 shows our process of producing a table from individual reports. In the produced table, we have two groups of columns (Google and others) divided by a grey column. In each cell, black background represents that the related application is using the API endpoint. The produced report is available online [9] for future studies.

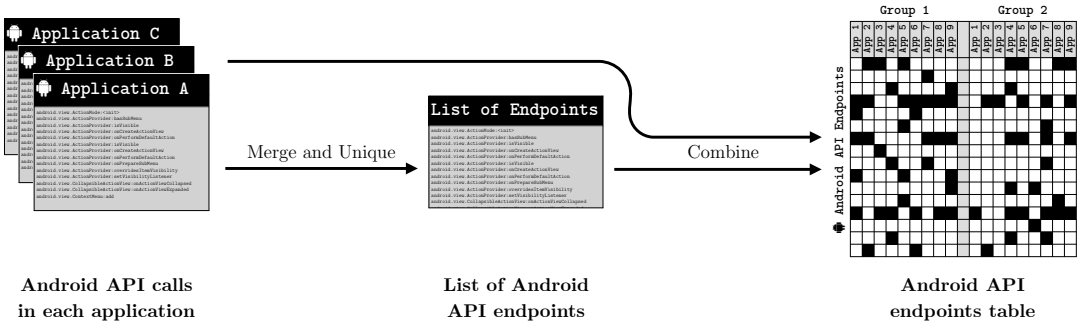


Figure 5.1: The process of combining the API use reports to produce a table

Figure 5.2 shows the histogram of frequency of how many applications are using each API endpoint. By looking at the histogram of our data, we noticed that there are a huge number of API endpoints that were detected in only one or two applications. We decided to remove these endpoints as outliers. Figure 5.3 shows the histogram after removing the outliers. 6,668 unique Android API endpoints remained in our

dataset.

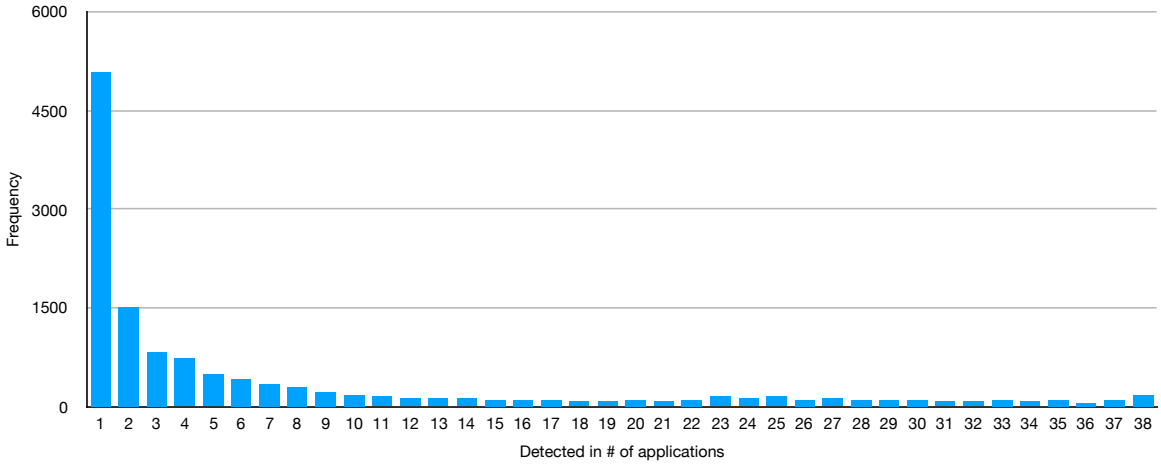


Figure 5.2: Histogram of data before removing the outliers

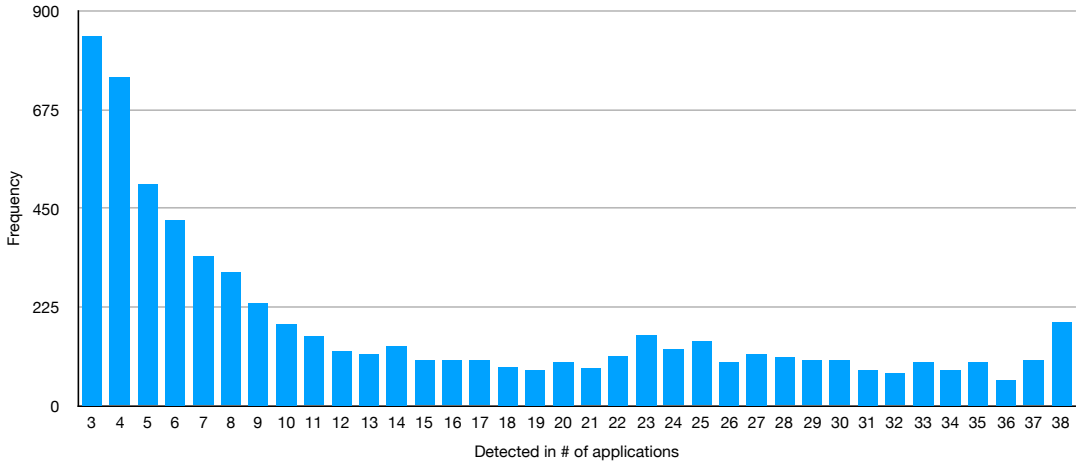


Figure 5.3: Histogram of data after removing the outliers

In the next section, we use the extracted data to answer our research questions.

## 5.4 Analysis and Results

In this experiment, we focused on four research questions which we will discuss in the following subsections.

### 5.4.1 Do Google applications use Android differently from other applications?

Yes, we found 755 endpoints that were used either only by Google applications or only by non-Google applications. Figure 5.4 shows 60 API endpoints that were used most differently by Google and non-Google applications. Black in the columns in front of API endpoints shows that the application has used the endpoint at least once.

| # | API Endpoint | Google's Applications | Non-Google Applications |
|---|---|---|---|
| 1 | android.webkit.WebViewClient:onReceivedError(O) | | |
| 2 | android.app.ApplicationErrorReport:<init>(O) | | |
| 3 | android.webkit.SslErrorHandler:cancel(M) | | |
| 4 | android.graphics.drawable.Drawable:setTint(M) | | |
| 5 | android.graphics.drawable.Drawable:setHotspotBounds(M) | | |
| 6 | android.graphics.drawable.Drawable:isAutoMirrored(M) | | |
| 7 | android.graphics.Bitmap$Config:toString(M) | | |
| 8 | android.content.pm.PackageManager$NameNotFoundException:printStackTrace(M) | | |
| 9 | android.app.ProgressDialog:requestWindowFeature(M) | | |
| 10 | android.app.Notification$Builder:setVibrate(M) | | |
| 11 | android.widget.TextView:getContentDescription(M) | | |
| 12 | android.webkit.CookieManager:removeExpiredCookie(M) | | |
| 13 | android.net.wifi.WifiInfo:getMacAddress(M) | | |
| 14 | android.graphics.drawable.Drawable:setLayoutDirection(M) | | |
| 15 | android.app.Notification$Builder:setNumber(M) | | |
| 16 | android.app.Notification$Builder:setLights(M) | | |
| 17 | android.app.Notification$Builder:setFullScreenIntent(M) | | |
| 18 | android.widget.FrameLayout:dispatchThawSelfOnly(M) | | |
| 19 | android.widget.FrameLayout:dispatchFreezeSelfOnly(M) | | |
| 20 | android.graphics.drawable.Drawable:getOutline(M) | | |
| 21 | android.graphics.drawable.Drawable:getDirtyBounds(M) | | |
| 22 | android.graphics.drawable.Drawable:getColorFilter(M) | | |
| 23 | android.graphics.drawable.Drawable:canApplyTheme(M) | | |
| 24 | android.graphics.drawable.Drawable:applyTheme(M) | | |
| 25 | android.content.ContentProviderClient:call(M) | | |
| 26 | android.app.Notification$Builder:setPublicVersion(M) | | |
| 27 | android.app.Notification$Builder:setGroupSummary(M) | | |
| 28 | android.app.Notification$Builder:setGroup(M) | | |
| 29 | android.app.Notification$Builder:setExtras(M) | | |
| 30 | android.app.Notification$Action$Builder:setAllowGeneratedReplies(M) | | |
| 31 | android.app.Notification$Action$Builder:addRemoteInput(M) | | |
| 32 | android.app.Notification$Action$Builder:addExtras(M) | | |
| 33 | android.widget.TextView:setTransformationMethod(M) | | |
| 34 | android.widget.TextView:announceForAccessibility(M) | | |
| 35 | android.widget.RelativeLayout:setPadding(M) | | |
| 36 | android.widget.RelativeLayout:setBackgroundDrawable(M) | | |
| 37 | android.widget.ImageView:getViewTreeObserver(M) | | |
| 38 | android.webkit.WebView:saveState(M) | | |
| 39 | android.webkit.WebView:restoreState(M) | | |
| 40 | android.view.accessibility.AccessibilityRecord:setScrollable(M) | | |
| 41 | android.view.accessibility.AccessibilityRecord:setScrollY(M) | | |
| 42 | android.view.accessibility.AccessibilityRecord:setScrollX(M) | | |
| 43 | android.view.View:isVerticalScrollBarEnabled(M) | | |
| 44 | android.view.View:isHorizontalScrollBarEnabled(M) | | |
| 45 | android.view.View:computeVerticalScrollExtent(M) | | |
| 46 | android.view.View:computeHorizontalScrollRange(M) | | |
| 47 | android.view.View:computeHorizontalScrollExtent(M) | | |
| 48 | android.content.pm.ShortcutManager:reportShortcutUsed(M) | | |
| 49 | android.app.Notification:getGroup(M) | | |
| 50 | android.app.Notification$Builder:setUsesChronometer(M) | | |
| 51 | android.app.Notification$Builder:setTimeoutAfter(M) | | |
| 52 | android.app.Notification$Builder:setSortKey(M) | | |
| 53 | android.app.Notification$Builder:setShortcutId(M) | | |
| 54 | android.app.Notification$Builder:setRemoteInputHistory(M) | | |
| 55 | android.app.Notification$Builder:setGroupAlertBehavior(M) | | |
| 56 | android.app.Notification$Builder:setCategory(M) | | |
| 57 | android.app.Notification$Builder:setBadgeIconType(M) | | |
| 58 | android.app.Notification$Builder:addPerson(M) | | |
| 59 | android.app.Dialog:onDetachedFromWindow(O) | | |
| 60 | android.widget.TextView:getTop(M) | | |

Figure 5.4: 60 most differently used endpoints by Google and non-Google applications

**Google specific endpoints:** We detected 444 endpoints that were used by Google applications and were never used by non-Google applications. On average, each of these endpoints was used in 24% of Google applications.

**Non-Google specific endpoints:** We detected 311 endpoints that were used by non-Google applications and were never used by Google applications. On average, each of these endpoints was used in 22% of non-Google applications.

For example, in Google applications we could not detect any API calls to endpoints in the *ProgressDialog* and *CameraParameters* classes. On the other hand, endpoints in the *NotificationChannel* and *HealthStats* classes were popular in Google applications and never used by others.

### 5.4.2 To what extent do Google applications use Android differently from other applications?

In order to explore how differently the two groups of applications use Android, we searched for endpoints that were popular in one group (used by at least 50% of the applications) and not popular in the other group (used by at most 50% of the applications). By applying these filters to our dataset, we obtained the following results:

**In Google applications:** We found 556 endpoints that were popular in Google applications and not very popular in non-Google applications, which is 8.3% of the unique endpoints.

**In non-Google applications:** We found 256 popular endpoints that were not used as often in Google applications, which is 3.8% of the unique endpoints.

This shows that Google and non-Google applications are using at least 812 endpoints (12.1% of the Android API) very differently.

### 5.4.3 Do Google applications adopt new capabilities more quickly than other applications?

To explore this question, we needed information about when each API endpoint was added to Android. The Android API documentation offers *"Added in API level"* for each endpoint. As of the day of conducting this experiment, Android has released 26 API levels. A higher API level shows that the endpoint was added to the Android API more recently.

This information is not present in the byte-code of the applications. Thus, we had to manually search for each endpoint in the Android API documentation and add the information to our dataset. As this is a time-consuming process and it requires human involvement, we decided to focus on the 60 (30 for each group) endpoints that Google applications and non-Google applications used most differently.

**In Google applications:** Google applications adopt new API endpoints faster than non-Google applications. Figure 5.5 shows the histogram of "Added in API level" for 30 API endpoints that were used only by Google applications (the most different ones). The average "Added in API level" in this group is 15.9 which shows that Google applications use a lot of new API endpoints that non-Google applications have not started using yet.

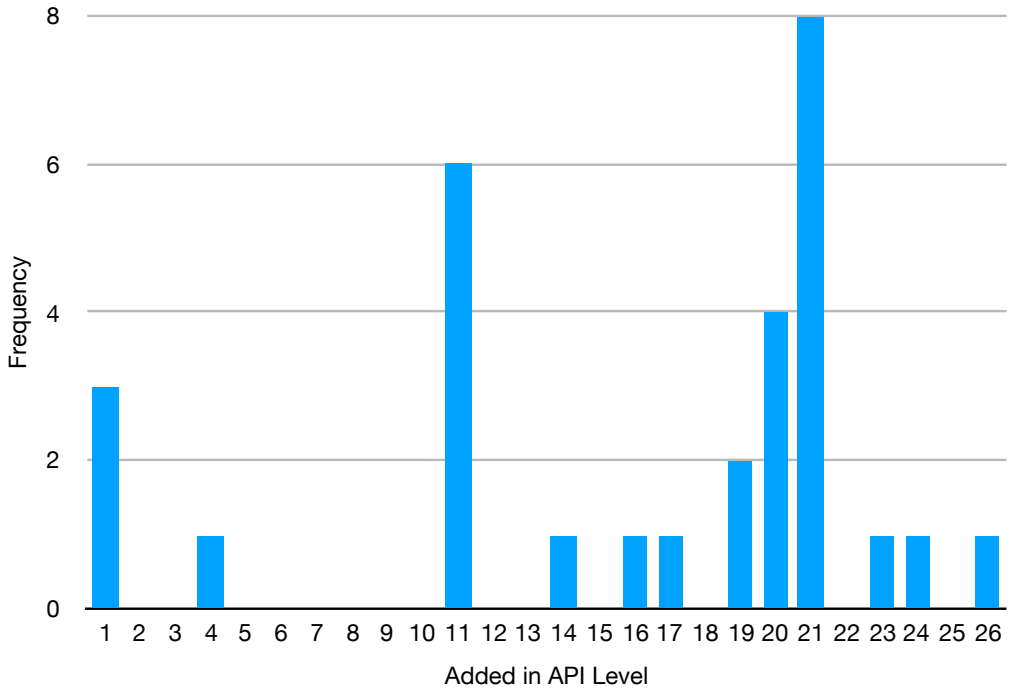


Figure 5.5: Histogram of "Added in API level" in top 30 endpoints used only by Google applications

**In non-Google applications:** Our observations suggest that non-Google applications are very slow in adopting new API endpoints compared to Google applications. Figure 5.5 shows the histogram of "Added in API level" for 30 API endpoints that were used only by non-Google applications (the most different ones). Most of the endpoints that were only used by non-Google applications were introduced in Android API level 1. The fact that Google applications have abandoned these API calls might suggest that they are replaced by new endpoints in Google applications. The average "Added in API level" in this group is 1.3 which shows that non-Google applications use a lot of old API endpoints that are not used by Google applications anymore.

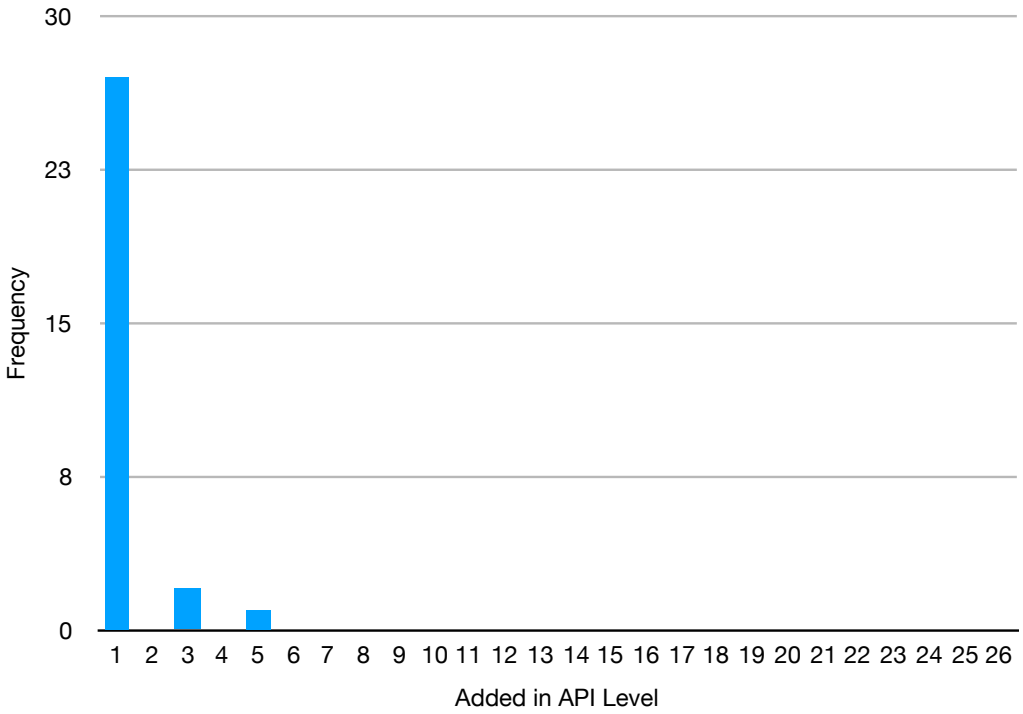


Figure 5.6: Histogram of "Added in API level" in top 30 endpoints used only by non-Google applications

Our general observation of the data suggests the same results. Google applications are using a lot of newly added capabilities of Android OS. For example, *Picture-in-Picture mode* and *Notification channels* were introduced in API level 26. Most of Google applications are using these feature, while no non-Google applications in our dataset are using them.

### 5.4.4 Do Google applications abandon deprecated capabilities more quickly than other applications?

For deprecated endpoints, the Android documentation has a *"Deprecated"* note in the endpoint's section. We collected the information for the 60 most differently used endpoints in our dataset. In Google applications, we found two popular API endpoints

that were marked as deprecated in Android documentation. On the other hand, in non-Google applications, we found 6 popular endpoints that were deprecated. Further manual inspection showed that there are also a large number of deprecated endpoints in the less popular endpoints in non-Google applications. For example, *ProgressDialog* and *CameraParameters* were deprecated and Google applications are not using them any more, while they are still popularly used in non-Google applications.

## 5.5 Conclusion

In this chapter, we discussed our process of selecting Android applications in order to conduct an experiment to see how differently the two groups of applications (Google and non-Google) use the Android API. These two groups use a lot of endpoints in a similar manner. At the same time, there were some endpoints which were used in one group frequently but not used in the other group at all. We explored four research questions and we concluded that Google applications and non-Google applications use the Android API quite differently. Additionally, a large number of endpoints are popular in one group while they are not used frequently in the other group. Finally, Google applications appear to more quickly adopt new features, and more quickly abandon deprecated ones. Although further studies are required, our observations suggest two possible conclusions: either other companies are failing to keep up with Android API changes, or Google is changing Android according to its own requirements and other companies do not find the newly added endpoints useful.

# Chapter 6

# Conclusion and Future Work

This thesis has presented a study of API use in Android applications. Specifically, we have compared how differently Google and non-Google applications use the Android API. This thesis began in Chapter 1 by a brief introduction, the goal of this thesis in addition to its contributions and outline.

Chapter 2 gave background information about Java, Android, Android applications and challenges in API analysis in Android applications. Additionally, it presented some pervious studies in Android API analysis and third-party library detection in Android applications.

Chapter 3 served as an overview of our study. It gave a high-level description of our process including an overview of our new method for extracting API use information from Android applications and our experiment for comparing Google and non-Google applications.

In Chapter 4, we proposed a new method for analyzing API use in Android applications and discussed how it deals with third-party library and obfuscation challenges.

Chapter 5 focused on the experiment part of our study. It presented our criteria

for application selection and the API data extraction steps in our experiment. Additionally, this chapter explored the results, discussed our observations and answered our research questions.

## 6.1 Contributions

**A new method for analyzing API use in Android Applications:** We have presented a new approach for analyzing API use in Android applications that works on most Android applications available in the Android markets. This method is robust to common obfuscation techniques in Android applications and it removes the noise produced by the unused part of third-party libraries.

**API use data of 38 popular Android applications:** We have extracted API use information from 38 Android applications, including 19 Google applications and 19 non-Google applications. This information is available online for future studies [9].

**A comparison of Android API use in Google and non-Google applications:** We have examined the extracted API information and found interesting differences between how Google and non-Google applications use the Android API.

## 6.2 Threats to Validity

While we tried to make our method for API analysis in Android applications as precise as possible, we understand that there are limitations to its precision.

First, we could not find a perfect way to accurately predict the dispatched method in dynamically bound method calls. While we are confident in our estimates, in some cases we might include the wrong method and miss the method that actually gets called at runtime.

Second, it is a common practice in Android development to extend a class from the Android API and override some of its methods, or implement an interface form Android API and pass an instance of that class to the Android framework in order to get executed in specific situations. As there are no direct calls to these methods in the bytecode of applications, we decided to include all of the methods in instantiated objects. In some cases this decision can result in including some unused parts of the applications.

Third, we understand that some parts of Android application code might not be reachable from entry points but may nevertheless get executed by Android. For example, customized view elements that developers mention in layout XML files will be executed but they might not be reachable by following the method calls from entry points of applications.

## 6.3 Future Work

Our method for analyzing API use in Android applications has countless other applications such as malware detection, energy usage analysis, and other analyses. In comparison to existing methods for dealing with obfuscation and third-party libraries, most of the previous API analysis studies can potentially have different results using this approach.

Another extension of this work might be to perform a similar comparison between other groups of applications to find out how different groups of applications are using Android differently.

# Bibliography

[1] Android API Change Statistics. https://developer.android.com/sdk/api_diff/23/changes.html. Last accessed on February 25, 2018.

[2] Android library statistics - AppBrain. http://www.appbrain.com/stats/libraries/. Last accessed on February 25, 2018.

[3] Android SDK Platform Release Notes. https://developer.android.com/studio/releases/platforms.html. Last accessed on February 25, 2018.

[4] Apache Commons BCEL ™. https://commons.apache.org/proper/commons-bcel/. Last accessed on February 25, 2018.

[5] Apktool - A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/. Last accessed on February 25, 2018.

[6] App manifest file - android developer's guide. https://developer.android.com/guide/topics/manifest/application-element.html. Last accessed on February 25, 2018.

[7] dex2jar - tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar. Last accessed on February 25, 2018.

[8] Enable multidex for apps with over 64k methods. https://developer.android.com/studio/build/multidex.html. Last accessed on February 25, 2018.

[9] Extracted API data from popular Google and non-Google Applications. https://docs.google.com/spreadsheets/d/1sfsaq9U805gZKwQ8-kf-T0saUAmDskxOoUVtSz_uynU/edit?usp=sharing. Last accessed on February 25, 2018.

[10] F-Droid - Free and Open Source Android App Repository. https://f-droid.org. Last accessed on February 25, 2018.

[11] Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. Last accessed on February 25, 2018.

[12] Google LLC - Android developer info on AppBrain. https://www.appbrain.com/dev/Google+LLC/. Last accessed on February 25, 2018.

[13] LibRadar - A detecting tool for 3rd-party libraries in Android apps. https://github.com/pkumza/LibRadar. Last accessed on February 25, 2018.

[14] Proguard - the open source optimizer for java bytecode. https://www.guardsquare.com/en/proguard. Last accessed on February 25, 2018.

[15] Shams Abubakar Azad. *Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations.* PhD thesis, Concordia University, 2015.

[16] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 356–367, New York, NY, USA, 2016. ACM.

[17] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 653–656, New York, NY, USA, 2016. ACM.

[18] Mojica, Israel J and Adams, Bram and Nagappan, Meiyappan and Dienst, Steffen and Berger, Thorsten and Hassan, Ahmed E. A large-scale empirical study on software reuse in mobile apps. *IEEE software*, 31(2):78–86, 2014.

[19] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122. IEEE, 2012.